Enabling Usable and Performant Trusted Execution

by

Ofir Weisse

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2020

Doctoral Committee:

Assistant Professor Baris Kasikci, Co-Chair
Professor Thomas F. Wenisch, Co-Chair
Research Professor Peter Honeyman
Assistant Professor Jean-Baptiste Jeannin

Ofir Weisse

oweisse@umich.edu

ORCID iD: 0000-0001-7564-6007

© Ofir Weisse 2020

To my parents, Rachel & Yossi

ACKNOWLEDGEMENTS

A PhD requires more than just overcoming intellectual challenges. Being a grad student in a place far away from home, speaking a language which is not my native tongue would have been significantly more challenging, without help and support from countless people. This help is typically not academical but can be crucial in order to survive, for example, the Michigan winter.

I want to thank Peter Honeyman for welcoming me to his home from day -1 (AKA visit day) and being a mentor for both my professional as well as personal life. While we may sometimes have different perspectives on various subjects, it has been a pleasure visiting his office almost every week of my PhD.

I want to thank my colleagues whom I worked with during my first steps in the PhD program: Timothy Trippel, Matthew Hicks, and Jeremy Erickson. Step 10 is never possible without steps 1,2, and 3.

Finishing this dissertation in a meaningful and impactful way would not have been possible without the guidance and, at times, mental support, from my advisors Baris Kasikci and Tom Wenisch (aka TFW). They took a chance with me when I was in between labs and for that I am grateful. They provided me with both academic-independence and (mostly useful) criticism. These were administered typically at the appropriate times. Special thanks to Jean-Baptiste for agreeing to jump in on my research and joining my dissertation committee.

I'd also like to acknowledge my lab mates (and not just because they insisted) for providing the proper intellectual environment to bounce ideas, and for tolerating me in the same office space. While they incessantly complained about me being too loud or interrupting their work, I still enjoyed their company. Being surrounded by ambitious people leaves one no other choice than pecking some aspirations of his own.

Finally, I'd like to thank the professors at the university who had an open door for me: Jason Flinn, Mosharaf Chowdhury, and Manos Kapritsos. Hearing their take on academic and personal life was an important part of my support system during the PhD program.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Table

ABSTRACT

A plethora of major security incidents—in which personal identifiers belonging to hundreds of millions of users were stolen—demonstrates the importance of improving the security of cloud systems. To increase security in the cloud, where resource sharing is the norm, we need to rethink existing approaches.

This thesis analyzes the practicality and security of trusted execution technologies as the cornerstone of secure software systems, to better protect user data and privacy.

Trusted Execution Environments (TEE), such as Intel SGX, have the potential to minimize the Trusted Computing Base (TCB), but they also introduce many challenges for adoption. Among these challenges are their significant impact on application performance and substantial effort required to migrate legacy systems to run on these secure execution technologies. Other challenges include managing trustworthy state across a distributed system and ensuring that individual machines are resilient to micro-architectural attacks.

In this thesis, I first characterize the performance bottlenecks imposed by SGX and suggest optimization strategies. I then address two main adoption challenges for existing applications: managing permissions across a distributed system and scaling the SGX mechanism for proving authenticity and integrity. I then analyze the resilience of trusted execution technologies to micro-architectural attacks on speculative execution, which put cloud infrastructure at risk. This analysis revealed a devastating security flaw in Intel processors, known as Foreshadow/L1TF. Finally, I propose a new architectural design for out-of-order processors that defeats all known speculative execution attacks.

To characterize the performance bottlenecks, unique to Intel SGX TEE technology, I devise a set of microbenchmarks. These benchmarks shed light on the impact of isolating trusted code inside an *enclave*—SGX secure execution context—and the overhead of using encrypted memory, showing that SGX may degrade performance by up to 79%. I then suggest an optimization strategy, HotCalls, that alleviates the performance bottlenecks created by SGX. HotCalls boosts throughput by 2.6-3.7x, and reduces application latency by 62-74%. Intel adopted the HotCalls concept into their SDK.

Improving raw performance is only the first step in realizing a practical distributed secure system. Specifically, In this part of my thesis I present two frameworks: TEE-SERVICE and *SovereignTEE*. TEE-SERVICE enables managing access permissions on a distributed secure system. *SovereignTEE* shifts trust management from the hardware vendor (Intel) to the cloud provider while maintaining SGX security guarantees.

In the next part of this thesis, I show how all three security pillars of SGX TEE can be broken using speculative execution attacks. When the first speculative execution attacks—Meltdown and Spectre—became public knowledge, the security community hypothesized that Intel SGX cannot be penetrated using these attacks. However, my work, as acknowledged by Intel and currently known under the name of "L1 Terminal Fault", shows how a malicious SGX enclave can break all three security properties provided by SGX TEE: confidentiality, secure storage, and proof of integrity.

Finally, in the last part of this thesis, I present *NDA*: a new architecture design that defeats all known variants of speculative execution attacks. Existing mitigations for Spectre and Meltdown attacks focus on ad-hoc mitigations to block specific mechanisms abused by the currently known exploits. To avoid the ceaseless arms race of blocking one attack after the other, I propose *NDA*, a technique for restricting speculative data propagation. I describe a design space of *NDA* variants that differ in the constraints they place on the dynamic scheduling and the classes of speculative execution attacks they prevent.

CHAPTER I

Introduction

Enterprises are increasingly shifting services that manage sensitive data, such as medical [22, 31, 93, 189] and financial [21, 89, 94] records, from on-premise platforms to public clouds in order to reduce management and provisioning costs. However, the shift to public clouds entails substantial risks. Cloud providers colocate virtual machines from mutually untrusting parties on the same hardware; a flaw in the infrastructure can easily lead to data compromise [67, 71, 245, 262]. Moreover, service owners—the cloud direct customers—must trust the cloud operator itself; public reports contain numerous examples where malicious actors [57, 68, 233, 244, 258] or legal action [20, 92, 97, 191] have compromised private data. As such, despite potential cost savings, many enterprises remain hesitant to use public clouds [33, 76, 192, 229, 259, 274].

The current trusted computing base (TCB) for public cloud services is too large [74, 75, 80, 110, 230, 235, 283] and includes essentially the entire software stack: a vulnerability in either the application, the OS, or the hypervisor can allow an attacker to access sensitive data and violate security requirements. Moreover, a malicious employee at the data center site may mount physical attacks [10], circumventing any software protection.

1.1 Trusted Execution Environments

Trusted Execution Environment (TEE) technologies, such as Intel SGX [29, 113, 183], AMD SEV [1, 26], and ARM TrustZone [19], offer a practical approach to minimizing the TCB. Instead of requiring external hardware, TEEs extend general-purpose processors with mechanisms to create a secure context, isolated from potentially malicious applications, OS, and hypervisor. This solution provides the benefits of hardware-backed security with the performance of a general-purpose processor. The TEE secure context provides confidentiality and integrity guarantees enforced in hardware, so that even the hardware owner (e.g., the cloud provider) cannot falsify the integrity of the computation or exfiltrate private data.

In short, TEEs allow a cloud provider to resell compute and storage within a "black box" that neither the service provider nor other co-scheduled tenants can open. I focus my study on Intel SGX, which is the most prominent production-grade TEE technology for server-class processors. Despite recent attacks on Intel SGX[1] [121, 269, 282]—which were immediately patched [25, 95, 187, 266]—SGX is the most prominent TEE technology adopted by commercial companies [10, 30, 82, 116, 206].

In the first three chapters of this thesis (II-IV), I address the key challenges for enabling trustworthy secure cloud computing, including performance impact, the practicality of deployment, and security. While TEEs have the potential to minimize the TCB they introduce a major impact on performance and substantially complicate the service deployment process.

1.2 The Impact of Speculative Execution Attacks on TEEs

Modern processors are vulnerable to devastating new speculative execution attacks, such as Meltdown [172] and Spectre [155]. In the chapter V, I analyze the vulnerability of Intel TEE, SGX, to speculative execution attacks, similar to Meltdown [172] and Spectre [155]. As stated by Intel, SGX does not protect against micro-architectural side channel attacks.

[1] Arguably, the most devastating attack on SGX is part of this thesis, described in chapter V.

Such side channel attacks exploit subtle timing variations resulting from contention on CPU micro-architectural resources to extract otherwise-unavailable secret information [221, 42, 217, 265, 292, 18, 84, 146, 85, 222, 104, 296, 291]. Recent works, Meltdown [172] and Spectre [155], combine micro-architectural attacks with speculative execution, allowing the attacker to read the entire address space of victim processes (if they contain vulnerable code) or of the operating system [124].

Much less is known, however, about the susceptibility of "perfect" SGX enclaves—that do not contain existing side channel vulnerabilities or other coding bugs— to side channel attacks. Thus, in my research, I ask: *Can an adversary extract secret data from an SGX enclave's address space even when the code running in that enclave does not itself have any security vulnerabilities?*

Next, I observe that SGX's integrity guarantees in the presence of side channels have received almost no attention from researchers. Thus, I ask, *what are the implications of side channel attacks on the SGX integrity guarantees? Can an adversary make an enclave operate on corrupted input data or corrupted state?*

Finally, given the importance of SGX remote attestation in establishing trust in the SGX ecosystem, I ask: *Can a side channel adversary erode the trust in SGX remote attestation? If so, what will it take to mount such an attack?*

I answer all three questions in the affirmative. I answer the first question by presenting several new attacks that compromise SGX's condentiality guarantees. I then use my attacks on SGX's condentiality properties to break SGX's integrity guarantees, thereby answering the second question. Finally, I use these attacks to recover the machines private attestation keys, thereby breaking SGX's attestation protocol and answering the third question.

1.3 Rethinking Out-of-Order Execution

In the last chapter of this thesis I discucss *NDA*: a new architecture design which defeats all known variants of speculative execution attacks. Speculative execution attacks [173, 154, 269, 282, 242, 256, 153, 124, 120, 176, 156] exploit micro-architectural behavior and side channels to exfiltrate sensitive information from a system. Unlike classical software exploits that modify and observe only architectural state (such as registers and memory), speculative execution attacks have demonstrated that attackers can retrieve secrets by controlling and observing micro-architectural state (e.g., the cache) during speculative wrong-path execution.

Existing software and hardware defenses are insufficient. First, Software mitigations are not applicable to existing binaries. Second, while hardware defenses have the potential to obviate the need to modify existing software, the currently deployed mechanisms are blocking only attack-specific mechanisms—which are not fundamental to speculative execution attacks. For example, STIBP [141] prevents the attacker from steering execution by polluting the BTB, but can be circumvented by other attack techniques [176, 156]. Finally, other proposed hardware mitigations [288, 150] block only a specific covert channel and recent work already showed other covert channels exist [242].

In this last chapter, I introduce *NDA* and show how it fundamentally defeats speculative execution attacks. I will analyze the performance impact of this new design when mitigating different classes of speculative execution attacks.

CHAPTER II

Hot Calls: A Fast Interface for SGX Secure Enclaves

2.1 Introduction

Cloud computing allows lowering the cost of computation and storage, outsourcing the acquisition and maintenance to a third party. Using hardware and software under the control of a third party implies substantial trust: trust that the service provider will not snoop on the data on its servers and will not tamper with the execution flow. Even if the cloud provider can be trusted not to actively snoop or tamper with processed data, users must also trust in the operating system, the virtual machine manager, and the firmware (BIOS & System Management Mode code - SMM). A compromise in the security of any of these layers, by means of remote attack or rogue employee tampering with the hardware, leads to compromising the information and the execution on the cloud.

In 2015, Intel released the Skylake micro-architecture, the first x86 production processor featuring a secure execution technology - Software Guard Extensions (SGX) [183, 113, 29]. This technology allows secure execution in user-space (ring 3) in a container called a secure-enclave, which is shielded from the OS, VMM, and SMM. Ideally, no vulnerability or intentionally malicious code in any of these layers should compromise the confidentiality or the integrity of the secure-enclave. No probing of physical buses outside the processor chip should compromise the security, as the memory is encrypted as well.

Recent work [240, 40, 34, 115, 297] has proposed new frameworks for performing large-scale computations and for porting existing applications to secure-enclaves. Although they provide qualitative discussion about the performance implications of running within an SGX enclave, it remains unclear what specific operations may slow down execution, and by how much. Such quantitative understanding is the cornerstone for constructing effective optimization strategies when developing secure-enclaves.

In this chapter I present a quantitative performance evaluation of SGX, quantifying the overhead of transferring control to and from secure-enclaves and encrypted memory I/O. First, I introduce a taxonomy of the operations involved in using the SGX framework and their costs in machine clock cycles. Based on this analysis, I propose a performance-boosting alternative interface to interact with secure-enclaves. This mechanism is adopted by Intel and integrated into SGX's SDK as *switchless calls* [263]. My research found that the overhead of calling a secure-enclave function is between 8,600 and 17,000 cycles (depending on cache state), compared to 150 cycles for a regular OS syscall [252], and compared to 1,300 cycles for a hyper-call in a KVM virtualization solution [72]. I also found that the mechanism allowing secure code to interact with the application or OS outside the enclave incurs between 8,200 and 17,000 overhead cycles (depending on cache state). This is a 54x-113x degradation in performance compared to regular OS calls.

I devise and evaluate several microbenchmarks to estimate the cost of transferring buffers to and from enclaves. While [107] suggests that the Memory Encryption Engine (MEE) adds no more than 12% overhead to the benchmark execution, I found that for the microbenchmarks encryption/decryption may add up to 102% increase in memory access time. On the *mcf* and *libquantum* benchmarks from SPEC 2006 [111], the slowdown was 55% and 420%, respectively.

The overhead of SGX-related calls becomes a significant bottleneck in applications with high system-call frequency. For instance, a database application serving 200,000 requests per second (*e.g., memcached*, as evaluated in Section 2.6.2) requires at least 200,000 system calls to transfer responses through the network. According to my measurements, each call consumes at least 8,200 cycles, totaling 1,640 million cycles. On a 4 GHz core, this amounts to 41% of the core time spent on merely facilitating the calls, without doing any actual work. My evaluation of non-trivial applications in Section 2.6, shows that this is not just a hypothetical problem.

Identifying that context switches used for facilitating system calls are a major bottleneck in SGX applications, I design and implement *HotCalls* - an alternative interface for calling enclave functions and requesting system calls by the enclave. *HotCalls* are based on a spin-lock synchronization mechanism, and provide more than an order of magnitude speedup. Compared to the standard SGX SDK [128] framework, *HotCalls* cost only 620 cycles per system-call in most cases, a 13-27x improvement.

I evaluated the performance of three non-trivial applications within SGX: *openVPN* (encrypted tunnel), *memcached* (memory based database), and *lighttpd* (fast HTTP server), using a straightforward approach to port them into SGX secure-enclaves. I show that, using *HotCalls*, it is possible to improve throughput by 2.6-3.7x and reduce the applications' response latency by 62-74%.

To summarize, in this chapter I present the following contributions:

- I identify and analyze fundamental operations in SGX technology that have major performance implications. I provide the first comprehensive evaluation of the latency of each such operation, by designing and running a set of microbenchmarks. Based on the microbenchmarks' results, I offer best practices for using SGX when performance is just as important as security.

- Leveraging the insights from the microbenchmarks, I design and implement a new interface to SGX, *HotCalls*, for communication between secure-enclaves and untrusted code. *HotCalls* are 13-27x faster than the existing mechanism provided by the SGX SDK.

- I evaluate the benefit of *HotCalls* on widely used applications: *openVPN*, *memcached*, and *lighttpd*, showing that the throughput of these applications can be improved by a factor of 2.6-3.7x and the response latency can be reduced by 62-74%.

2.2 SGX - Background

With the *6th Generation Intel Core* processors, *Skylake*, Intel introduced the Software Guard Extensions (SGX) instruction set that enables the use of a secure execution environment [183, 113, 29]. Similar to ARM TrustZone [19] *secure world*, SGX allows creating a secure execution context, called a *secure-enclave*, protected from the operating system and other user applications. Unlike the *secure world* in ARM TrustZone, the secure context created by SGX has only user-level privileges, and each user application may create several distinct secure-enclaves. A secure-enclave is a reverse sandbox - it protects the user-level software from being compromised by the environment: the operating system, the virtual machine manager, the BIOS (via SMM), and the hardware surrounding the CPU chip. Any of these may be malicious (like adversary OS Iago attacks [58], or hardware cold boot attacks [110]) or compromised (an OS, VMM, or SMM vulnerability [75, 283]). SGX allows clients to securely run software on untrusted servers maintained by a third party such as Amazon cloud computing, Microsoft Azure, or other cloud computing providers.

Guaranteeing confidentiality and integrity of execution on a remote third party server is not trivial. The cloud provider has inherent access to the hardware (memory buses, BIOS image) and the virtual machine management software (VMM), allowing the provider to eavesdrop on the memory contents and the execution flow of all software running on its servers. A security breach may also be introduced by a rogue employee in the cloud company, modifying the BIOS image or patching the VMM software. SGX allows running software operating on secret data in the cloud, without compromising its security. Ideally, it should not be possible for the cloud provider to affect the execution flow of the software, or inspect the secret data being processed, beyond the impact of a potential denial of service attack. SGX secure-enclaves may be used to implement secure databases, software using a secret key to encrypt/decrypt data, or other services processing sensitive data while running on hardware or software under the control of a third party.

The technical details of SGX instructions are detailed in the Software Developer Manual [137]. At boot time, the BIOS defines an area in memory called the Enclave Page Cache (EPC). This is part of the Processor Reserved Memory (PRM) area, which cannot be accessed by any software, regardless of its privileges. The EPC is encrypted by the Memory Encryption Engine (MEE) [107] residing on the processor die. Every processor has two master secrets saved as fused keys, set uniquely at manufacturing time for each individual processor. The first master secret is used to derive memory encryption keys and it is not kept in Intel's records. The second master secret is used to derive a public-private authentication pair, used for attestation, and it is stored in Intel's database. The MEE protects against hardware attacks, trying to snoop on the data when it is in transit to and from memory. The MEE also provides integrity protection, preventing rollback attacks, and protects against malicious modification of the linear-to-physical mapping by a malicious OS or VMM.

The secure context is created by initializing a secure-enclave using the ECREATE [1] instruction. Memory pages containing code and data are copied into the enclave's encrypted memory by invoking the EADD and EEXTEND instructions. This code is referred to as the *trusted code*. The pages added to the enclave are hashed to generate an enclave *measurement*. After all the trusted code and data are transferred into the enclave, the *measurement* is finalized by invoking EINIT. During the attestation process [138], the CPU uses the relevant master secret to sign the measurement and generate a report. The report is passed to a remote client (running on a trusted machine), which then contacts Intel's servers to verify that the signature was produced by a genuine Intel processor. The remote client can then provision secret data to be processed by the enclave via a secure channel that is created as part of the attestation process. The interested reader may refer to [69] for additional information.

[1]SGX supports only two instructions: ENCLU and ENCLS. All operations such as ECREATE, EADD, EINIT, *etc.* are considered leaf functions of ENCLU or ENCLS, but are referred to as simply *instructions* for clarity.

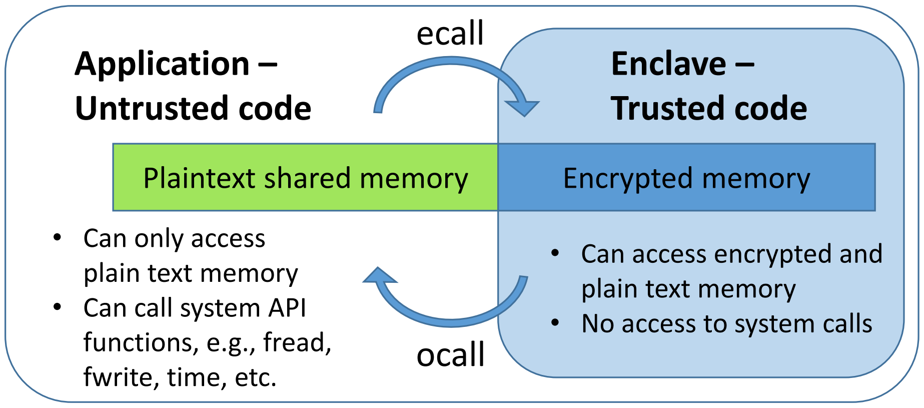

ecall
Application – Untrusted code
Enclave – Trusted code
Plaintext shared memory
Encrypted memory
• Can only access plain text memory
• Can call system API functions, e.g., fread, fwrite, time, etc.
• Can access encrypted and plain text memory
• No access to system calls
ocall

Figure 2.1: Interaction between the application and the secure-enclave: control is transferred to the enclave via *ecall*; requests for OS API calls are processed via *ocalls*.

2.2.1 Application-Enclave Interaction

Entering the enclave: Figure 2.1 illustrates the communication mechanisms between the application and the secure-enclave. After the secure-enclave is initialized, the only way for the *untrusted code* (outside the enclave) to start executing the *trusted code* (inside the enclave) is by invoking the EENTER instruction. EENTER performs the context switch into the enclave, saving the state of the *untrusted code* and restoring the last known state of the *trusted code*. This context switch is conceptually similar to VMENTER and VMEXIT used for virtual machine context switches in Intel's VTX technology [196]. To ease development of secure-enclaves, Intel provides wrapper code, called *ecall* (for entry call), to perform the preparation of the environment and invoke the EENTER instruction [131, 132].

Accessing external resources: Because the enclave is *trusted code* running with user-level privileges, *i.e.*, ring 3, it has no access to hardware or other OS resources. In order to gain access to external resources, such as the file system, network, or clock, the enclave must exit to the *untrusted code*. It can do so via the EEXIT instruction. EEXIT performs the reverse context switch and switches back to the *untrusted code*. The wrapper code to do so is called *ocall* (for out call).

Declaring edge calls: *ecalls and ocalls* are considered edge functions, as they cause execution to cross security boundaries. The functions' parameters need to be marshalled

#	Micro-benchmark	Description	Median Latency (cycles)
1	Ecall (warm cache)	Calling a secure enclave function with no parameters, and immediately returning. See Fig. 2a (solid line).	8,640
2	Ecall (cold cache)	Same as above, the entire cache is flushed between consecutive experiments. See Fig. 2a (dotted line).	14,170
3	Ecall buffer transfer	Calling a secure enclave function, passing 2KB buffer to / from / to&from the enclave. Other buffer sizes are depicted in Fig. 4.	9,861/11,172/10,827
4	Ocall (warm cache)	Exiting the secure enclave to execute an untrusted call. See Fig 2b (solid line)	8,314
5	Ocall (cold cache)	Same as above, the entire cache is flushed between consecutive experiments. See Fig. 2b (dotted line).	14,160
6	Ocall buffer transfer	Calling untrusted code, passing a 2KB buffer to / from / to&from the untrusted code. Other buffer sizes are depicted in Fig. 5.	9,252 / 11,418 / 9,801
7	Reading memory	Consecutively reading from a 2 KB buffer in encrypted/plaintext memory in chunks of 64 bits. Other buffer sizes are depicted in Fig. 6.	1,124 / 727
8	Writing memory	Consecutively writing to a 2 KB buffer in encrypted/plaintext memory, in chunks of 64 bits. Other buffer sizes are depicted in Fig. 7.	6,875 / 6,458
9	Cache load miss	Reading 8 bytes (64 bits) from encrypted/plaintext memory	400 / 308
10	Cache store miss	Writing 8 bytes (64 bits) to encrypted/plaintext	575 / 481

Table 2.1: Microbenchmarks targetting fundamental operations using SGX secure enclaves. Every microbenchmark consists of 10 batches of 20,000 experiments, totalling 200,000 measurements. For microbenchmarks involving memory operations, the relevant memory addresses are evicted from the last-level cache prior to every single measurement.

and copied from encrypted memory to plaintext memory, and vice versa. For the boundary-crossing to be secure, several security checks need to be performed on the call's parameters, particularly in the case that they are pointers. To ease the development of SGX enclaves, Intel provides an edge function creator tool called *edger8r* (pronounced edgerator), to automatically generate secure wrapper code of *ecalls* and *ocalls*. If some of the parameters passed to the edge function are buffers, the specific wrapper code generated depends on whether the buffers are used as input parameters, output, or both.

To automatically generate the ecalls and ocalls code, the programmer must use an SGX-specific syntax to declare the edge functions in an EDL extension file. The declaration includes the parameters each function receives, and their attribute: input, output or both. The *edger8r* then parses the EDL file and generates wrapper glue code for ecalls and ocalls. The glue code consists of two parts: *trusted* and *untrusted*. My proposed *HotCalls* framework makes use of this generated code to facilitate the calls.

2.3 Overhead of Fundamental Operations

Inspecting the SGX technical documentation raises several questions with respect to performance:

What is the overhead of a secure context switch into the enclave, and out of it? Every access to non-user-space resources, such as files, network, clock, *etc*. requires a context switch to the untrusted code, to perform an OS API call. If the enclave code performs such requests at high frequency, the time spent on the context switches will have a drastic impact on the program's performance. Table 2.1 describes the microbenchmarks we evaluated. Microbenchmarks 1, 2, 4, and 5 specifically measure context switch latencies.

What is the cost of passing parameters and data between the application and the secure-enclave? In order to service these requests, parameters and buffers must be transferred from the secure-enclave to untrusted execution, and vice-versa. Microbenchmarks 3 and 6 in Table 2.1 measure the cost of transferring data in each direction.

What is the cost of accessing encrypted memory? The enclave memory resides in the EPC and it is encrypted. The Memory Encryption Engine (MEE) provides both confidentiality, integrity, and protection from rollback attacks. Providing these security guarantees is expected to come at a performance cost. Microbenchmarks 7 and 8 measure the access time for consecutive memory buffers of various sizes, compared to access times for regular (not encrypted) memory. Microbenchmarks 9 and 10 measure the access time of non-consecutive reads and writes, and estimate the cache miss latency of encrypted memory, compared to that of regular memory.

2.3.1 Experimental Setup

All experiments executed on a Supermicro server X11SSZ-QF, 64 GB DDR4 RAM @ 2133 MHz, Intel Core I7-6700k 4GHz with 4 hyper-threaded cores (total of 8 logical cores). Dynamic frequency and voltage scaling were disabled; the operating system was the Ubuntu server 14.04 LTS. The SGX SDK version we used is 1.5.80.

Measuring methodology: The execution time in clock cycles is estimated using the *read time stamp counter* instruction - RDTSCP. RDTSCP is a serialized variant of RDTSC, obviating the need to combine the costly CPUID instruction with RDTSC. On production-deployed SGX systems, RDTSC and its variants are not allowed within the enclave, hence all RDTSCP calls must be executed in the untrusted code. Calibration of RDTSCP shows it is accurate up to +/- 2 cycles. Each microbenchmark consists of 10 groups of 20,000 runs, totaling 200,000 test executions.

When using RDTSCP to measure cycle count of a user-space operation, it is important to ensure that there are no context switches to the operating system, which would contaminate the measurement. To avoid this contamination, we ran each experiment many times. Since interrupts and context-switches to the OS are infrequent, repeating the experiment multiple times ensure that the majority of the measurements are not interrupted. Moreover, any context-switch to the OS while the application is executing inside the enclave, causes an *Asynchronous Exit* - AEX, which forces the execution to jump to a known location in the *untrusted code*. We monitored this location in order to count the number of AEX events. Out of 200,000 measurements per micro benchmark, around 200-300 experienced an *Asynchronous Exit*. Hence, we discarded those runs for the sake of performance estimation.

2.3.2 Measuring Ecalls Overhead

When the untrusted code wishes to initiate a trusted function, it does so via an *ecall*. The transition into and out of the trusted code is implemented partially in software, *i.e.,* the SDK, and partially in hardware via the EENTER and EEXIT instructions.

Software interface: The ecall is a wrapper to the EENTER instruction. The ecall first locates the enclave with the specified ID, then acquires a read/write lock, finds an available Thread Control Structure (TCS), saves Advanced Vector Extensions (AVX) [79] state, checks for floating point exceptions, and finally calls the EENTER instruction.

Hardware interface: The EENTER instruction preforms a secure context-switch as described in Intel's Software Developer Manual (SDM) [137]. Most microcode operations in EENTER involve disabling debugging/tracing mechanisms and defensive checks of enclave management structures: SGX Enclave Control Structure (SECS) and Thread Control Structure (TCS). After validating the SECS and TCS structures, the registers representing the untrusted context (*e.g.,* RAX, RSP *etc.*) are backed up and the enclave context is loaded instead. At the completion of the trusted function execution, EEXIT performs the reverse context switch, and un-suppresses the debugging/tracing mechanisms. As these operations potentially involve sparse encrypted-memory accesses, they may add significant latency (see Section 2.3.4, microbenchmarks 9,10 in Table 2.1, and Figure 2.8).

To measure the latency of performing an *ecall*, we created an empty *ecall*, *i.e.,* a trusted function that receives no parameters and returns no parameters. Since running RDTSCP inside the enclave generates a fault, we can only measure the execution time of entering and exiting the enclave together. The solid line in Figure 2.2a depicts the cumulative distribution function (CDF) over 200,000 measurements. Over 99.9% of the measurements are between 8,600 and 8,700 cycles. For comparison, [252] estimates a transfer to the OS and back in 150 cycles, and [72] estimates hyper-calls to the hypervisor as taking 1,300 cycles (KVM hypervisor on x86 processor). These measurements reflect performance with a warm cache. Because of the repetitive 1 of the tests, the memory structures that are accessed to execute the ecall are in cache for most runs. To eliminate this artifact, we conducted the same experiment, but flushed the entire last level cache (LLC) before each run. The dotted line in Figure 2.2a depicts the CDF of this experiment: the round trip time of executing an ecall is between 12,500-17,000 cycles, with a median of 14,170 cycles, that is, 83-113x slower than an OS system call.

2.3.2.1 Transferring Memory To/From the Enclave

When transferring parameters to a secure function, the SDK framework generates code to serialize all the parameters inside a single contiguous data structure. This data structure

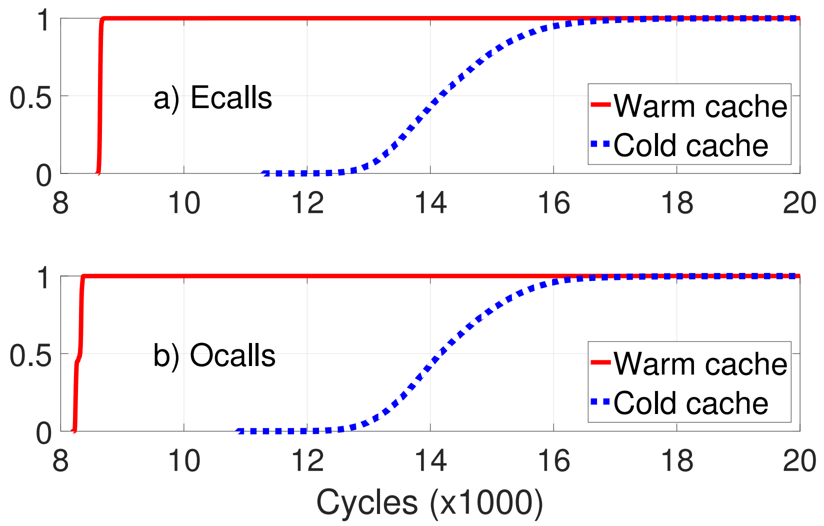


Figure 2.2: CDFs of ecalls and ocalls performance. In cold cache experiments the entire 8 MB LLC cache was flushed prior to every experiment, causing relevant data structures and code needed for the ecalls/ocalls to be fetched from memory. a) ecalls: with warm cache, 99.9% of the calls complete between 8,600 and 8,680 cycles. With cold cache 99.9% of the calls take between 12,500 and 17,000 cycles. b) ocalls: without flushing the cache 99.9% of the calls complete in 8,200 - 8,400 cycles. With cold cache 99.9% of the ocalls take between 12,500 and 17,000 cycles.

is in the insecure memory and a pointer to the data structure is transferred to the trusted function. To avoid data leakage from secure memory, the trusted function wrapper verifies that the entire data structure pointed by the pointer is outside the enclave memory.

When dealing with buffers, the programmer can choose among four options: *user_check*, *in*, *out*, and *in&out*. The programmer selects the option when declaring the ecall in the EDL file. The EDL file is written by the programmer with a specific Intel-provided syntax, to declare edge functions (ecalls and ocalls), the parameters they receive, and additional permissions for each edge function. Figure 2.4 reports the round trip time in cycles of ecall including transferring buffers to&from the enclave. The cycle count for transferring 2 KB buffers are shown in Table 2.1, microbenchmark 3.

Zero copy: The *user_check* option means that the SGX framework treat the pointer provided as if it were a value parameter. No security checks are carried out to validate if it points to encrypted or regular memory and no copy is done. This is useful if this pointer is pointing to encrypted memory (received earlier from the enclave), or in case of transfer of a pointer, which will be used later by the untrusted code. An example can be a FILE pointer that the enclave will later use in a *fread* call.

Copying in: The *in* option tells the *edger8r* tool to generate wrapper code that will allocate memory in the secure memory, according to a size parameter supplied by the untrusted code, and then copy the buffer into the enclave. Memory encryption is transparent to software: memory writes to secure memory are first written un-encrypted in cache, and are encrypted by the MEE when the cache line is evicted to RAM. The pointer that will be given to the trusted function implementation will point to a location within the enclave encrypted memory. This is useful especially in cases where a threat of Time-Of-Check-Time-Of-Use attacks (TOCTOU) exists. For example, if the secure-enclave checks a cryptographic signature of a given data, and then uses the (supposedly verified) data for a critical operation, while between the time of check and the time of use the *untrusted code* might have changed the data. In order to measure the accurate latency of transferring new data into the enclave, we removed the buffers inside and out of the enclave from the cache by calling *clflush* on the relevant addresses, before each measurement.

Copying out: The *out* option is used when the untrusted code passes a buffer as an output argument, *i.e.*, the trusted code fills the buffer with data. Using the *out* option generates wrapper code that allocates a buffer in secure memory according to a size parameter supplied by the untrusted code, zeroes the entire buffer, and passes it as the pointer for the trusted function. Upon return, this buffer is copied back to the insecure memory. The security reasoning behind zeroing the buffer is to avoid information leakage. Since the buffer is allocated on the secure memory heap, it may initially contain secret data. Should the trusted function fill only part of the out buffer, this secret data will be copied back to the insecure memory, leaking secret data similar to the HeartBleed bug [74]. To prevent data leakage, the SDK zeroes the buffer using a proprietary version of *memset*, which operates byte-wise. This is extremely inefficient on a 64 bit platform, and explains the added latency when using the *out* option.

Copying in&out: The *in&out* option is used when the untrusted code passes a buffer as

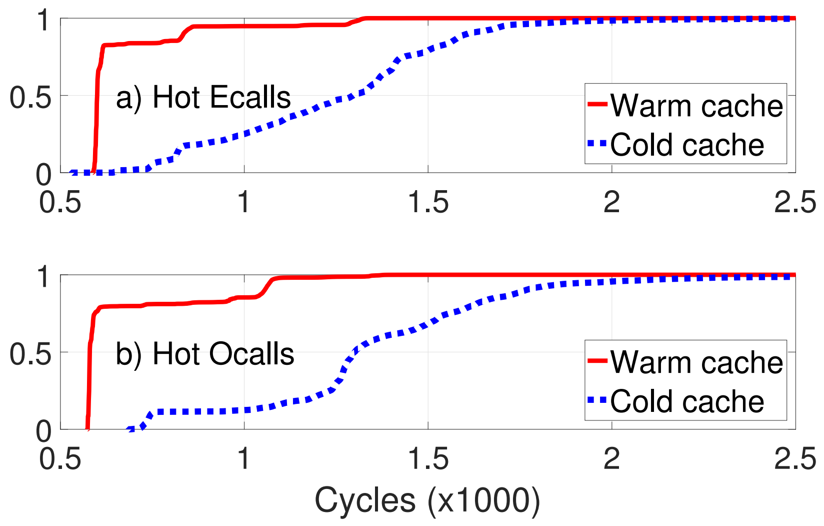


Figure 2.3: CDF of HotEcalls and HotOcalls. Over 78% of the calls are executed in less than 620 cycles, and 99.97% are completed within 1,400 cycles. For comparison, the native SGX SDK calling mechanism is 13x-27x times slower. HotCalls' footprint in memory is extremely small, compared to native calls, reducing the chances of cache misses during the HotCall execution.

input *and* output argument. In this case the generated wrapper code allocates a buffer in secure memory, copies the data from the insecure memory, and passes the new allocated buffer to the trusted function. Upon return, the buffer is copied back to the insecure buffer, obviating the need to zero the allocated buffer, like in the case of the *out* option. Copying in&out is faster than using just the *out* option, as the *memcpy* used by the SDK is more efficient than the byte-wise *memset* used in the *out* option.

2.3.3 Measuring Ocalls Overhead

When the secure code requires access to external resources, such as network, files, clock *etc.,* it needs to invoke out-calls.

Software interface: Similar to ecalls, ocalls are declared in the EDL file. The *edger8r* tool provided in the SGX SDK generates trusted and untrusted glue code. The trusted code marshals data structures, performs security checks on pointers values, and then executes the EEXIT instruction. The untrusted code organizes the input arguments, calls the requested OS API call (*fopen, send etc.*), marshals data to be returned to the enclave (the ocall output arguments) and executes the ERESUME instruction to resume execution of the trusted code.

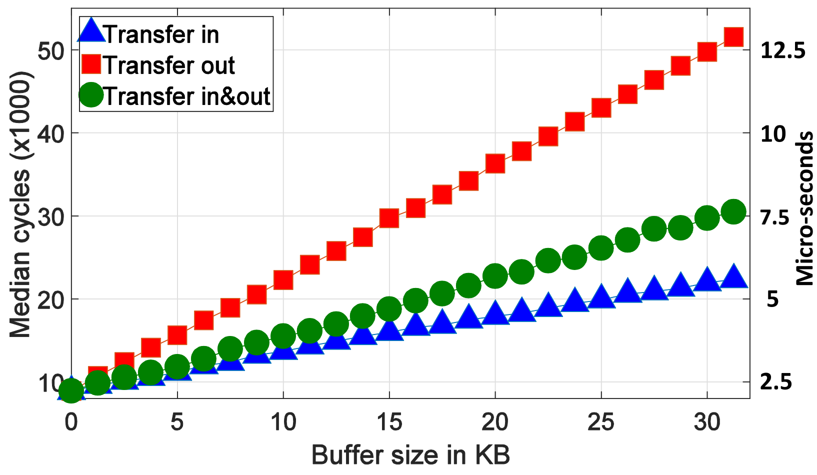


Figure 2.4: Latency of ecall + transferring a buffer in/out/in&out. Transferring a buffer out is extremely taxing due to the inefficient *memset* implementation in SGX SDK.

Hardware interface: The operations performed by EEXIT are as described in Section 2.3.2. The operations performed by ERESUME instruction are described in Intel's SDM [137]. ERESUME performs similar operations as EENTER, but resumes execution of the trusted code from the instruction after EEXIT.

As in ecalls, pointer parameters can be marked in the EDL file as *user_check, in, out, in&out* to instruct the *edger8r* how to generate the code. Figure 2.2b shows the round trip latency for performing an ocall. Figure 2.5 shows the performance implication of using the in/out/in&out options when transferring buffers to/from ocalls.

Zero-copy: As before, *user_check* entails zero-copy and no checks. This is useful when passing pointers that are provided by the OS, such as a file pointer.

Copying out: contrary to ecalls, in the case of ocalls, the *in* option means "into the ocall", *i.e.,* from the secure memory *out* to the insecure memory. The wrapper code verifies that the pointer points to a location within the enclave. It then allocates memory on the insecure stack according to a size parameter supplied by the enclave (no use of *malloc* here). Finally, it copies the buffer to the insecure memory. Upon re-entry to the enclave the allocated memory is freed by unwinding the insecure stack.

Copying in: The *out* option stands for "out of the ocall, into the enclave". The wrapper code allocates memory on the insecure stack, according to a size parameter supplied by the enclave. The newly allocated buffer in the insecure buffer is then zeroed. After the ocall itself returns, the buffer is copied back into secure memory. In my opinion, zeroing the buffer in the insecure memory has no security benefit. The untrusted code can access this memory anyway, prior to invoking the latest ecall. As mentioned before, zeroing the buffer is carried out via *memset*, which the SDK implements as byte-wise zeroing. This is extremely inefficient, and explains why the *out* option is much slower than the *in&out* option. We observe that zeroing a buffer in the plaintext memory does not add a security benefit, and thus this operation can be removed. In Section 2.6, we evaluate the impact of the *No-Redundant-Zeroing* approach on common applications.

2.3.4 Measuring Memory Access Overhead

The enclave code can access both regular memory, and enclave memory. The enclave memory resides within the Enclave Page Cache (EPC) and is encrypted by the Memory Encryption Engine (MEE), which resides on the processor's die and is described in [107]. The MEE provides confidentiality, integrity, and anti-roll-back protections for the entire EPC. These guarantees are provided by maintaining an *integrity tree*, with its root stored on the processor's die. A full walk of the tree involves several memory accesses. Therefore, a "MEE cache" is used to prevent significant performance costs, when accessing nearby memory addresses.

The work presented in [107] also analyzes the potential performance degradation of using encrypted memory, and evaluates the worst case benchmark to exhibit a 12% overhead when using encrypted memory. However, my measurements shows substantially higher overhead, as will be detailed shortly.

Consecutive buffer access: Figures 2.6, 2.7 plots the memory read/write times when accessing encrypted and plaintext memory, for different buffer sizes. Entries 1 and 8 in Table 2.1 list the results of reading and writing 2 KB buffers. The memory read & write microbenchmark consists of accessing 8-bytes (64 bit) aligned words of consecutive

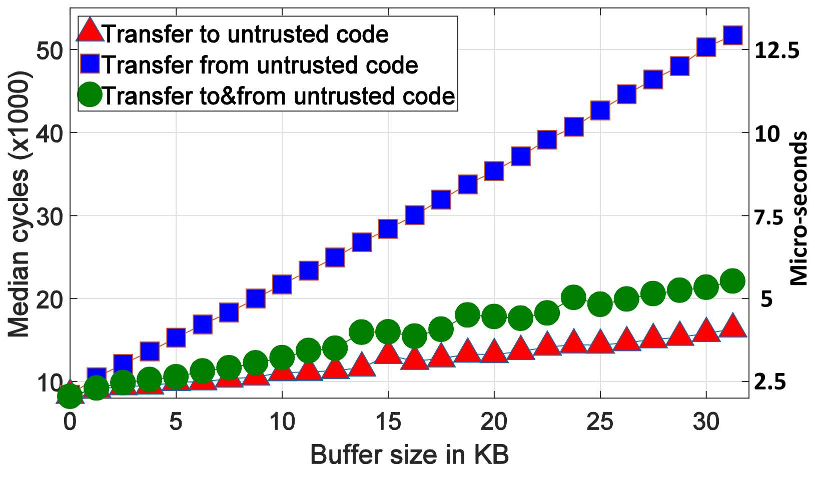


Figure 2.5: Latency of ocall + transferring a buffer to/from/to&from untrusted memory. Transferring from untrusted code has high latency, due to (redundant) zeroing of the buffer in the untrusted memory with the inefficient *memset*, provided by the SDK.

addresses, for different buffer sizes. The buffers were flushed out of the last level cache (LLC) before each experiment, and *mfence* was called before the final call to RDTSCP, to ensure the operations had completed. When measuring write latency, the experiment was completed by flushing the buffer from cache via *clflush* followed by *mfence* prior to calling the final RDTSCP.

Cache misses: To estimate cache-load-miss and cache-store-miss latency, we performed the same read and write experiments, accessing only the first 8 bytes (64 bit) of an address which is aligned to the cache line size (64 bytes on the tested machine). The cache line was evicted from the LLC before each measurement. The first four bars of Figure 2.8 shows the results on the micro benchmarks for reading and writing in encrypted memory, and for cache miss penalties. The cycle counts are detailed in lines 9,10 of Table 2.1. Cache-load misses and cache-store-misses are 30% and 19.5% slower when accessing encrypted memory vs. plaintext memory.

SPEC 2006 memory intensive benchmarks: To test the effect of more complicated memory access patterns, we selected three highly memory-intensive benchmarks from SPEC 2006 [111]: *mcf, libquantum,* and *astar*. Figure 2.8 compares the latency of the memory microbenchmarks mentioned and the selected benchmarks from SPEC 2006. Previous work published by Intel [107] measures several benchmarks from SPEC 2006, showing that, in the worse case, the encryption overhead is roughly 12%. In my measurements, *mcf* runs 55% slower in the SGX enclave, and *libquantum* runs 5.2x slower. A likely explanation for the extreme slowdown in the case of *libquantum*, is that it required 96 MB of memory, while the entire Enclave Page Cache (EPC) is 93 MB. This forces paging out encrypted memory pages, which requires further SGX operations.

2.3.5 Lessons Learned

We now discuss insights from the measurements, which can be used by developers designing secure-enclaves to devise optimization strategies. These insights were also instrumental in my design of *HotCalls*, as we discuss in Section 2.4.

Cost of ecalls & ocalls: Compared to regular OS syscalls, an ecall is 54x more cycles at best (8,200 vs 150) when the cache is warm, and 83-113x at worst when cold (12,500-17,000 vs 150). If an application has high call rate, for example, 100,000 calls per second, on a 4 GHz core the ocalls will consume 20-40% of the execution time. The applications evaluated in Section 2.6 exhibited more than 200,000 calls per second, as detailed in Table 2.2. My solution, *HotCalls*, proposes an alternative calling mechanism that reduces this latency to as low as 620 cycles per call, which is 13-27x faster than the default ecalls and ocalls mechanism.

Ocalls vs. Ecalls: Ocalls may execute slightly faster than ecalls. Transferring buffers from the enclave to the untrusted application is faster using ocalls: 9,252 cycles for ocalls *in* vs. 11,712 cycles for ecalls *out* (2 KB buffers). This insight may lead to an optimization strategy of using ocalls to receive data from the enclave, rather then delivering it via an output parameter with ecalls.

Cost of memory access: Write accesses of encrypted memory incur 6.5-19.5% overhead, and read accesses incur 30-102% overhead, depending on buffer size. For software that is memory read intensive, we can estimate the impact on throughput in the following

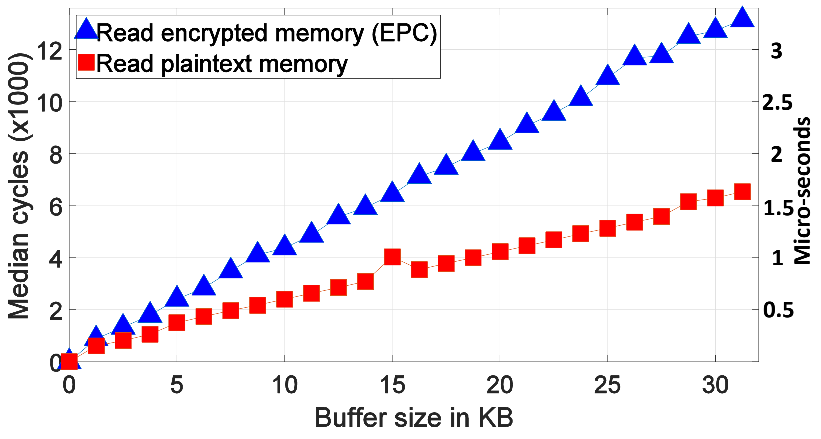


Figure 2.6: Latency of consecutive memory reads, for encrypted and plaintext memory. All buffers were evicted from the cache prior to every experiment. The overhead of reading encrypted memory of sizes 2,4,8,16,32 KB is 54.5%, 68%, 71%,94%,102%, respectively.

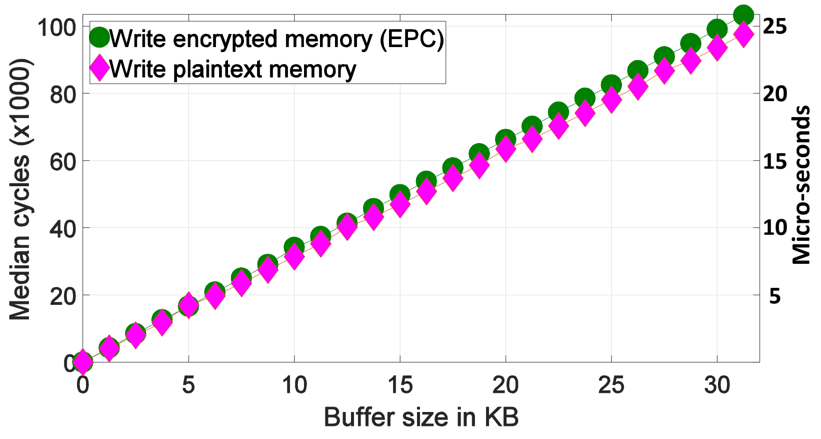


Figure 2.7: Latency of consecutive memory writes, for encrypted and plaintext memory. All buffers were evicted from the cache prior to every experiment. The overhead of writing encrypted memory is roughly around 6% for all buffer sizes above 1 KB.

way: without encryption, N memory reads are carried out in time T, and with encryption in $1.5T$ (assuming 2 KB buffers). Thus, we can expect that encryption slow down throughput to $\frac{N}{1.5T}/\frac{N}{1T} = 66\%$ throughput, not accounting for any other SGX performance impact.

Selecting the right transfer method: For both ecalls and ocalls, the *out* option is extremely inefficient. This is due to the inefficient implementation of *memset* in the SDK, used for zeroing buffers. Assuming that the untrusted code has nothing to hide from the enclave, it is more efficient to use the *in&out* option instead. Despite the fact that the buffer will be redundantly copied to the secure memory, this will save 885 / 1617 cycles for ecalls / ocalls in the case of a 2 KB buffer.

Opting for user_check: If the enclave just dumps data to the output buffer, then there is no threat of the untrusted code modifying the data while the enclave code is processing it. In that case it is preferable to use the *user_check/zero-copy* option, and have the enclave directly write data to un-encrypted memory. This will save about 3,000 cycles on a 2 KB buffer (11,712 vs 8,640 cycles). Developers should be careful when using this option. If the secure-enclave performs encryption or decryption in place, having access to the partially-processed buffer during the encryption/decryption process may lead to exposing the secret key.

Further optimizations: The latency incurred in buffer transferring is incurred by *memcpy* and *memset*. The SGX stdlib implementation of *memset* is operating on memory byte-wise, which is extremely inefficient on a 64 bit processor. Using a more optimized version of *memset* may significantly improve performance. Additionally, when large buffers need to be transferred, it may by beneficial to use optimized versions of *memcpy*, utilizing Advanced Vector Extensions (AVX) [79] instructions which are able to copy words larger than 64-bits efficiently. Intel may wish to include this optimization in future versions of the SGX SDK.

2.4 HotCalls: An Optimized SGX Interface

Motivated by the fact that using the default SDK calling mechanism may lead to a 113x slowdown, we now present *HotCalls*, an alternative mechanism to perform ecalls and ocalls, leading to an order of magnitude performance improvement. Compared to 8,200-17,000 cycles required for SDK ecalls/ocalls, *HotCalls* can be as fast as 620 cycles. While SGX calls rely on expensive secure context switches, *HotCalls* operations are based on using shared un-encrypted memory.

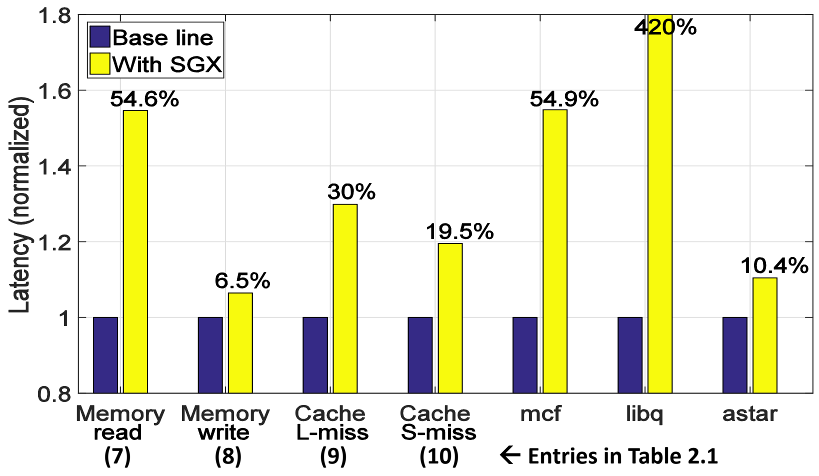


Figure 2.8: The overhead of memory encryption on memory-access speed: L and S stand for Load and Store. Memory reads and writes are of consecutive 2 KB buffers. *mcf*, *libquantum* (libq), and *astar* are memory intensive benchmarks from SPEC 2006.

2.4.1 HotCalls Architecture

Edge calls in SGX are context-switch operations, similar to VMENTER and VMEXIT used for virtual machine context switches in Intel's VTX technology [196]. Previous work exists on optimizing communication mechanisms between hardware and software (interrupt handlers), and virtual machine manager and guest operating systems (hyper-calls or VM-exits). An approach that has shown to be effective is avoiding the software context switch by using shared memory as a communication channel and dedicating a thread to poll for new messages. This has been tried in Linux NAPI to optimize access to hardware [238] and in virtualization scenarios to eliminate the need for an expensive context switch [174, 159]. We take a similar approach and propose an architecture that consists of a *requester* and a *responder*, communicating via un-encrypted shared memory.

Figure 2.9 illustrates this architecture, where the enclave code is the requester, and the untrusted code is the responder. The requester is the party requesting a call, while the responder is an *On Call* thread, standing by, waiting for a call. It does so by constantly polling a shared memory location. Synchronization of the shared memory is provided using a spin-lock. When the requester makes a call, it acquires the spin-lock and checks a shared Boolean variable to verify that the responder is not currently busy. If the responder is available, the requester copies relevant data to unencrypted shared memory, and points to that data via the **data* pointer. The code to encapsulate parameters within the *data* structure is the same code used by the SDK ecalls/ocalls mechanism, that is automatically generated by the *edger8r* tool. To support more than one specific call, *e.g.,* read, write *etc.,* the requester specifies the ID of the call it is requesting. This is an entry ID to a function call table, known to the responder. The call table approach is similar to the SDK implementation of ecalls and ocalls. Once the data pointer and the requested call ID are in place, the requester signals "go" to the responder, by marking the responder as busy, and releases the lock. The responder is constantly monitoring the same shared memory and executes the relevant call when requested.

2.4.2 Practical Considerations

Spin-lock: Use of standard POSIX MUTEX is not possible, as it requires calling upon the operating system services, defeating the entire purpose of HotCalls. Synchronization techniques using MONITOR/MWAIT instruction entails several thousands of cycles, similar to regular SGX calls [28]. The SGX SDK provides a spin-lock implementation as *sgx_spin_lock*. This is a straightforward busy-wait implementation and does not relate to SGX, so it can be used by both the enclave and the untrusted code.

Minimizing self-contention: To ensure that both the requester and responder get a chance to acquire the spin-lock, *PAUSE* instructions are added after releasing the lock. This gives a chance to other threads to try and acquire the spin-lock. The *PAUSE* instruction was designed by Intel specifically to improve performance in spin-lock busy wait loops, by minimizing memory order violations of speculative loads, and also to help reduce power consumption.

Maximizing utilization: As the responder is constantly monitoring the shared memory, it is effectively using 100% of the logical core. The utilization can be considered as the amount of time the responder is spending on *ExecuteCall* vs. the time spent on acquiring the lock and checking the Boolean flag value. This utilization can potentially be improved by sharing the responder thread with several requesters.

Preventing starvation: Contention of several requesters on the responder can cause the requester to loop many times before it acquires the lock and the responder is available. The maximum worst-case wait time is therefore potentially unbounded. As a mitigation for this potential starvation, the requester can set a timeout - a maximum number of times to check if the responder is available. If the timeout expires, the requester can fall back to using regular SDK calls. In my experiments and evaluation of applications, we set this timeout to 10, and it never expired. Nevertheless, we find this mechanism vital for producing reliable code.

Conserving resources at idle times: When there are not many calls, the responder wastes CPU resources, constantly polling shared memory. To conserve resources during idle times, a timeout counter can be set. The counter is decremented when there is no request waiting (step 2 in Fig. 2.9, right side), and reset when a request arrives (in step 3). When the timeout counter expires, the responder set a *sleep* flag, and goes to wait on a conditional variable (POSIX pthread_cond, or *sgx_thread_cond*). The requester notices the *sleep* flag is set and signals the condition variable before issuing the request.

Marshalling parameters to be transferred to the function: The SDK generates code for packing ecall/ocall parameters. The framework we built for the evaluation automatically uses this code to pack/unpack the parameters and copy buffers if needed. This solution allows us to also avoid redundant buffer zeroing when transferring data from the untrusted code, without compromising security. The performance implications of removing the redundant *memset* will be detailed in the next section.

2.4.3 Empirical Evaluation of HotCalls

The implementation of HotCalls consists of 115 lines of code. Similar to the microbenchmarks described in Section 2.3, we performed 10 batches of 20,000 measurements each, totalling 200,000 measurements. Figure 2.3 shows the CDF of the latency of HotEcalls and

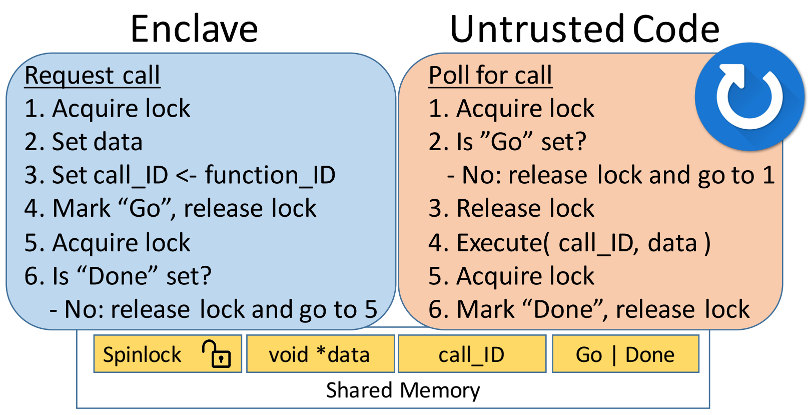


Figure 2.9: HotCall architecture. The secure-enclave requests a call by signaling a request via a shared variable in un-encrypted memory, together with the ID of the requested function. The responder thread in the untrusted side continuously polls the shared memory to check if a call request has been made.

HotOcalls. In both cases, more than 78% of the calls took less than 620 cycles (warm cache). Over 99.97% of the calls took fewer than 1,400 cycles. For comparison, the ecalls/ocalls mechanism provided by the SGX SDK requires 8,200-17,000 cycles.

2.4.4 Implications of Using an Additional Core

The benefit of using an additional logical thread to utilize HotCalls can be analyzed from two perspectives: application's throughput and overall power consumption. Analysis of both perspectives depends on whether HotCalls increases the application's throughput by more than a factor of 2, as explained below.

Throughput: When considering optimizing an application with HotCalls, the overall throughput increase by using HotCalls should be compared to the potential benefit of simply adding an additional worker thread. This is not always possible, as some applications are developed in a single thread. The additional extra thread cannot increase the overall throughput by more than a factor of 2. Hence, HotCalls are preferred over adding an additional worker thread, when it more than double the throughput.

Power consumption: When the responder is idle, or underutilized, it issues the *PAUSE* instruction in a loop, therefore it is not expected to consume much power. If the responder is idle for relatively long periods of time it can conserve power by waiting on a conditional variable, releasing the core resources, as suggested in Section 2.4.2. When the responder is busy, and the overall throughput increases by more than a factor of 2 (as for all the applications evaluated in Section 2.6) even if the power consumption doubles, the power per given throughput unit is still more efficient when using HotCalls.

2.5 Security Analysis

In order to assess to security implications of using HotCalls we examine the modifications which may affect the trusted code running inside the enclave.

Using shared plaintext memory for communication: The HotCalls technique for passing data structures between the enclave and the untrusted code is no less secure than the SGX SDK's mechanism. HotCalls source code for marshalling data structures between the enclave and the untrusted code is *the same code* used by the SDK's ecalls and ocalls implementation, generated by the *edger8r* tool. Any manipulation possible on data, which was marshalled by HotCalls, is also possible when using the SDK's ecalls and ocalls.

Attacks on the *data* pointer: Any security breach possible via manipulating the *data* pointer used by HotCalls is also possible on the pointer passed by the SDK's implementation of ecalls and ocalls. For ecalls, the responder (inside the enclave) identifies the request for a call, and passes the *data* pointer to the original function created by the *edger8r* tool. From then on the source code is identical to the default SDK implementation, including all security checks on the pointer in untrusted memory, and the copying of relevant buffers by using *in* or *out* modifiers (see Section 2.3.2.1).

In the case of ocalls, the HotOcall wrapper in the trusted code is almost identical to the original ocall code generated by the *edger8r* tool. The only difference is replacing the call to the SDK function *sgx_ocall*, responsible for invoking the EEXIT instruction, with code requesting a *HotCall*, similar to the "Request call" function illustration in Figure 2.9.

Requesting a function via call_ID: The technique of setting a function number in shared memory is also utilized by the SDK. The “call_ID” in HotCalls is comparable to the “ocall_index” variable used by the SDK. Any manipulation on “call_ID” is also possible on the “ocall_index” passed by the SDK to the untrusted ocall. Such manipulation to the “call_ID” or “ocall_index” will cause the untrusted code to execute a wrong function, hence no new vulnerability is introduced.

Using the spin-lock located in shared memory: Tampering with the synchronization provided by the spin-lock will either cause a denial of service (DoS) due to a deadlock, which is out of the SGX threat model or will cause multiple threads to access the same data in plaintext memory at the same time. Similar to HotCalls, the SDK’s ecalls and ocalls involve pointers in untrusted memory, which are accessible to the adversary in the SGX threat model. Therefore, the adversary can manipulate these pointers to cause multiple threads to access the same memory simultaneously, whether the threads are executing trusted or untrusted code.

2.6 Evaluating HotCalls on Applications

To evaluate the performance impact of SGX on complex software we chose three applications: memcached, openVPN, and lighttpd. Figures 2.10 and 2.11 show the throughput and latency of these applications in normal execution and when running inside an SGX enclave. Each application, at its peak utilization, is performing hundreds of thousands of Linux API calls, each second. Table 2.2 lists a breakdown of the most frequent API calls in each application, and the execution time spent on facilitating the calls. Using HotCalls and the *No-Redundant-Zeroing* approach we were able to reach a 2.6-3.7x throughput boost, compared to the unoptimized implementation, and reduce the average response latency by 62-74%.

Application	Frequent Calls (Calls x1000 / second)	Total Calls	Core Time

Memcached	read(66.5), sendmsg(66.5)
	RunEnclaveFucntion(66.5)
		200K	42%

OpenVPN	poll(87), time(87), getpid(13.6), write(30), recvfrom(30), read(13.6) sendto(13.6)	275K	57%
Lighttpd	read(49),fcntl(25), epoll_ctl(25), close(25), setsockopt(25), __fxstat64(25) inet_ntop(12),accept(12), inet_addr(12),ioctl(12), __open64_2(12), sendfile64(12) shutdown(12),writev(12)	270K	56%

Table 2.2: Number of API calls (in thousands per second) in non-optimized memcached, openVPN and lighttptd, running inside a SGX secure-enclave. Each ocall, including both software and hardware interfaces (see Section 2.3.3), takes roughly 8,300 cycles (assuming a warm cache). On a 4 GHz core, the execution time is thus $N_{calls} \cdot 8,300$ / $((4 \cdot 10^9))$, which is listed in *Core Time* column.

2.6.1 Efficient Application Porting

SGX Enclaves shield the code running within them from the rest of the system. Communication to and from the enclave is carried out via defining edge functions (ecalls and ocalls). Previous work [224, 40, 34] argued that it is desirable to port applications into an SGX enclave as a whole, minimizing the number of code modifications. Any change to production-grade software may introduce new bugs and potentially new security vulnerabilities. We determined to take a similar approach as the baseline SGX implementation of the applications under test. The main ecall was defined as simply calling the application's original *main* function. We used the makefile provided by the SGX SDK [128] as a guideline for building the enclave shared object. Any OS API call used by the application will result in an ocall.

Identifying API calls: Any call to a function outside the code-base results in an *undefined reference* error at link time. Examples of such functions are *fopen, fread, time, socket* and so forth. Each application under test had between 93-144 such *undefined references*. For each function, it is required to generate a wrapper function that will be executed inside the enclave, and an EDL declaration of an ocall, describing the nature of the arguments (input, output, size of buffers), and finally a landing function in the untrusted code which will call the relevant OS API. We developed a framework to identify the undefined references and generate the needed trusted/untrusted wrapper code. As it is sometimes hard to infer programmatically the input/output nature of an argument, and its size in memory, the framework allows adding exceptions by hand. The wrapper code allows adding counters that estimate the number of calls per second of each function. A breakdown of the most frequent calls per application is presented in Table 2.2.

Marshalling data structures: In HotCalls, only a void pointer is transferred between the enclave and the application (and vice versa). The *edger8r* tool provided by the SDK generates marshalling code to pack/unpack parameters from a structure, perform the necessary security checks, and copy buffers. The framework automatically extracts the marshalling code generated by the *edger8r* tool, to generate wrapper code to be used with HotCalls.

Corner case API calls: Some API calls require special attention. A new thread creation with *pthread_create* eventually calls a function inside the enclave, using a pointer provided to the call in *start_routine*. If *start_routine* points to code residing inside the enclave, this call will fail. Therefore, we created a new *ecall* edge function RunEnclaveFunction, which receives a pointer to code inside the enclave and jumps to it, in the same way *pthread_create* would.

The structures *pthred_mutex_t, pthread_cond_t* are used as synchronization mechanisms. The SDK provides alternatives to these structures: *sgx_thred_mutex, sgx_thread_cond*, that should be used instead. The SDK also provides matching locking/unlocking and waiting/signaling operations. Whenever these synchronization mechanisms were used, we replaced them with the SGX SDK alternatives.

2.6.2 Memcached

Memcached is a key-value RAM database. It is typically used as a caching layer between web-servers and the back end database to boost performance. Memcached is widely deployed: *e.g.*, Facebook, Zynga, Twitter, Youtube [168, 236]. As suggested by *CryptDB* [223], it is desirable to protect the confidentiality and integrity of a database by encrypting it. Completely porting the database application to run within an SGX enclave offers similar security guarantees as *CryptDB*, with minimal development effort.

A thorough workload analysis of deployed memcached instances is provided in [37]. The majority of the requests are under 2KB for the size of the requested value. As a caching layer, memcached performance is measured by the number of requests handled per second and the latency for replying to each request, as observed by the requesting client.

We evaluated memcached version 1.4.31, on the same platform described in Section 2.3. We tested the performance with *memtier benchmark* [231], a tool developed by Redis Labs to evaluate memcached performance. The benchmark used the binary protocol, the ratio of SET:GET was set to 1:1, and the data size of the payload was set to 2KB. A total of 4 million requests were issued, from 4 concurrent threads. We ran memcached with a single thread. To avoid hindering the performance due to link-capacity we ran both memcached and memtier-benchmark on the same machine - using the loopback as the network interface. The original memcached source was able to service, on average, 316,500 requests per second, with average response latency of 0.63 milliseconds. Unoptimized porting of memcached into the SGX enclave, as described in Section 2.6.1, reduced the throughput to 66,500 requests per second, and the average response latency to 2.97 milliseconds, a 79% reduction in serviced requests and a 4.7x increase in latency.

Porting memcached to run inside an enclave exposed 93 external API references. Table 2.2 provides a breakdown of the most frequent API calls: *read* and *sendmsg*. On average *read* and *sendmsg* are called from within the enclave 66,500 times per second. Memcached utilizes *libevent* to wait on a socket, and receives a callback when new data is available. Since

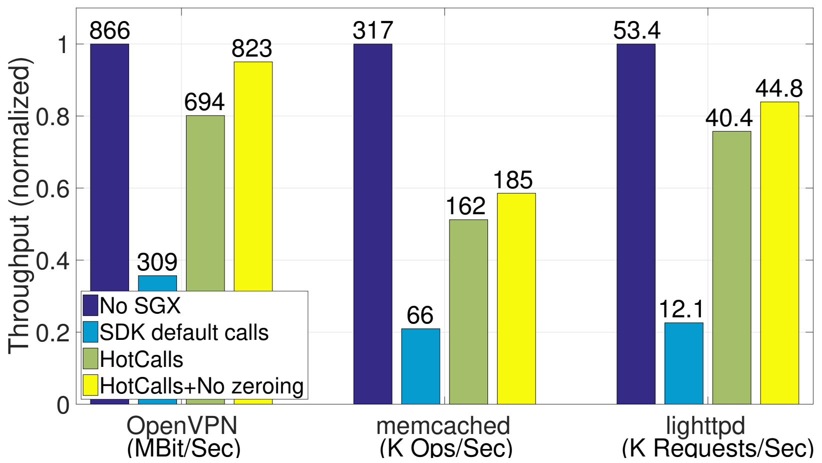


Figure 2.10: Optimizing throughput with *HotCalls* and No-Redundant-Zeroing. The measurements are normalized to running without SGX. Memcached, by nature, is a memory-intensive application, and therefore optimization is limited by the performance of the memory encryption engine.

the callback function is inside the enclave, it requires invoking an ecall, RunEnclaveFunction. RunEnclaveFunction is called on average also 66,500 times per second. To ensure that the counters do not interfere with the performance measurement, we repeated the benchmark with the counting code removed. Measured results were similar.

There are total of 199,500 edge calls per second. As ecall requires at least 8,600 cycles, and ocall requires at least 8,200 cycles, this sums up to over 1.7 billion cycles per second spent merely on transferring control between SGX and untrusted code, not accounting for memory transfers. On a 4GHz core, this is 42% of the core time spent on SGX context switching.

Results: Utilizing HotCalls for *read*, *sendmsg*, and RunEnclaveFunction increases the throughput to 162,000 requests serviced per second, and reduces the response latency to 1.23 milliseconds, a 2.4X increase in throughput and 58% decrease in latency. Figures 2.10 and 2.11 depicts HotCalls impact on throughput and latency. The *read* API call is receiving a buffer from the network, therefore using the *out* category of ocalls. Removing the redundant zeroing performed by the SGX generated code increased throughput to 185,000 requests per second, and reduced response latency to 1.08 milliseconds. This is 2.8x increase compared to SGX SDK calls, and a 64% reduction in latency.

Fundamental limitation: Even with HotCalls, the throughput is still only around 60% of the non-SGX baseline. Memcached is by nature memory intensive. The memory accesses are uniform across the memory-stored database, leading to poor spatial locality, and therefore suffer from many cache misses. As suggested by the memory access microbenchmark and *mcf* benchmark in Section 2.3, memory access latency of 2KB buffers may be slowed down up to 55%, potentially causing throughput degradation up to 35%, just accounting for memory access overhead. This is inherent to using encrypted memory. Recent work has suggested a speculative loading mechanism to improve the performance of encrypted memory [112], and may improve the performance of memory intensive applications such as memcached.

2.6.3 OpenVPN

openVPN [213] is a well known widely used open-source Virtual Private Network (VPN) solution that provides secure encrypted tunnels between two endpoints connected to the Internet. openVPN utilizes OpenSSL library [212] as the cryptographic implementation. Compromising the secret keys used by openVPN compromises the security of the tunnel, potentially allowing an outsider to inspect tunnel communication or inject traffic. Therefore, it may be desirable to port openVPN into an SGX enclave to protect encryption and authentication keys.

We evaluated openVPN version 2.3.12, using OpenSSL version 1.0.2. We created a secure tunnel between the SGX machine specified in Section 2.3, and a desktop machine Intel NUC 5i7RYH with 16 GB of DDR3, running Ubuntu Desktop 16.04 LTS, connected in a 1 Gbit/sec link. To evaluate the throughput, we used *iperf3* [144]. We ran *iperf3* for 60 seconds to estimate the actual maximum TCP bandwidth between the desktop and the SGX machine and found it to be 935 Mbit /sec. We then ran *iperf3* over the openVPN tunnel (without modification) and measured a TCP bandwidth of 866 Mbit /sec, showing that the Ethernet link is not fully saturated. Unoptimized porting of openVPN into the SGX enclave,

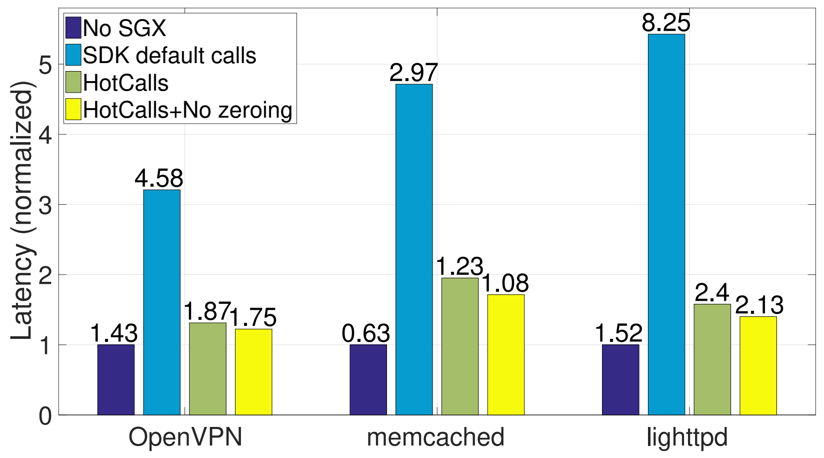


Figure 2.11: Optimizing latency with *HotCalls* and No-Redundant-Zeroing. The values above the bars are in milliseconds: For openVPN the values are the average ping round-trip time. For memcached and lighttp the values are the server's average response latency.

as described in Section 2.6.1, reduced the TCP bandwidth to 309 Mbit/sec, a 64% decrease. We estimated the round trip latency using a flood ping, sending 1 million requests, with preload of 100 requests before waiting for the response. For native openVPN, the average round trip latency was 1.427 milliseconds. The SGX implementation increased the average latency to 4.579 milliseconds, a 3.2x increase.

Porting openVPN to run inside an enclave exposed 131 external API references. Table 2.2 provides a breakdown of the most frequent API calls, totaling roughly 275,000 ocalls per second, wasting 57% of the core time merely on SGX context switches. A surprisingly frequent API call is *getpid*. This call is invoked by OpenSSL whenever a cryptographic context is called. Other frequent API calls are *inet_ntop, inet_addr*. These are utility functions, which could be implemented inside the secure-enclave to reduce the number of ocalls and improve performance.

Results: Utilizing HotCalls for all seven frequent API calls, increases the bandwidth to 694 Mbit/sec, and reduces the round trip latency to 1.873 milliseconds, a 2.25x increase in bandwidth and 60% decrease in latency. Results are depicted in Figures 2.10 and 2.11. The calls *recvfrom and read* receive a buffer from the untrusted code, therefore using the SDK *out* category of ocalls. Removing the redundant zeroing performed by the SGX generated code increases bandwidth to 823 Mbit/sec and reduces the round trip latency to 1.747 milliseconds: a 2.66x bandwidth increase and 62% decrease in latency.

2.6.4 lighttpd

lighttpd [167] is an open source, light-weight web server optimized for speed and serving many concurrent requests. It runs single-threaded in a single process. We evaluated lighttpd version 1.4.41. We evaluated its performance using *http_load* [7]. The measurement consisted of 100 concurrent clients connections, fetching a total of 1 million 20 KB pages. The connections were over the local loopback to maximize available link bandwidth. Unmodified *lighttpd* was able to serve an average of 53,400 pages per second, with average response latency of 1.52 milliseconds.

Unoptimized porting of *lighttpd* into SGX enclave, as described in Section 2.6.1, reduces the number of requests per second to 12,100, and increases the response latency to 8.25 milliseconds, a 77% decrease in throughput and 5.4x increase in latency. Porting *lighttpd* to run within an enclave exposed 144 external API references. Table 2.2 gives a breakdown of the most frequent API calls, totaling in roughly 270,000 ocalls per second, costing 56% of the core time. The API calls *inet_ntop*, and *inet_addr* don't require OS involvement and can be implemented inside the enclave, reducing by 9% the number of ocalls. The rest of the calls, however, do require access to external OS resources, and can not be optimized into the enclave.

Results: Utilizing HotCalls for all 14 frequent API calls, increases the throughput to 40,400 requests per second and reduces the response latency to 2.40 milliseconds, a 3.3x increase in throughput and 70% decrease in latency. Results are depicted in Figures 2.10 and 2.11. The calls *read* and *inet_ntop* receive a buffer from the untrusted code, therefore using the SDK *out* category of ocalls. Removing the redundant zeroing performed by the SGX generated-code increases throughput to 44,800 requests per second, and reduced the response latency to 2.13 milliseconds, a 3.7x increase in throughput and 74% decrease in response latency compared to an unoptimized SGX implementation.

2.7 Related Work

SGX related sources: The SGX official documentation is in Intel's Software Developer Manual [137]. Explanation of the technology can be found in "Intel SGX explained" [69], and in a presentation given at Black Hat [12]. None of these documents provide specific measurements for the various operations supported by SGX, nor suggest optimization strategies.

Similar technologies: *Sanctum* [70] is an alternative secure execution technology for the RISC-V [278, 35] open processor. *ARM Trustzone* is ARM's secure execution solution [19], allowing co-existence of a *secure world* OS, in parallel to the *normal world* OS to manage sensitive data and operations. SecureME [64] introduced a potential architectural support for protecting user space applications' memory from other applications and privileged code such as the OS, VMM or SMM code. Overshadow [63] proposed a VMM enforced mechanism to offer similar protection, but only against other applications and the OS.

Previous work on SGX: Drawbridge [224] is a solution for moving most of the OS operations into user space to improve security. Haven [40] makes use of Drawbridge in order to port entire applications with most of their OS-support needs inside an SGX enclave. VC3 [240] is an implementation of map-reduce across multiple servers, running on SGX secure-enclaves. SCONE [34] shows how to run docker containers inside a secure-enclave, and suggests techniques to improve the container performance. These solutions are very effective but are mainly applicable to applications running inside docker. Ryoan [115] offers a framework that allows one enclave to trust another enclave from a different provider, to receive secret data, guaranteeing that it would not be leaked. A possible side-channel attack against SGX is described in [286], exploiting the leakage of memory access patterns in page granularity, *i.e.,* 4 KB. Although all the above mention performance implications of using SGX secure-enclaves, no specific measurement of operations is provided. Therefore, it was not previously clear what optimization strategies should be made to improve performance when developing enclaves.

Related optimization work: The approach of polling for events, instead of being called, was suggested for use with hardware in Linux NAPI [238], to minimize the need for a context switch to interrupt handlers. In the virtualization context, this approach has been investigated in [159, 238] as a way to accelerate accessing hardware or virtualized hardware, saving the costly context switch between guest OS and hypervisor. The implementation of *HotCalls*, suggested in this chapter, is inspired by these approaches.

2.8 Conclusion

Identifying the bottlenecks of Intel's SGX technology is an important step to make secure computation practical. Leveraging the insights from the microbenchmarks allows developers to focus the optimization effort effectively. *HotCalls* is an alternative calling interface for SGX secure-enclaves, which optimizes the calling latency by a factor of 13-27x. Using *HotCalls*, the throughput of common applications can be boosted by up to 3.7x, and the response latency can be reduced by up to 74%. In order for security-enhancing technologies to be prevalent, they need to be practical. The best practices discussed in this chapter and *HotCalls* are an important step in making SGX both a secure and a practical solution. The HotCalls mechanism presented in this chapter influenced Intel's design of *switchless calls* in the SGX SDK.

CHAPTER III

TEE Service: Practical Trusted Execution Services for the Cloud

3.1 Introduction

In the previous chapter, I analyzed the impact of SGX on applications' performance when they execute on a single machine. In this chapter, I analyze and address the challenge of managing distributed secure services using SGX across multiple machines.

To protects their clients data, service providers traditionally opt to run their software in private clouds [33, 76, 192, 229, 259, 274]. And yet, because of the difficulty of securing infrastructure, it is unclear that private clouds are actually less risky [81, 198, 199].

Trusted Platform Modules (TPMs) and Hardware Security Modules (HSMs) help reduce the trust in the software stack, but are slow and/or expensive [6, 38]. While TPMs can perform cryptographic operations to verify the boot-software integrity, they are far slower than CPUs: *e.g.,* five RSA-2048 bit signatures / second [38] vs 4,000 (§3.7). Similarly, HSMs are expensive ($2,500–$30,000 [6]) and require physical deployment. Hence, they are typically used only when required by regulation, such as for root certificate authorities and the payment card industry [219]. Low-end HSMs yield poor cryptographic operation throughput relative to general-purpose processors (70-95% less, see §3.7), making them impractical for high-throughput applications (*e.g.,* servers, databases).

Although promising, the mechanisms provided by TEEs—code integrity and secure memory—are, by themselves, insufficient for a trusted multi-party distributed service. Distributed services involve multiple interacting entities—the service *owner* and various mutually untrusting *clients*—each with different security objectives. At the same time, frequent software update is the norm in a cloud service. The *owner* wants convenient and secure *client* access control and the ability to upgrade and maintain software. *Clients* must validate service identity and maintain trust across service updates.

Stateful services and software updates open the possibility of rollback attacks that are not sufficiently precluded by existing TEE primitives. SGX's native rollback protection mechanism, monotonic non-volatile counters, are provided by the *Platform Software* (PSW). PSW is implemented outside of SGX TCB, in the Management Engine (ME [5]), which has been compromised in the past [2, 3]. Other solutions [47, 115, 180] either (1) rely on client trust in at least a subset of TEE machines, (2) require multi-client collaboration, (3) lack access control mechanisms, and/or (4) provide no means for clients to track service state. As such, these solutions may not be readily applicable for cloud computing. For instance, it is unclear if services' clients (*e.g.,* bank clients) have any incentive to trust each other.

In this chapter I propose TEE-SERVICE, a framework that allows machines without local HSMs or TEE support to request secure services. In this paradigm, a TEE-SERVICE-*client* communicates with one or more TEE-enabled machines that act as TEE-SERVICE-*server*. The *owner* (administrator) of a TEE-SERVICE can define which functionality is available for which *client*.

TEE-SERVICE works by decoupling management of permissions, audit of service state, and secure communication (§3.4, §3.6). In TEE-SERVICE, permissions are managed by the *owner*, while *clients* audit service state. TEE-SERVICE provides the same security guarantees that programs using a local HSM or TEE enjoy, while overcoming the practical challenges of cloud deployment. TEE-SERVICE *owners* can securely manage fine-grained client permissions for different services. In turn, TEE-SERVICE clients can audit that state-modifying operations they perform are not reverted and that the trusted software was not modified. TEE-SERVICE automatically notifies clients of state and software updates and allows them to approve the changes. Finally, TEE-SERVICE also allows TEE-enabled machines in the data center to have access to a distributed secure computation service. Consequently, TEE-enabled machines are not limited to their local resources and can delegate secure execution to remote TEEs.

To demonstrate the effectiveness of TEE-SERVICE, we built TEE-CRYPT, a service for remotely performing cryptographic operations, much like the functionality of a local HSM. These operations include RSA and ECDSA [201] signing, AES-GCM [204] operations, and AES-CMAC [118]. TEE-CRYPT provides stronger security guarantees than software-only security services [9, 211, 212] and outperforms HSMs. In particular, TEE-CRYPT is faster than HSMs by 4×-8× for RSA and ECDSA operations, 2.7×-14.9× for AES-GCM encryption—at the fraction of the cost (500$ for an SGX-capable PC [23] vs. 2,500$ for a low-end HSM [6]). We integrated TEE-CRYPT into *NGINX* [200] and *Lighttpd* [167] web servers and Linux Pluggable Authentication Module (PAM [170]), demonstrating TEE-CRYPT is a viable solution to augment systems' security. TEE-Service is fully functional and ready to be open-sourced.

To summarize, in this chapter I make the following contributions:

- TEE-SERVICE—a framework that builds upon TEEs to enable multi-party distributed services with trusted access control, software update, and use of secure services.
- TEE-CRYPT—a prototype that uses TEE-SERVICE to perform remote cryptographic operations, much like a local HSM, but with better performance and lower cost.
- A demonstration of TEE-CRYPT with popular web servers, to show that TEE-SERVICE incurs reasonable overhead while improving security.

3.2 Challenges of Building TEE-SERVICE

TEE-SERVICE must address the practical challenges of multi-party distributed cloud computing while maintaining trust. We now elaborate on the specific objectives.

State Freshness. The first challenge is to assure clients that any state-modifying operation they perform cannot be reverted. While some TEE's protect against state rollback when it is stored in RAM, there is no such guarantee when the state is backed in persistent storage. SGX provides persistent storage via the *sealing* API [29, 131]. However, data stored to the file system via the *sealing* API has no rollback protection, rooted inside SGX TCB, which implies that the state of the service may be reverted when it is stored on persistent storage (PSW monotonic counters are outside of SGX TCB and are a limited resource). For instance, in a trusted cryptographic operations service, signing keys may be revoked. However, a rollback attack on the persistent storage where key information is kept can restore a revoked key.

A straw-man solution for preventing state rollback is by using the monotonic counters provided by the Intel SGX SDK Platform Software [131]. However, the SDK implementation of the counter depends on Intel Management Engine (ME), which has been compromised (see CERT and CVE reports [2, 3]). The ME lies outside the SGX TCB [5], and hence cannot be trusted for rollback protection in the context of the SGX threat model. Moreover, the SDK manual [131] states "SGX limits the number of monotonic counters (MC) an enclave can create", making monotonic counters a scarce resource.

Existing rollback solutions, such as ROTE and LCM [47, 180], ensure state freshness by using an interactive protocol among various clients. However, collaboration among clients may not be feasible in a cloud computing scenario, where clients are independent entities. For example, multiple clients of a bank may not be interested in communicating with each other. We address the *State Freshness* challenge without an interactive multi-party protocol by relaxing the requirement for state freshness to only hold for each client individually without sacrificing correctness of the overall state. TEE-SERVICE provides a mechanism for clients to verify that program state (from their perspective) only progresses forward (§3.4.5). Additional details about prior work such as ROTE, LCM, and Ryoan [47, 115, 180] and the difference between them and TEE-SERVICE are discussed in §3.8.

Secure Access Control. It is challenging to securely manage access control for users of a *trusted service*, such as TEE-SERVICE. For example, some users may have the permission to only read data (*e.g.,* medical records), while other users may be allowed to perform updates on the data. The administrative mechanism to grant clients access to a trusted service (*e.g.,* a cryptographic operations module) should be equally secure as the trusted service itself. While it is possible to store an Access Control List (ACL) to permanent storage, existing TEEs (*e.g.,* Intel SGX, AMD SEV) do not provide adequate protection against rollback attacks on persistent storage (as explained in *State Freshness* below). For instance, the owner of a service may revoke access permissions of a user, which will update the ACL on file. Even though the ACL file may be encrypted and signed, a user that can access the ACL file (by owning the TEE machine or by attacking it) and rollback the file to a previous version, which is also properly encrypted and signed. After rolling back the ACL file, the user can regain their revoked permissions. We solve this challenge via a mechanism that enforces the approval of the ACL version by the owner (or its delegate) prior to serving any client (§3.4.4).

Trusted Updates. The third challenge is that it is hard to perform service software updates in the presence of TEE signing and attestation. The service owners and clients have different incentives. The owner wants to keep the service software up to date while the clients want to prevent the owner from maliciously updating the software to a version that will leak their private data.

Specifically, while software integrity is technically guaranteed by TEEs, it is challenging for service owners to modify their software while maintaining clients' trust in the system state, and allowing the clients to verify that they are communicating with the *same* server with the *same* state prior to the update. Any modification to the code running in a TEE will generate a new code signature and will, therefore, appear to the clients as a *different* server.

TEE-SERVICE owners should be able to easily upgrade software, while TEE-SERVICE clients need to approve the software update. If the user chooses to approve the software update, they should be able to continue using the service seamlessly, proceeding with the same system state as before the update. We address this challenge by tying the trusted communication keys to the *specific* software-version, while the service state is tied only to the service owner's identity (§3.4.2 and §3.6).

3.3 Background & Threat Model

We now provide a brief background on TEEs and the services they provide followed by a description of the different actors associated with TEE-SERVICE and their interactions as shown in Fig. 3.1. We also explain the different types of attacks and potential compromises to which each actor is susceptible.

3.3.1 TEE Key Properties – Background

As mentioned in the previous chapter, TEE technologies, such Intel SGX [29, 113, 183] and AMD SEV [1, 26], allow the creation of a secure execution environment to isolate critical software from the rest of the system. TEEs also provide mechanisms for verifying code authenticity, proving the authenticity to a remote party, and storing data on persistent storage (though without rollback protection).

Code Authenticity. The TEE allows measuring the code and data loaded into it by computing a cryptographic digest (*e.g.,* SHA-256) on the entire memory contents of the TEE. The digest is the same on every load and is signed by the software developer at development time. When an application is loaded into the TEE, the TEE verifies the signature that came with the code matches the digest it just computed.

Attestation. Attestation allows an application running within a TEE to prove to a remote party that it is indeed running on a genuine and secure TEE, and not on an emulation. During the attestation process, the TEE also provides the remote party with the digest of the code and data loaded into the TEE. However, when the software changes due to a software upgrade, all remote parties need to repeat the attestation process and must verify that they are communicating with *the same* server with *the same* configuration and state.

TEE Encrypted Storage. TEE encrypted storage (*e.g., sealing* [29] in Intel SGX) allows a TEE application to store data to the local file system using an encryption key that can only be re-generated on the *same* machine. Therefore, other machines cannot decrypt the stored data.

3.3.2 The Actors

Service Owner. The *owner* is the entity that implements and manages trusted services (*e.g.,* processing medical images or digitally signing documents with a private key). Code running in TEEs needs to be signed by the owner's private key and the matching public key should be known by all other actors. The *owner* is also responsible for managing the permissions of TEE-SERVICE-*clients* with respect to different services by maintaining an Access Control List (ACL).

TEE-SERVICE Clients. *Clients* rely on TEE-SERVICE for using trusted services. Clients can upload sensitive data to TEE-SERVICE and they can perform queries, computations, and updates on their own data or someone else's data, which they may not be allowed to access directly. For instance, clients of a trusted cryptographic key service can upload a new signing key to TEE-SERVICE and use that key to sign a certificate. Alternatively, clients might be allowed to create and use a key without knowing its value.

TEE-SERVICE Servers. *Servers* are TEE-capable machines running the trusted software managed by the *owner*. The TEE technology on the servers computes a cryptographic digest (*e.g.,* SHA-256) of the code running in the server, which the *owner* also signs with their private key. *Clients* can securely query TEE servers to verify the code's authenticity (*i.e.,* who wrote the code?) and the integrity (has the code been tampered with?).

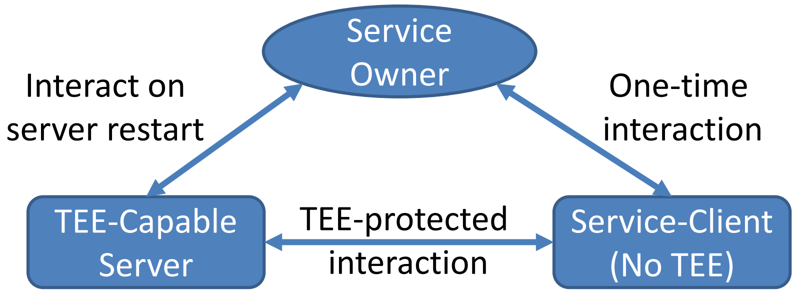


Figure 3.1: The actors: the service owner loads the service software into the TEE-capable servers. The service users remotely access the TEE-SERVICE and have hardware-assisted security via the TEE.

3.3.3 Types of Attacks – Security Analysis

Similar to previous work [34, 40, 115, 180, 240], we consider a powerful adversary that is capable of modifying the OS and the hypervisor, that can mount physical attacks to manipulate main memory contents, and that has complete control over the network traffic. We assume that the TEE guarantees are always maintained: only signed code can execute within the TEE and it is impossible to extract the secure memory contents or revert it to an older state (as opposed to persistent storage content, which the adversary may manipulate as explained in *State Freshness* in §3.2). We now enumerate the consequences of compromising the machines of the *owner*, *clients*, and *servers*.

Compromise of the *Owner's* Machine. We assume that the service *owner* has at least a single machine that can be fully trusted, regardless of possessing a TEE or not. Any compromise of this machine will allow an attacker to add new *clients* and grant them permissions to any service. However, existing *clients'* data which is protected within the TEE will still be secure as long as the private key used for signing the service's code is not compromised. If that signing key is compromised, the clients will still be able to detect software change and decide if they want to trust to new software, before it will be able to access their data.

While the service *owner* is probably initially trusted by the *clients*—the *clients* only accept service software versions (see software update in §3.4.5) which protect the service state with an additional secret provided by the *client*. This scheme prevents the *owner* from updating the software to a version that leaks *client's* secrets.

Compromise of a *Client's* Machine. If a *client's* machine is breached, the attacker can request services from TEE-SERVICE for which the client has permissions granted by the *owner*. For instance, if TEE-SERVICE is providing cryptographic operations, such as encryption and signing, it should be possible for a client, C_g, that is responsible for *generating* keys to upload them into a *TEE-server*. Another untrusted client, C_u, (without access to the key material) can then *use* the keys, *e.g.,* for signing. If the client C_u is breached, they can issue signatures at the attacker's discretion. However, it is not be possible for any *client* to export the key material itself.

Compromise of TEE-*Servers*. If a TEE-*server* is compromised, TEE security guarantees will still hold. In particular, the cryptographic digest of the code in the TEE will remain correct, because it is not possible for an attacker to modify the code once it is loaded into the TEE's secure context. Program state maintained in secure memory is also protected from modification and rollback. However, for persistency, the TEE-*server* must save both the ACL and the service state information to files in permanent storage. If the TEE-server is breached, we assume the attacker can modify the files or restore them to an older version.

SGX's threat model explicitly excludes cache timing, branch-shadowing, page-table, and synchronization attacks [53, 100, 164, 217, 243, 279, 286]. It is up to the implementation to mitigate the risk created by these attacks, as instructed by Intel's enclave writer's guide [130]. Defending against these attacks is orthogonal to the properties of TEE-SERVICE, which is not in the scope of this chapter.

3.4 TEE-SERVICE Design

We now describe the design of TEE-SERVICE and how it addresses the challenges discussed in §3.2. In TEE-SERVICE, every actor uses an identifier, *A-ID*, to authenticate their identity with other actors (§3.4.1-§3.4.2). Every *TEE-server* has exactly one *owner*, which maintains a list of registered *clients* and their respective ACLs (§3.4.3-§3.4.4). The service *owner* is only conceptually unique. In practice, the owner could reside in multiple machines with replicated state (via its delegates, §3.4.4) to avoid being a single point of failure. The *clients* know the A-ID of the *owner* and that of the *TEE-servers* they want to use. To allow *clients* to audit the service state, each *server* maintains a service-specific state with a revision number that the *clients* can monitor (§3.4.5).

We show the high-level design of TEE-SERVICE in Fig. 3.2, and the detailed properties and relations of the actors we introduced in §3.3.2. In addition to the service *owner*, *clients*, and *TEE-servers*, we show another actor, namely *delegates*, which are permitted to perform certain operations on behalf of the service *owner*, preventing the *owner* from becoming a performance bottleneck. For example, the *owner* can permit a *delegate* to approve the ACL version of a *server* (§3.4.4).

3.4.1 Actor Identity – A-ID

TEE-SERVICE allows actors to identify each other using an Actor-ID (A-ID). A-ID is an integral part of the TEE-SERVICE secure communication protocol and therefore cannot be forged (see §3.4.2). To enable secure communication, every actor generates private-public key-shares, which will later be used to establish shared communication secrets with other actors. To couple actors' identities with the secure communication secrets, TEE-SERVICE uses the cryptographic digest of the actor's public key share as A-ID.

For instance, in Intel SGX [131], the key sharing cryptographic algorithm is Elliptic Curve Diffie-Hellman (ECDH) [202], and the private and public key shares are a, Ga, respectively. The A-ID is then computed as the SHA-256 digest of the public key share Ga, namely A-ID=SHA-256 $(G \cdot a)$. This definition of A-ID couples an actor's identity to the shared secret established with it.

In TEE-SERVICE, the *owner* advertises its *A-ID* to the *clients* in a secure way (*i.e.,* by publishing a certificate of the A-ID signed by a known certificate authority). When the

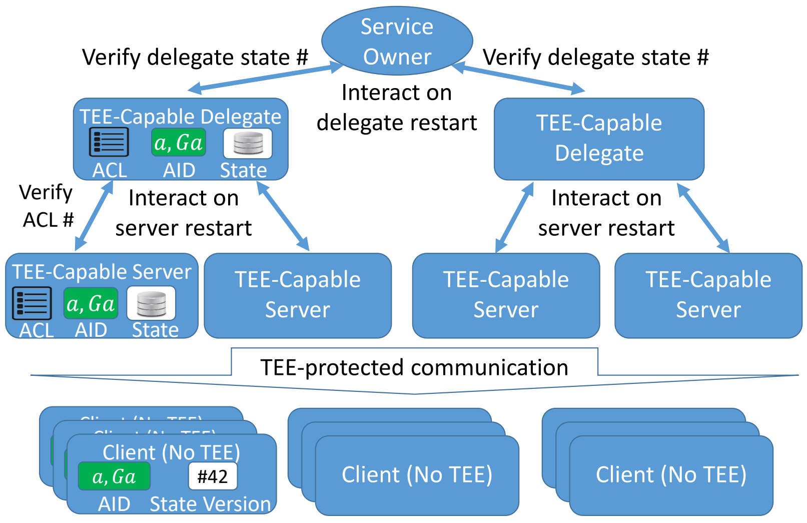


Figure 3.2: TEE-SERVICE Architecture.

owner contacts a *TEE-server* for the first time, the *server* generates its private public key shares, derives its own *A-ID*, and records the *owner*'s A-ID. In turn, a *client* communicates its *A-ID* to the *owner* when registering to use TEE-SERVICE for the first time.

3.4.2 Trusted Communications

Secure communication between actors is key to the security of TEE-SERVICE. While it is desirable to simply use existing TLS implementations [119], these do not provide means for authenticating the TEE hardware or the specific software on top of the TLS secure session.

Actors must verify the identity, A-ID, of the party with which they communicate, and trust that the communication is confidential, integrity protected, and not a replay of an older communication. TEE-SERVICE actors communicate securely by establishing a shared cryptographic secret (*e.g.,* using ECDH [202]). The TEE-SERVICE protocol augments the SGX remote attestation [131, 138] to address the challenges discussed in §3.2. For instance, using vanilla SGX attestation does not allow a remote party to identify the specific *server*, *i.e.,* that this physical *server* is managed by the service *owner*, or that this is the same *server* that the *client* communicated with prior to a software update.

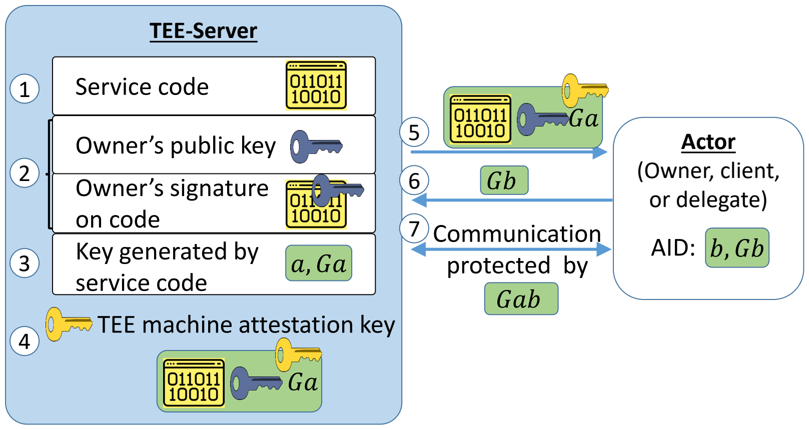

TEE-Server
1
Service code
011011 10010
2
Owner's public key
Owner's signature on code
3
Key generated by service code
a, Ga
4
TEE machine attestation key
Ga
5
Ga
6
Gb
7
Communication protected by
Gab
Actor
(Owner, client, or delegate)
AID: b, Gb

Figure 3.3: Using TEE *attestation* to establish secure communication (§3.4.2).

Bootstrapping Trust Between a *Server* and a *Client*. When an actor communicates with a *TEE-server* for the first time, the *TEE-server* uses TEEs' *attestation* mechanisms [1, 106, 138, 139] to prove that it is running in a genuine TEE and to establish a shared secret with the actor, as shown in Fig. 3.3: The *TEE-server* loads the trusted service code into the TEE's secure memory (step 1). To verify the authenticity of the code, the TEE checks that the code's signature matches *owner*'s public key (step 2). To establish a shared-secret for secure communication, the service generates a separate private-public key share a, Ga (step 3). To prove that the service code is indeed running in a genuine TEE hardware, the *TEE-server* uses the TEE's *attestation* API to access the *attestation key*. The *server* uses the *attestation key* to sign the blob comprising the trusted service code, the *owner*'s public key, and the key share Ga (step 4) and sends it to the *client* (step 5). The *client* can then verify the attestation signature [1, 139] on the blob and trust that the public key share Ga (and hence the *server* A-ID) belongs to the *specific* service code with the matching *owner*'s signature, running on a real TEE. The actor sends the *server* its public key share Gb (step 6), which is then used by the *server* to compute the actor's A-ID. The *server* verifies that the actor represented by the computed A-ID is permitted to receive services. The rest of the communication is secured by the master secret Gab (step 7).

Secure Communication Session. After establishing the shared secret, Gab, (step 7 in

Fig. 3.3), the *client* initiates a communication session with the *server* to request services. The *server* and the *client* use the shared-secret together with random nonces they generate to create an ephemeral AES-GCM session-key according to NIST guidelines [62, 204] and similar to TLS 1.2 [119]. Every message has a sequence number and AES-GCM tag to prevent replay or tampering.

Securely Storing the Communication Secrets. To enable future communications with the same *clients*, even after a server restart, the *server* needs to persistently store the shared secrets. TEEs provide a mechanism to store data on persistent storage using a key that is derived from the TEE-machine's identity, the digest of the software running in the TEE, and the *owner*'s public key that is used to sign the software [26, 29] (step 2 in Fig. 3.3). In TEE-SERVICE, the shared-secret, *Gab*, is stored to persistent storage in a way that only allows the *specific TEE-server* with a *specific* software version to retrieve it whenever TEE-SERVICE restarts. If the *owner* updates *TEE-server*'s software, the shared secret can no longer be loaded from persistent storage, forcing the *server* and the *client* to re-establish the shared secret. Since the *client* has to participate in re-establishing a new secret, it will detect the software change and can refuse to communicate with the new software. Section §3.4.5 further elaborates how TEE-SERVICE securely maintains the service state after software update.

3.4.3 The ACL – Granting Access to Services

In TEE-SERVICE, the *owner* can manage what services are accessible to which *clients*. The *owner* manages *client*-specific Access Control Lists (ACLs) on every *server*, identifing *clients* by their respective A-IDs.

***TEE-Server* Setup.** The first actor to communicate with the *server* is considered its *owner*. During the attestation process [1, 138, 139] (Fig. 3.3) the *owner* records the *server*'s public key, *Ga*, and computes the corresponing *server*'s A-ID. The *owner* sends the *server's* AID to the *clients*, which are allowed to request services from the specific *server*. The *owner* then registers with the *TEE-server* the A-IDs and permissions of the *clients*.

Accepting New *Clients*. After setup, the *TEE-server* only accepts communication from *clients* with public keys matching the A-IDs registered by the *owner*. When a *client* contacts the *TEE-server* for the first time, she verifies *server*-machine is the one intended by the *owner* in two ways. First, the *client* derives the A-ID of the *server* from the *server*'s public key, *Ga*, and compares it to the *server*'s A-ID that was reported by the *owner* earlier. Second, the *client* requests the *server* to report its *owner*'s A-ID, and compare it to the *owner*'s identity it received when registering for the TEE-SERVICE. Since the *server* software is running on a genuine TEE (as proven by the attestation process) the *client* trusts that the *server* cannot lie about its *owner* identity.

3.4.4 Preventing ACL rollback

For ACLs to persist across server reboots, they are securely serialized to the file-system using the TEE's serialization mechanisms [1, 29, 131] (*e.g., sealing* in Intel SGX). TEE serialization mechanisms provide integrity protection but do not defend against an adversary reverting a file to an older version. The lack of rollback protection presents an attack vector on ACLs. For instance, if the *owner* decides to revoke access permissions for a specific *client*, the access-revocation can be reverted by replacing the ACLs with older versions.

TEE-SERVICE addresses the *Secure Access Control* challenge (§3.2) by allowing the *owner* to verify the version of the ACLs before the *server* is allowed to serve *clients*. In TEEs, the integrity protection of securely stored files is provided by a cryptographic MAC. TEE-SERVICE uses the MAC of the ACLs as version information. Whenever the *server* is restarted, the *owner* checks that the version is correct, and only then allows the *server* to serve requests. Since the *server* runs inside a TEE, its code cannot be modified to falsely report the version or bypass the service restriction until the *owner's* approval is received. Lastly, since the communication between the *owner* and the *server* is replay-protected (§3.4.2), it is impossible to replay the *owner's* approval of the MAC at a later time. In the threat model (§3.3), we assume the *owner's* machine is trusted and cannot be reverted.

Requiring all *servers* to contact the *owner* upon every restart is a performance bottleneck. To overcome the bottleneck and make TEE-SERVICE scalable, the *owner* can delegate the approval of the ACL-versions to special *delegates*, as depicted in Fig. 3.2. *Delegates* are *clients* of TEE-SERVICE which are only allowed to approve ACL versions. *Delegates* run inside a TEE (and hence are trusted) and securely save the appropriate ACL-versions of *TEE-servers* on local storage. To prevent rollback of the *delegate*'s state, the owner verifies delegates' state version before they can approve other *TEE-servers*. Upon an ACL update, the *owner* updates all the delegates with the latest approved ACL version.

3.4.5 Non-revertible Service State

State Revision Number. We now describe how TEE-SERVICE addresses the *State Freshness* challenge (§3.2). For the service to be useful, its state (*e.g.,* medical records, bank account balances, or cryptographic keys) needs to be persistent across restarts. In TEE-SERVICE, the state is securely stored using the TEE storage API (*e.g.,* SGX *sealing*), which exposes the service state to rollback attacks.

To detect state rollbacks, TEE-SERVICE maintains a per *client* state revision number representing the state version. Every state-modifying operation advances the revision number of the affected *client*(s). When a *client* requests a service (either state-modifying or state-preserving), the *server* replies to the request with the current state-version, which the *client* then records. The state-version should always advance as a monotonic non-decreasing counter. If a *client* receives a state-version lower than expected, she detects that the state has been rolled back. Since the Management Engine (ME) can only support a limited number of monotonic counters (see §3.2) the ME may not be suitable for keeping track of clients' state revisions.

In §3.5, we discuss a TEE-SERVICE use-case of a service providing cryptographic operations. The classification of operations for this use case is depicted in Table 3.1. In this example, the operations for creating new keys are considered state-modifying services, while the operations which depend on existing keys (encryption, signing *etc.*) are considered state-preserving operations.

Limitations. In TEE-SERVICE, we assume that every client is only interested in making sure that their own transactions were not reverted. Namely, when a *client* C_0 performs two operations, they want to verify that the second operation is executed on a state which represents the execution of previous operations they requested. Between these two operations, another *client* (C_1) may have requested their own operations, which may have been reverted by an attacker. However, by inspecting the state revision number, client C_0 trusts that their prior operations have not been reverted. C_1 can also inspect the state to detect that it has been reverted. We discuss how *client*s can collaborate to receive stronger guarantees in §3.8.

Enabling *Server*'s Software Update. We now explain how TEE-SERVICE addresses the *Trusted Updates* challenge (§3.2), and allows continued operation even if the *server* software is updated. While TEEs ensure code integrity, it is up to the service implementation to provide a secure way to continue operation, with the same state, after a software upgrade.

To solve this challenge, TEE-SERVICE uses different keys to store the service state and the communication keys. The service state is secured using a storage encryption key, which is derived from the TEE-machine identity, the *owner*'s public key (which was used to sign the software), and a secret-share provided by the *client*. This storage key is independent of the *server*'s software version. Using this storage mechanism allows future versions of TEE-SERVICE to load the service state from persistent storage given the following conditions: it runs on the *same* physical machine, the *server*'s software was signed by the same *owner*'s public key, and the *client* gave permission (via providing her secret-share).

The communication keys (*Gab* in Fig. 3.3), however, are stored using a key derived from the specific software version and cannot be reloaded by the updated software. This scheme prevents the *owner* from updating the software without notifying the clients: When a *client*

	#	Service Name	C_0 permissions	C_1 permissions
State-modifying	1	GEN_KEY_RSA	Allow	Deny
	2	GEN_KEY_ECDSA	Deny	Deny
	3	GEN_KEY_AES	Allow	Deny
	4	UPLOAD_KEY	Deny	Deny
State-preserving	5	SIGN_RSA	Allow	Allow
	6	SIGN_ECDSA	Deny	Allow
	7	ENCRYPT_AES_GCM	Allow	Deny
	8	DECRYPT_AES_GCM	Allow	Deny
	9	AES_CMAC	Allow	Allow

Table 3.1: ACLs of example TEE-CRYPT-clients (C_0, C_1). Services 1–4 are considered state-modifying as they alter encryption keys. Requests for these services will advance the state revision counter. Services 5–9 depends on the service state, but don't modify it (§3.4.5).

communicates with the server with new software, it has to negotiate new communication keys. The *client* validates the server's identity via the *server*'s Actor-ID, verifies it accepts the software update and then establishes new communication secrets. More details regarding such persistent storage keys in Intel SGX are described in §3.6.

3.5 TEE-SERVICE Use Cases

TEE-SERVICE is useful for managing services where different clients have different access permissions and security objectives, such as managing bank accounts, medical records, and cryptographic keys. We will focus on the cryptographic services use case as the services and their performance can be compared to existing hardware solutions—HSMs. We present and analyze the implementation of TEE-CRYPT—the example secure service that we built using TEE-SERVICE.

TEE-CRYPT provides cryptographic services, similar to software solutions such as OpenSSL, SoftHSM, and BouncyCastle [9, 211, 212], but with stronger security guarantees. A system breach of any of the above solutions immediately enables the attacker to retrieve the cryptographic keys, while in TEE-CRYPT, they are protected within the TEE. For example, TEE-Crypt can protect *OpenSSL* [212] from security bugs such as *HeartBleed* [74]: The *HeartBleed* vulnerability allows an attacker to read data from the web server's heap (through the *Heartbeat* message), which may contain private keys. If OpenSSL is integrated with TEE-Crypt (as we evaluate in §3.7.2) the memory containing the key material is protected within the TEE.

At the other extreme, Hardware Security Modules—HSMs (*e.g.,* [86, 117, 261, 268])—provide strong protection against advanced physical attacks and self destruct if an attacker physically tampers with them [203]. This threat model is not necessarily reasonable for all users.

TEE-Crypt presents a sweet spot between classical software and hardware solutions—TEE-Crypt provides TEE-backed security guarantees at a fraction of the cost of a low-end HSM. Due to the mechanical defenses built into HSMs and their specialized cryptographic hardware (depending on the model), their cost ranges between $2,500–$30,000 [6]. Compared to HSMs, an SGX-capable machine can cost as little as $500 [23].

3.5.1 Securing Web Servers

TEE-Crypt is useful when a distributed set of machines wishes to share crytographic keys while protecting the keys from exposure if any machine is compromised. Moreover, TEE-Crypt can restrict how various actors in the system are allowed to use the keys. In such a scenario, TEE-Crypt can replace many HSMs.

An example use case for such distributed trust management arises in Content Delivery Networks (CDN) such as CloudFlare and Akamai. A CDN company hosts web services for its customers on a remote data center, supporting both non-encrypted (HTTP) and encrypted (HTTPS) access to the websites they host.

Currently, a CDN's customers (the website-owners) must provide the CDN with their private key. This model forces the custumers to trust the CDN's security practices and that the CDN will not give away the customer's private key (*e.g.,* to law-enforcement). If the

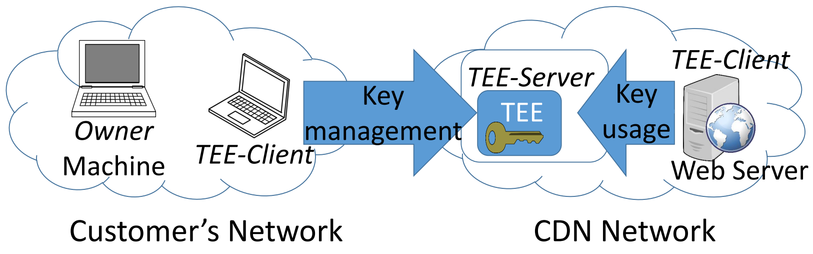


Figure 3.4: Using TEE-CRYPT to protect web servers deployed by a Content Delivery Network (CDN). The *TEE-client* on the customer's side (left) is only permitted to upload keys or generate them within the TEE. On the other hand, the *TEE-client* running on the CDN's network (right) is only permitted to request signing operations using the keys that already exist.

customer's private key is exposed, an attacker can impersonate the customer's web server. The impersonating web server can then be used to retrieve web clients' credentials.

Once the breach is detected, the private key and its matching certificate must be revoked by the customer. However, revoking certificates is not always effective [8, 11], and the attacker may be able to keep impersonating the customer for a while, even after the breach is detected.

TEE-CRYPT Solution. To provide HTTPS communication on CDN servers, TEE-CRYPT supports algorithms such as RSA, ECC [201], and AES-GCM [204]. Table 3.1 lists the different operations supported by TEE-CRYPT. Even if the CDN is breached, there is no need for certificate revocation, because the CDN has no access to the key. Once the breach is mitigated, the attacker will no longer be able to impersonate the web server because they never get access to the private key.

In this TEE-SERVICE use case, the website owner (CDN's *customer*) is the *owner*, and the machines operated by the CDN are the *servers*. There are two classes of *clients* in this scenario, as depicted in Fig. 3.4. The first class of *clients* (left side in Fig. 3.4) are the employees of the website owner who are designated to manage the signing keys. These clients require permissions to generate cryptographic keys or upload them to TEE-CRYPT. They do not need permissions to use the keys themselves. The second class of TEE-CRYPT *clients* (right side in Fig. 3.4) are the web servers maintained by the CDN company. To provide HTTPS access to a website, the web servers need to be able to sign

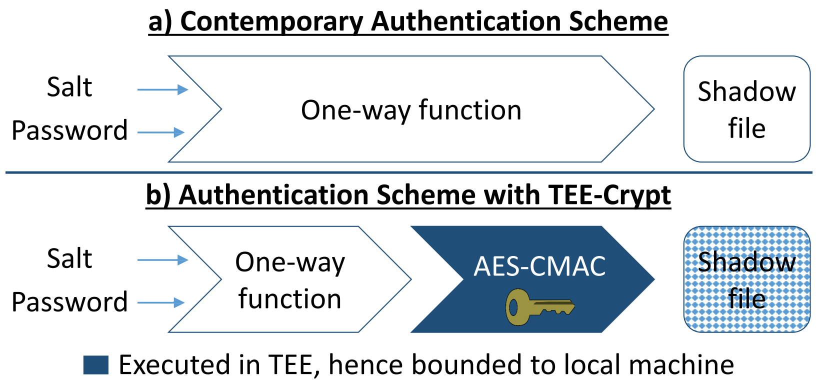

a) Contemporary Authentication Scheme
Salt
Password
One-way function
Shadow file
b) Authentication Scheme with TEE-Crypt
Salt
Password
One-way function
AES-CMAC
Shadow file
Executed in TEE, hence bounded to local machine

Figure 3.5: Eradicating offline dictionary and Rainbow table attacks with TEE-Crypt. a) In contemporary authentication, if the *shadow* file is leaked, it is possible to perform an offline attack to guess passwords. b) Using TEE-Crypt to compute an AES-CMAC on the salted hash and storing only the MAC in the *shadow* file eliminates the ability to conduct an offline attack.

the TLS handshake parameters [119]. From the ACL perspective, the web servers (acting as TEE-Crypt-*clients*) require permissions to use the cryptographic keys, but they don't need permissions to generate, read, or upload keys to TEE-Crypt.

Security Analysis. If the web server machine is breached (acting as the TEE-Service *client*), the attacker can request signing operations from the *server* until the breach is detected. Without TEE-Crypt, the attacker has access to the private RSA key material, therefore the key would have to be revoked. Thanks to TEE-Crypt, there is no need to revoke the key, because it will never be exposed to the attacker.

3.5.2 Secure Authentication

Modern Linux systems employ Pluggable Authentication Modules (PAM) [239] as a unified authenticating mechanism. According to NIST guidelines [108], users' passwords should be saved in a salted, hashed form. Password hashes are typically saved in a file, sometimes known as the *shadow* file. In a typical organization, the IT administrator defines authentication policies that are then enforced by the PAM. When a PAM initiates an authentication request, it salts and hashes the password that the user enters and performs a comparison with the value stored in the *shadow* file (Fig 3.5a).

If a system is compromised, an adversary gains access to the *shadow* file and can mount dictionary or rainbow table attacks [207]. To increase the difficulty of mounting these attacks, the selected hash algorithm is time and memory intensive. However, a hacker can speed up these attacks by leaking the *shadow* file and then use specialized hardware and distributed computing across multiple machines [179], utilizing frameworks such as *John the Ripper* [214]. Because the accelerated computation is done off-site, this is considered an *offline* attack.

TEE-CRYPT Solution. TEE-CRYPT mitigates *offline* attacks on a leaked *shadow* file by binding the one-way hashing computation to a specific physical machine. This concept is illustrated in Fig. 3.5b. The one-way function is extended by computing the AES-CMAC [118] (a keyed hash) on the salted hash, and the resulting MAC is saved to the *shadow* file. When a user attempts to log in to the system using a password, the modified PAM salts and hashes the password as in existing implementations, but also requests TEE-CRYPT to compute AES-CMAC on the salted hash, using a key protected within the TEE. The resulting MAC will then be compared to the value stored in the *shadow* file.

In this TEE-SERVICE use case, the IT administrator of an organization is the *owner* and *clients* are all the systems, running PAMs which authenticate users. The *servers* are TEE-capable machines in the organization running TEE-CRYPT and computing the AES-CMAC.

Security Analysis. If one of the *clients* running PAM is breached, the attacker's ability to guess passwords via a dictionary attack is limited to the maximum request rate of TEE-CRYPT. It is not possible to accelerate the attack by leaking the *shadow* file, as the one-way function can only be computed using the AES-CMAC key that is tied to the *server* machine.

Note that any login attempt performed while the machine is infected will expose the user password as it can be intercepted when it is sent to the server. Using TEE-CRYPT to secure authentication mechanisms builds upon the status quo mechanism of salting and hashing, therefore it does not introduce any new attack vector.

3.6 Implementation

We implemented TEE-Service using Intel SGX. According to the AMD SEV technical preview [1], TEE-Service can also be implemented using SEV. In SGX, the code running within TEE is called a *secure enclave*. We used SGX Linux SDK/PSW version 1.9.100, driver version 0.10. For attestation and shared-secret establishment, we used the SGX SDK's [131] default implementation, which uses ECC cryptography with a key size of 256 bits. The implementation consists of 7566 lines of C/C++ code, and 2848 lines of Python code. The trusted code consists of 3574 lines in C/C++.

To support cryptographic algorithms, TEE-Crypt's implementation uses both Intel's *Integrated Performance Primitives* (IPP [126]) cryptographic library and LibreSSL [166], version 2.4.4. Secure storage of ACLs, shared communication secrets, and the service state is implemented by using SGX's *sealing* API [29, 131]. The sealed contents in SGX's *sealing* mechanism are encrypted using AES-GCM and contain a 16-byte MAC. In TEE-Service, the version identifier of ACLs (§3.4.4) is the MAC computed during the sealing process.

In the SGX API, there are two available mechanisms for deriving sealing-keys for storage. In the first mechanism, the TEE derives the sealing-key from the *owner*'s public key used to sign the *server* software (step 2 of Fig. 3.3). Using this key derivation mechanism, different software versions of the *server* enclave can unseal each other's files, as long as they are signed by the same *owner*'s public key. In the second key derivation mechanism, the TEE derives the sealing-key also from the cryptographic digest of the *server* software. In contrast to the first key-derivation mechanism, only the *exact same server* software can unseal the files it previously sealed. TEE-Service uses both mechanisms for deriving sealing keys to provide *Trusted Updates* (§3.2).

The first challenge for *Trusted Updates* is enabling modification of the *TEE-server* software without having to reconfigure the ACLs and without resetting the service state. Therefore, TEE-Service protects both the ACLs and the service state by using a key that the TEE derives from the *owner*'s public key. This allows future versions of the *server* to unseal the ACLs and the service state when they are loaded from the file system.

The second challenge for *Trusted Updates* is to make sure the *clients* are informed when the *server* code is updated. To this end, TEE-SERVICE seals the shared communication secrets with a sealing-key, which the TEE derives from the specific *server*'s code. This ensures that any future versions of the *server* will not be able to read the communication secrets and will be forced to renegotiate the secrets with the *clients*. Since the *clients* have to participate in the renegotiation, this key derivation mechanism prevents the *server* from updating its software without notifying the *clients*. When a *client* renegotiates the communication keys, she receives the signature of the *server*'s new software (§3.4.2) and decides if she accepts the server's new implementation.

3.7 Evaluation

In this section, we address the following questions about TEE-SERVICE: What is the latency and throughput of requesting services (§3.7.1)? What is the latency and throughput of our case study cryptographic service, TEE-CRYPT, and how does it compare to existing secure alternatives (§3.7.1)? What is the integration effort and the performance impact of using TEE-CRYPT on real applications? Specifically, we evaluate the performance impact of TEE-CRYPT on web servers (§3.7.2) and on the Linux Authentication Module (§3.7.3) we discussed in §3.5.

We evaluated TEE-CRYPT on a Supermicro server with 64 GB DDR4 RAM @ 2133 MHz and an Intel Core I7-6700k 4GHz with 8 logical cores. Dynamic frequency is disabled, and the operating system is Ubuntu server 16.04 LTS. Unless stated otherwise, requests were sent from a separate machine with 32 GB DDR4 RAM @ 2400 MHz and an Intel i7-7700HQ 2.8GHz, over a 1 Gb/sec link. Average link latency is 0.78 *ms*.

3.7.1 Micro-Benchmarks

Latency and Throughput of Native TEE-SERVICE. To evaluate the latency incurred by TEE-SERVICE, we implemented a *ping* operation, which simply echoes the data sent to the *server* back to the *client*. In a loopback test (i.e., excluding network delays), ping latency is 35.09 μ-seconds. The throughput is 55,680 requests/second.

Latency and Throughput of TEE-CRYPT. We evaluated the throughput of TEE-CRYPT for AES-GCM, RSA, and ECDSA algorithms. The implementation of the *TEE-server* consists of a dispatcher thread listening for requests arriving on the network interface and worker threads executing within the TEE.

The throughput of TEE-CRYPT scales with the number of worker threads. We evaluated the throughput of the RSA signing algorithm between two machines for three different key sizes: 2048, 3072, and 4096 bits. Fig. 3.6 depicts the transactions performed per second vs. the number of worker threads. We evaluated the ECDSA algorithm for the following elliptic curves: secp256k1 (256 bits), secp384r1 (384 bit), and secp521r1 (521 bits). Fig. 3.7 depicts the achieved transactions per second vs. the number of worker threads.

We compare the throughput of TEE-CRYPT with throughputs of four HSMs. Fig. 3.8 shows a normalized comparison of TEE-CRYPT and the four HSMs. TEE-CRYPT performs 4$\times$ more RSA-2048 operations per second than the *Gemalto A700* HSM, 8$\times$ more ECDSA-256 operations than *KeyPer Professional* HSM, and 2.7$\times$–14.9$\times$ more AES-GCM encryption operations than *IBM 4767* and *Gemalto A700* HSMs, respectively.

3.7.2 Securing Web Servers' Private Keys

We now evaluate the performance impact of TEE-CRYPT for securing web servers' private keys, as we discussed in §3.5.1. We evaluate TEE-CRYPT's performance impact on two web servers: *NGINX* version 1.12.2 [200] and *Lighttpd* version 1.4.41 [167]. *NGINX* and *Lighttpd* use *OpenSSL* [212] internally to support TLS 1.2 [119]. In an unmodified setup, the web server's administrator would place the server's private key on the server's

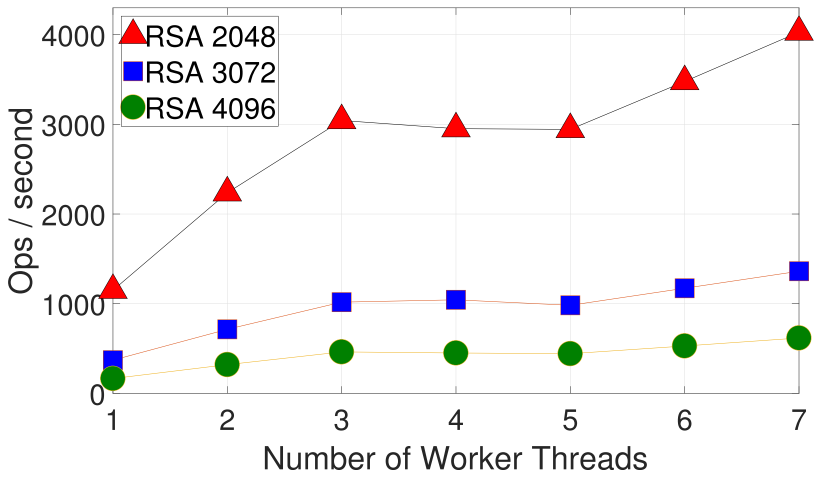


Figure 3.6: TEE-CRYPT-RSA Throughput scalability. The number of threads running is the worker threads + one dispatcher thread. Standard deviation is below 5%.

machine, which will be loaded in plaintext to the web server's memory. When a TLS coonection is made to the web server and the server performs the TLS handshake, the server uses the private key (via OpenSSL) to sign the handshake parameters.

Integration Effort. We modified *OpenSSL* (version 1.0.2j) to utilize TEE-CRYPT for signing operations. The majority of *OpenSSL* is unmodified and run outside of SGX. The web server key pair is generated using TEE-CRYPT and the private key is protected within the TEE. Only the public key, together with a certificate, is exposed to the (potentially compromised) web server application. During the TLS handshake, the modified *OpenSSL* (running outside of SGX) requests TEE-CRYPT to sign the TLS key-exchange parameters. *OpenSSL* then passes the signed key exchange parameters to the web server, which in turns sends them to the web client. The TLS handshake continues unmodified from here on. Modifying *OpenSSL* to use TEE-CRYPT required 130 lines of C code. We tested the patched web servers against Firefox 51.0.1 on Linux and Windows 10.

Performance Impact. We compare the throughput of the modified TLS web servers to the original versions. We use the http_load [7] benchmarking tool to measure the time to complete 10,000 HTTP GET requests (250 in parallel) for a 20 KB file. We evaluate the throughput for RSA keys of the following sizes: 2048, 3072, 4096 bits. The results are

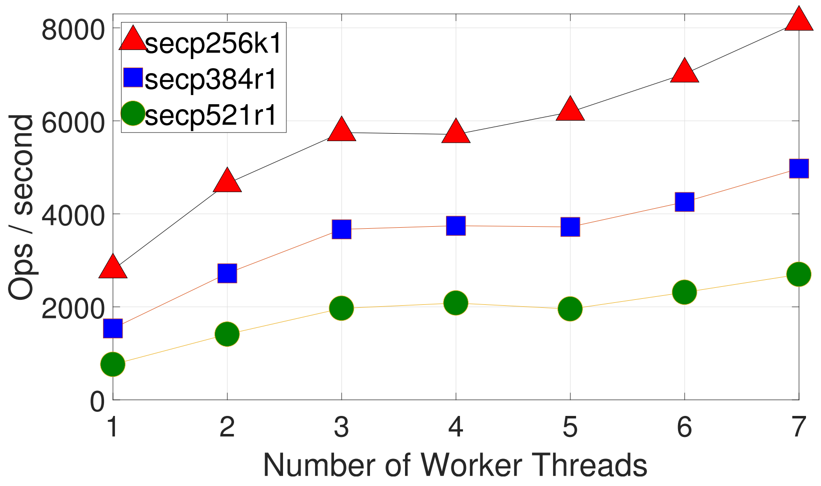


Figure 3.7: TEE-Crypt-ECDSA Throughput scalability, for three elliptic curves: secp256k1 (256 bits), secp384r1(384 bit), and secp521r1 (521 bits). The total number of threads consists of the worker threads + one dispatcher thread. Standard deviation is below 5%.

depicted in Fig. 3.9, showing that TEE-Crypt incurs on average 33% throughput reduction in exchange for adding substantial security.

3.7.3 Securing Authentication Mechanism

We now evaluate the performance impact of TEE-Crypt for securing authentication mechanisms and eradicating offline attacks on the PAM *shadow* file, as we discussed in §3.5.2.

Integration Effort. We implemented a PAM module that mimics the *pam_unix* module behavior, but also computes an AES-CMAC on the salted hashed password using TEE-Service (see Fig. 3.5b). The CMAC key is protected within the TEE, binding it to the physical machine. When a user wishes to authenticate, they enter their password and the PAM module is invoked. Next, the module computes the salted hash in the same way the *pam_unix* module does, and sends an *AES-CMAC* request to TEE-Service with the salted hash. Finally, the resulting MAC is compared to the value stored in the *shadow* file. The TEE-Service PAM was implemented in 147 lines of code.

Performance Impact. We developed a test application to measure the performance of the TEE-Service augmented PAM module. We measure the time for the test application

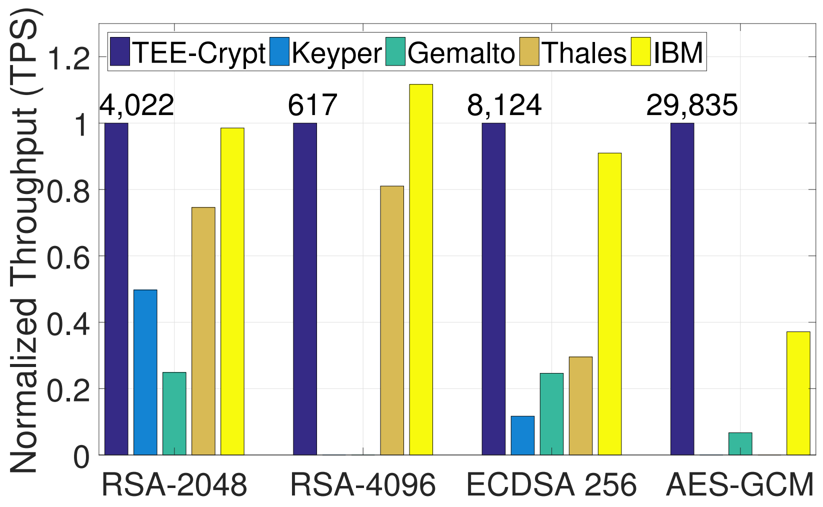


Figure 3.8: Normalized transactions per second (TPS) of TEE-CRYPT vs. 4 Commercial HSMs: KeyPer Professional [268], Gemalto Network HSM A700 [86], Thales nShield 6000+ [261], and IBM 4767 [117] (not all specifications provide numbers for all operations).

to complete 10,000 consecutive authentication requests. The baseline implementation (without TEE-SERVICE) is capable of performing 496 authentications per second, while the augmented PAM module (with TEE-SERVICE) can authenticate 471 users per second; only a 5% degradation in throughput.

3.8 Related Work

TEEs in general and Intel SGX in particular have proliferated research on building secure systems [34, 40, 50, 115, 240, 297], finding new attacks [53, 162, 164, 279, 286] and proposing defenses [47, 160, 169, 180, 246, 250].

Prior work such as *ROTE* [180] and LCM [47] suggested mechanisms to prevent rollback attacks in a multi-party system that runs on TEEs.

ROTE [180] is a mechanism for mitigating rollback attacks in distributed systems using SGX. However, it requires extensive collaborations between the participating entities, and can be defeated if the adversary controls enough SGX-capable machines. A typical deployment of a service is done using a single cloud provider, *e.g.,* Google Cloud, Amazon EC2, Microsoft Azure, etc. In this scenario, the cloud provider can compromise 100% of

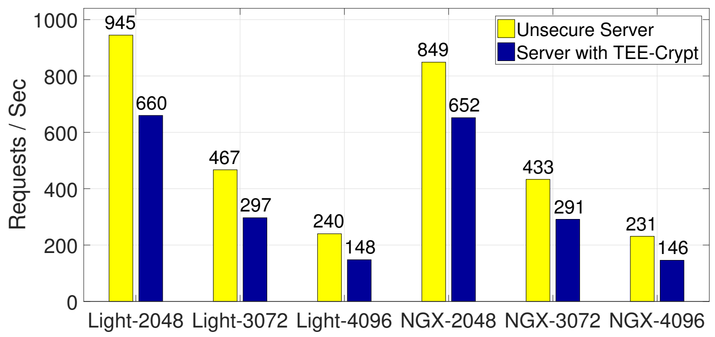


Figure 3.9: Performance impact of TEE-CRYPT on pages served per second over HTTPS for *Lighttpd* (Light) and *NGINX* (NGX) web servers, for three RSA key sizes: 2048, 3072, and 4096 bits. TEE-CRYPT incurs reasonable overhead, while substantially improving security.

the machines, defeating the distributed trust mechanism. TEE-SERVICE takes a different approach of assuming that the *owner* has at least a single machine that is not compromised, and thus provides a simpler solution with less mandatory coordination between the enclaves.

LCM [47] is a technique for collaborating clients to detect rollback attacks. However, while clients are interested in verifying that their transactions have not been reverted, they may not be willing to invest the resources required to coordinate with other clients. TEE-SERVICE provides a means for every client to ensure that their transactions were not reverted, without contacting other clients. TEE-SERVICE can potentially be extended to support collaboration among clients by allowing them to receive from the *server* a hash of all the version numbers, and compare it with each other.

Ryoan [115] is a framework that allows users to trust that code running inside a TEE does not leak private data. *SCONE* [34] is a mechanism that uses SGX to protect Docker [184] containers. However, both *Ryoan* and *SCONE* do not address the need of service *owners* to upgrade their software while preserving the normal operation from the *clients'* perspective. Moreover, neither *Ryoan* nor *SCONE* has a mechanism to allow clients to track the service state and protect from rollback attacks.

3.9 Conclusion

In this chapter, I introduced TEE-SERVICE, a remote secure services management framework running on TEEs. TEE-SERVICE enables machines in a data center with no TEE support to use secure services remotely, while improving TEEs' security guarantees. TEE-SERVICE provides service *owners* with TEE-backed secure management of access permissions and enables deployment of software updates. TEE-SERVICE also allows service-*clients* to verify that they are communicating with the *servers* that the service *owner* intended, that the service state has not been tampered with or rolled back, and that the *server* software has not been modified.

Using TEE-SERVICE, I implemented a software security module, TEE-CRYPT, that provides cryptographic services while improving security guarantees of classical software solutions. I used TEE-CRYPT to improve the security of an authentication mechanism (PAM) in Linux. I also integrated TEE-CRYPT with the *NGINX* and *Lighttpd* web servers, demonstrating that TEE-CRYPT is a viable solution to protect the web servers' private keys. TEE-CRYPT outperforms commercial HSMs by a factor of $4\times$-$8\times$ for RSA and ECDSA operations, and $2.7\times$-$14.9\times$ for AES-GCM encryption, at a substantially lower cost (more than $5\times$).

CHAPTER IV

SovereignTEE: Enabling Large Scale Adoption of Trusted Execution in the Cloud

4.1 Introduction

In the previous chapter I proposed a framework for allowing service owners to manage configuration and ACLs of distributed services and end-users (service clients) to verify the services' state has not been reverted without their knowledge. In this chapter, I tackle two additional major challenges for practical deployment of trusted services: scaling attestation and migrating trusted services.

As a reminder, Fig. 4.1 illustrates the existing SGX ecosystem and the relationship between the service owners (the cloud customers), the cloud provider, and the end users in the cloud using SGX. Unfortunately, the existing SGX ecosystem poses two addtional substantial impediments to scalable adoption of trusted execution in cloud services.

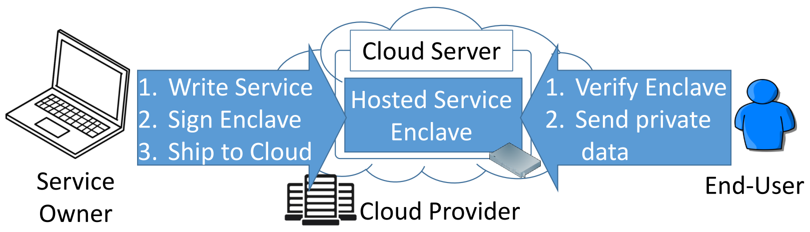


Figure 4.1: Hosting SGX-powered services in the Cloud. The Service owner signs the enclave implementing the service. The cloud provider chooses a machine to host the service enclave. The end-user remotely verifies the enclave authenticity before sending it private data.

First, SGX's attestation mechanism—the root of trust upon which SGX's security guarantees are based—requires active participation of the hardware vendor (Intel) *each time* an end user connects to a secure service. Service owners must be vetted by Intel to register their software signing key [130, 131] and end users must contact Intel's attestation servers *each time* they validate security guarantees [122, 139]. Since attestation is machine-specific, each user must repeat the attestation process whenever they connect to a different hardware device or whenever the service to which they connect migrates to a different machine. This mechanism interposes the hardware vendor into the relationship of the cloud provider, service owner, and end user, creating new challenges for scaling services. In other words, establishing trust (per user, per server, and sometimes per connection) depends on the availability of the hardware vendor's attestation servers, which neither the service owner nor the cloud provider control.

Second, SGX's mechanisms are tightly coupled to a specific piece of hardware. For example, SGX's secure storage mechanism [29] relies on sealing keys that can only be regenerated on the same physical machine. However, a fundamental tenet of virtualization in a cloud computing environment is that *hardware is fungible*. Cloud providers rely on mechanisms like live migration [46, 66, 275] to perform regular system maintenance and to optimize the packing of virtual machines onto available hardware [36, 165, 248]. Upon hardware failure, services can be rapidly restarted on a replacement system. Unfortunately, SGX's sealing and attestation mechanisms do not enable secure transfer of service state between machines—a serious impediment to typical cloud operations. Today, a service developer must roll their own application-specific solution to migrate data [41, 225], export files, distribute and share keys, or provide failure recovery [195, 257].

Recent work introduced frameworks to leverage SGX for secure execution [34, 41, 115, 240, 264]. Frameworks like SCONE [34] and Graphene [264] enable easy porting of existing applications to run in SGX enclaves. However, they neither address migration of enclaves across machines nor do they focus on providing availability and failure recovery.

VC3 [240] introduces a mechanism to prove to an end-user that an enclave indeed runs on a given cloud provider's machine, but it does not decouple the hardware vendor from the attestation process. As a result, the cloud provider lacks full control over service availability. Existing proposals [17, 41, 105, 218] for migrating secure enclaves or secrets assume that both the source and target machines are operational simultaneously, which can lead to significant over-provisioning of cloud resources and does not allow recovery of secrets if the source machine is no longer available.

No existing framework (1) decouples the hardware vendor from the attestation process required upon connection establishment, and (2) supports secure state transfer/sharing and seamless restart that are key to achieving failure recovery and high availability.

To address these two critical deployment and scalability challenges, in this chapter I introduce *SovereignTEE*: a framework that shifts the responsibility of managing trusted execution from the hardware vendor to the cloud provider. This paradigm of *autonomous attestation* removes the hardware vendor as a mandatory participant in each attestation operation *while preserving* SGX*'s security properties (i.e., confidentiality and integrity of data)*.

Decoupling the hardware vendor from the management of the trusted execution environment also enables the cloud provider to flexibly manage secure storage, thereby enabling failure recovery, high availability, and seamless data migration, without compromising integrity or confidentiality.

To allow the cloud provider to support high availability independently of Intel's infrastructure, *SovereignTEE* interposes itself as an intermediate attestation authority. *SovereignTEE* is attested by Intel and hence trusted by end-users. This setup decouples cloud users from Intel's attestation services [139] thereby, eliminating the dependency on the availability of the hardware vendor.

To enable failure recovery and secure migration of data between machines in the data center, *SovereignTEE* generates and transparently manages storage keys. Existing techniques for enclave migration [17, 218] rely on the source machine and enclave to be fully functional so they can actively participate in the data transfer. Furthermore, these methods assume that data is only accessed by a single enclave at a time. In contrast, *SovereignTEE* enables failure recovery even if the source machine cannot boot and allows multiple enclaves to access the same secure storage simultaneously. Using its attestation scheme, *SovereignTEE* allows securely sharing the storage secrets with other machines belonging to the same cloud provider (§4.3). The key derivation scheme ensures only enclaves written by the same owner can access the storage.

I demonstrate *SovereignTEE*'s ability to support high-availability (via autonomous attestation) and failure recovery (via secure data sharing) with five case studies of widely-used real-world systems, including two web servers, two databases, and a secure communication tunnel (§4.4). More specifically, I first show how end-users can perform remote attestation with the cloud service enclaves and verify their authenticity with the same trust level provided by the unmodified SGX attestation. Next, I demonstrate the ease of data sharing between instances running on different machines. I then show how *SovereignTEE* transparently enables failure recovery when the applications are restarted on a new machine, improving availability by an order of magnitude. I additionally show that *SovereignTEE* improves performance: up to 2–7% in Kops/sec and 3.5–6.5% improvement for read and write throughput (§4.4.2). Finally, I show that *SovereignTEE* requires minimal integration effort.

To summarize, in this chapter I make the following contributions:

- A new model for managing secure enclaves in a cloud infrastructure, which decouples the hardware and its vendor from the services running on it and thereby enables highly available services
- A secure key distribution mechanism that enables seamless data migration and data sharing between servers running in the same cloud infrastructure, thereby enabling efficient failure recovery

- An evaluation of real-world applications to demonstrate the benefits of *SovereignTEE*. *SovereignTEE* improves services performance by 2–7% and also improves availability by an order of magnitude

4.2 Background & Motivation

In this section, we describe the necessary background on Intel SGX that is central to the design of *SovereignTEE*. For better readability of this chapter, some of the details described in previous chapters are reiterated here. We point out the challenges and limitations of the current ecosystem and motivate the need for *SovereignTEE*.

SGX Basics. SGX [29, 140] enables a user process to create one or more trusted execution contexts, called *secure enclaves*, that are protected from other enclaves and privileged software, such as the Operating System (OS), hypervisor, and firmware (BIOS). At enclave initialization, code and data are copied into the enclave memory using the SGX API [136].

Code residing inside the enclave memory is considered *trusted*. Any other code, including the OS or the hypervisor code, is considered *untrusted*. The *trusted* and *untrusted* parts can request services from each other by invoking enclave calls (*ecalls*) and out calls (*ocalls*).

In SGX nomenclature, the enclave code and the enclave-writer's identity are represented by two cryptographic digests called MRENCLAVE and MRSIGNER [131, 136], which we further refer to as ENC-ID and SIG-ID, respectively. ENC-ID is the cryptographic digest (SHA-256) of the code and data copied to the enclave memory at creation time. SIG-ID is the cryptographic digest of the enclave creator's public key, thereby representing the identity of the enclave creator. We denote the ENC-ID and SIG-ID of an enclave *e* signed by owner *o* as ENC-ID_e and SIG-ID_o, respectively.

The enclave creator further uses their private key to sign the value ENC-ID to prove the authenticity of the enclave code and data to any 3rd party. The enclave's digest (ENC-ID), the enclave-writer's signature on ENC-ID, and his public key constitute the enclave header

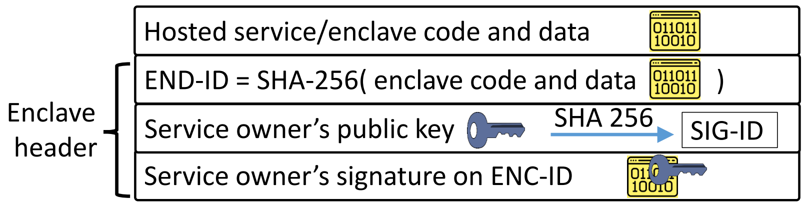

Hosted service/enclave code and data
END-ID = SHA-256(enclave code and data)
Service owner's public key
SHA 256
SIG-ID
Service owner's signature on ENC-ID
Enclave header

Figure 4.2: Enclave and its ENC-ID and SIG-ID. The service owner uses his public key, identified by SIG-ID, to sign ENC-ID. ENC-ID and SIG-ID are used as identifiers of the enclave for loading, attestation, and secure storage.

(Fig. 4.2).

The SGX SDK provides several *architectural* enclaves, signed by Intel, required for launching and attesting (*i.e,* validating the authenticity of) user enclaves. A user enclave can only be started after receiving a *launch token* from the architectural *Launch Enclave*. For a user enclave to prove to external entities that it is indeed an SGX enclave (and not an emulator), it must receive an attestation proof (called a *quote*) from the architectural *Quote Enclave* [69, 130, 131]

Local and Remote Attestation. The attestation process allows proving to a (local or remote) verifying party the integrity of (1) the enclave's code (ENC-ID), (2) the enclave creator's identity (SIG-ID), and (3) a key-share for establishing a shared secret between the parties.

The key-share serves an important role in the attestation process, as it ties the attestation to a specific secure session, based on the shared-secret that is established between the proving and the verifying parties. The key-share is typically a Diffie-Hellman Key Exchange handshake [202, 205], which proves to the remote verifier that there is no *man in the middle* intervening in the key-exchange handshake. *SovereignTEE* replaces the key-share in the attestation process with a *SovereignTEE*-specific public key, as we detail in §4.3.3.

SGX attestation can be performed *locally* with a neighboring enclave, running on the same processor, or with a *remote* verifying party, running on a separate machine.

Local Attestation. To implement local attestation, the SGX API provides a mechanism called *reporting* [29]. Fig. 4.3 depicts the reporting process. At a high level, during local

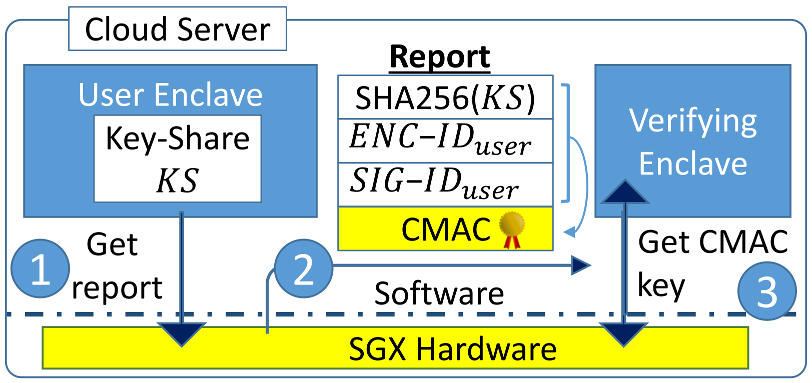


Figure 4.3: Local Attestation. (1) The user enclave asks the processor to generate a CMAC on its own ENC-ID, SIG-ID, and the digest (SHA-256) of the key-share, for a specific target enclave. (2) The CMAC'd report is sent to the verifying enclave. which (3) requests the CMAC key from the hardware, and then verifies the CMAC.

attestation, a user enclave performs attestation by proving to a local verifying enclave that it is running on genuine SGX hardware.

To initiate local attestation, the user enclave requests the processor to generate a *report*, containing the digest of the key-share[1] (step 1 in Fig. 4.3). The processor verifies that the request was executed within the user enclave and then generates a report containing the user enclave's identifiers, ENC-ID$_{user}$, and SIG-ID$_{user}$.

To target the report at a specific local verifying enclave, the processor computes a Cryptographic Message Authentication Code (CMAC) of the report (step 2), using a key that can only be derived using the verifying enclave's identifiers, ENC-ID$_{verify}$ and SIG-ID$_{verify}$. To verify the report's CMAC, the verifying enclave requests the processor to generate the CMAC key (step 3).

To ensure that the CMAC key is only accessible to the verifying enclave, the processor derives it from a report-key, known only to the processor, and the identifiers belonging to the enclave requesting the key: ENC-ID$_{verify}$ and SIG-ID$_{verify}$. Using the CMAC key, the verifying enclave verifies the report and thereby trusts the user enclave is running on the *same* processor as the verifier.

Remote Attestation. Enclaves perform *remote attestation* to prove to remote parties

[1]We simplified the reporting process for clarity.

(*e.g.,* end-users) the integrity of their code, the identity of the enclave creator (*e.g.,* the service owner), and the authenticity of the key-share. A user enclave starts *remote attestation* by performing *local attestation* with the *Quote Enclave* as the verifying enclave. That is, the user enclave proves to the *Quote Enclave* that it generated a specific key-share and that they are both running on the same machine.

The *Quote Enclave* has sole access to an attestation key [51, 122], which it uses to sign a blob containing the report. The process by which the *Quote Enclave* receives the attestation key from Intel is called *provisioning* [122] and depends on a secret fused into the CPU at manufacturing time [69, 136]. The signed blob is called a *quote*. The *quote* is returned to the user enclave, which in turn sends the *quote* to the remote party.

The remote party cannot verify the *quote* on its own and must contact Intel Attestation Service (IAS [139]) to verify the *quote*. IAS tracks all the attestation keys that have been provisioned to Intel's processors and can verify that the *quote* was signed by a genuine *Quote Enclave*. IAS's response is further signed by Intel's private key to indicate Intel as the attesting entity. The signed response effectively serves as a *transcript* that can be used at any time by any party (*e.g.,* a cloud end-user) to verify that the user enclave is a genuine SGX enclave that generated the specific key-share.

Adoption Challenge 1. IAS must approve any remote attestation *quote*, making IAS a potential scalability bottleneck and an impediment to high availability. In a typical enclave use case, remote end-users establish trust with the service enclave by performing *remote attestation* when they communicate for the first time, establishing a trusted shared-secret via the key-exchange. The produced *quote* is only valid for the specific machine the enclave runs on and for the current end-user performing the key-exchange handshake.

As a result, IAS is a bottleneck for enabling services' availability: IAS needs to be contacted once per enclave per machine per remote end-user. Verifying the *quote* with IAS is the *only* guarantee that an end-user has to ensure that they are communicating with a genuine SGX enclave and not an SGX simulator. Consequently, if IAS is not available due to a service outage or heavy load, end users cannot establish trust with the enclave.

***SovereignTEE* Goal 1** is to enable high availability of services by decoupling the *cloud provider's* infrastructure availability from *Intel's* infrastructure. *SovereignTEE* achieves this goal by providing a trustworthy attestation process without depending on Intel's individual approval of service owners' software for each new service, per machine, per end-user. Additionally, as we explain in §4.3.3, *SovereignTEE* extends the attestation functionality to not only verify the service owner's identity but to also vouch that the service enclave has no known vulnerabilities at the time of attestation.

Long Term Secure Storage. For secure persistent storage, enclaves encrypt files using a deterministic encryption key generated with the SGX *sealing* API [29, 131]. The sealing key depends on multiple components: a 32-byte KEY-ID, the enclave's SIG-ID, a secret fused into the CPU at manufacturing time, and (optionally) the enclave's ENC-ID. The generated sealing key can then be used by the enclave to encrypt file contents before they are saved to persistent storage. As a result, the OS, which is untrusted, cannot decrypt file contents.

KEY-ID is not considered a secret and can be saved in plain text alongside the encrypted file contents. KEY-ID's purpose is to ensure that a fresh sealing key is derived every time, so that encrypting the same file with different KEY-IDs will yield different cipher texts.

Adoption Challenge 2. Storage migration in Intel SGX cannot be managed automatically by the cloud provider, because the sealing key depends on a secret fused to a physical processor and cannot be regenerated on a different machine.

Typically, applications running in the cloud can migrate from one physical machine to another to allow failure recovery via file system crash consistency. Moreover, cloud management software may choose to migrate a secure application (and the corresponding enclave) depending on resource usage or maintenance status.

Alas, in the current SGX ecosystem, once the migration is complete, the application will not be able to access the files to which it previously had access, because the sealing key cannot be regenerated on the new physical machine.

A related challenge arises when several instances of an application, running on different machines, try to access data that resides in shared storage, or when an application receives input from another remote application. Such common scenarios cannot be supported without extending the enclaves to coordinate storage keys.

Service operators might handle machine failures or enable flexible migration by having enclaves proactively deliver the storage keys among themselves [17, 41, 105, 218, 225] or managing key distribution on external dedicated hardware [203]. However, both approaches prevent automatic resource management by the cloud provider. Proactive key delivery requires knowledge of the target machines prior to a failure, which in turn mandates pre-allocating machines to serve as backup. Unfortunately, most cloud providers do not provide any control over physical resource provisioning [54] and static machine allocation is inefficient [148, 227]. Therefore, machine pre-allocation will force the providers to shift their resource management policy and will likely affect overall costs for customers. Key management by service owners requires additional infrastructure on the owner's part and prevents the cloud provider from automatically restarting a service enclave on a new machine, which may be required due to unanticipated failures or maintenance events.

***SovereignTEE* Goal 2** is to enable scalable secure storage while (*a*) enabling the cloud provider to automatically manage its resources and migrate service enclaves for failure recovery or maintenance purposes, (*b*) allowing services to share data over common storage, and (*c*) preventing the service owner from having to dedicate additional in-house infrastructure ahead of time.

4.3 *SovereignTEE* Design

SovereignTEE's overarching goal is to increase services' availability by eliminating the dependency between the service software and the hardware on which it runs. This goal is achieved by augmenting three crucial mechanisms of the SGX ecosystem: launching (§4.3.2), attestation (§4.3.3), and secure storage (§4.3.4). Fig. 4.4 compares the current

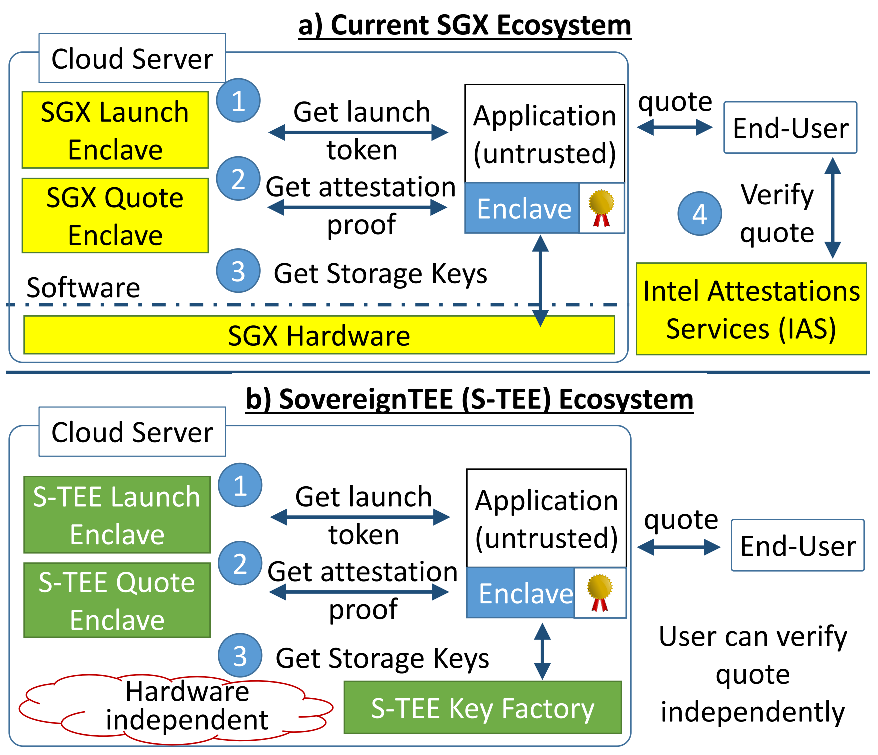


Figure 4.4: Current SGX Ecosystem vs. *SovereignTEE* Ecosystem. a) Enclaves depend on Intel provided services for launching (1), attestation (2,4), and for secure storage (3). b) *SovereignTEE* masks the Intel components for all steps in the life-cycle of enclaves.

enclave lifecycle in the SGX ecosystem to the lifecycle of an enclave in the *SovereignTEE* ecosystem. Each mechanism in *SovereignTEE* derives its trust from the original SGX mechanisms, thereby allowing the service owner and end-users to trust the replacement mechanisms.

In addition to eliminating the dependency on the hardware, replacing these mechanisms also allows *SovereignTEE* to augment them with new features to make SGX more flexible and practical in a cloud environment. Specifically, the launch and attestation mechanisms in *SovereignTEE* allow the cloud provider to launch and provide attestation proofs only to software versions that it white-lists, and deny attestation for (possibly vulnerable) deprecated versions (§4.3.3.3). The *SovereignTEE* Key-Factory allows applications to seamlessly migrate and share encrypted files across machines, without modifying the service software (§4.3.4).

We now explain the design of *SovereignTEE* and its mechanisms. Since *SovereignTEE* is SGX-specific, we also detail the relevant implementation details.

4.3.1 *SovereignTEE* Threat Model

Despite the fact that the cloud provider has full control over their servers and software stack, *SovereignTEE* provides the same security properties provided by SGX. Like previous work [34, 40, 115, 180, 240], we consider the cloud provider to be a powerful adversary that is capable of modifying the OS and the hypervisor and that has complete control over network traffic.

Similar to prior work [34, 115, 240], denial of Service (DoS) attacks are out of scope, as SGX only provides security and integrity guarantees. Cache timing, branch-shadowing, page-table, and synchronization attacks [53, 100, 164, 217, 243, 279, 286] are also considered out of scope. All SGX enclaves, including Intel's Quote and Launch enclaves, are vulnerable to these attacks [127, 130]. Using specific defense mechanisms [160, 180, 246, 250] is orthogonal to *SovereignTEE*'s design.

Rollback protection of persistent storage can be achieved by using the monotonic counters provided by the Intel SGX SDK Platform Software (PSW [131]). Relying on PSW counters, however, inherently hinders migration of services between machines. We therefore assume that rollback attacks on persistent storage are prevented using mechanisms which are independent of a specific machine such as proposed in *ROTE* [180] and LCM [47].

Lastly, we assume that the classical SGX attestation process is sound and resilient to speculative execution attacks [59, 155]. *SovereignTEE* components are implemented in SGX enclaves, so remote users can trust them using default SGX attestation (§4.3.3.2). Cloud providers should install OS and micro-code mitigations for SGX-specific attacks (*e.g.,* Foreshadow [269, 282]), which the clients can verify via remote attestation [121].

4.3.2 *SovereignTEE* Launch

SovereignTEE enables large-scale adoption of SGX by shifting the burden of vetting enclave-writers from the hardware vendor to the cloud provider by using the *SovereignTEE Launch Enclave* instead of Intel's architectural Launch Enclave. During machine setup, the cloud provider delivers their enclave signing key, SIG-ID$_{cloud}$, approved by Intel, to a *SovereignTEE* Launch Enclave. During launch time, the *SovereignTEE* Launch Enclave uses SIG-ID$_{cloud}$ to re-sign the service enclave code, as illustrated in Fig. 4.5.

Validation of Service Owners' Identity. *SovereignTEE* delegates responsibility for validating the enclave-writer's identity, SIG-ID$_{service}$, from Intel to the cloud provider. Intel's Launch Enclave permits execution in release-mode only to enclaves signed by a key matching an approved SIG-ID. This policy makes it harder for malware writers to run malicious software under SGX protection [243], as they must acquire or steal an approved key.

At a high level, the *SovereignTEE* Launch Enclave vets the service owner, instead of Intel (step 1, Fig. 4.5), verifies the enclave code is approved to launch (step 2, Fig. 4.5), and replaces the service owner's signature on the enclave code with the cloud provider's signature (step 3, Fig. 4.5). Since the cloud provider's signature is white-listed by Intel, the provider can effectively obtain the launch token on behalf of the cloud service.

Replacing the enclaves' signatures with SIG-ID$_{cloud}$ has side effects which affect remote attestation and secure storage. We further discuss these side effects and how *SovereignTEE* addresses them in §4.3.3 and §4.3.4.

Protecting the Cloud Provider's Key, SIG-ID$_{cloud}$. The integrity of the verification process and the confidentiality of the cloud provider's private key is protected by SGX's security properties. However, if the *SovereignTEE* Launch Enclave runs on the same machine as the service enclaves, the cryptographic implementation must be resilient to known side channel attacks mounted by malicious end-users (*e.g.,* controlled channel and branch shadowing [53, 164, 286]). Defense mechanisms have been investigated in prior work [61, 246, 250] and are orthogonal to the *SovereignTEE* design. Alternatively, the cloud

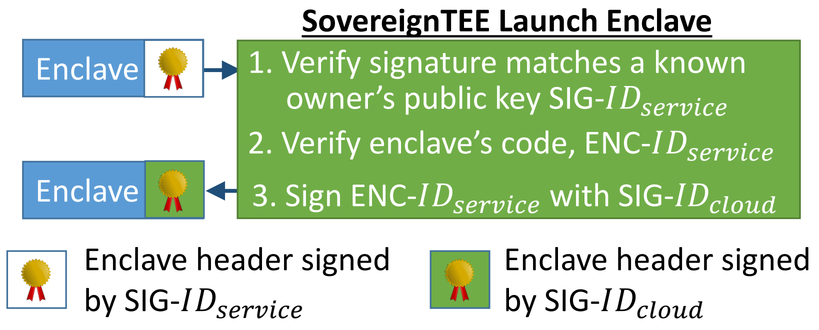

SovereignTEE Launch Enclave
Enclave
1. Verify signature matches a known owner's public key SIG-$ID_{service}$
2. Verify enclave's code, ENC-$ID_{service}$
3. Sign ENC-$ID_{service}$ with SIG-ID_{cloud}
Enclave
Enclave header signed by SIG-$ID_{service}$
Enclave header signed by SIG-ID_{cloud}

Figure 4.5: *SovereignTEE* Launch Enclave. (1) Verify the enclave owner's signature. (2) Verify that the service is allowed to run. (3) Re-sign the enclave's code with the cloud provider's Intel-vetted key, SIG-ID$_{cloud}$.

provider can better protect his private key by running the *SovereignTEE* Launch Enclave on a dedicated machine that is physically isolated from adversarial end-users, preventing side channel attacks such as Foreshadow [121, 269].

SovereignTEE's basic launch functionality (step 1 in Fig. 4.5) is similar to Intel's Launch Enclave: verifying that SIG-ID$_{service}$ belongs to a known service owner. In addition, the *SovereignTEE* Launch Enclave allows the cloud provider to also inspect the code of service enclaves.

Enabling Service Code Inspection. Shifting the enclave validation process from Intel to the cloud provider enables the provider to verify enclave contents, rather than just the enclave-writer's identity as the Intel Launch Enclave does. We conjecture that Intel chose to validate only the enclave-writer's identity because it would be costly (possibly infeasible) for Intel to keep track of all possible enclaves for all different services that are executing on all of Intel's processors.

However, the cloud provider has the ability, and more importantly the incentive, to know what software is running on its infrastructure. Similar to Apple's App Store [32] and Android's Google Play Protect [90, 91], the cloud provider can inspect the service enclave's code prior to its deployment. After approving the service, the cloud provider can store the service enclave's ENC-ID$_{service}$ in a repository of white-listed enclaves. The repository is then used by the *SovereignTEE* Launch Enclave prior to re-signing service enclaves.

During the launch process, the *SovereignTEE* Launch Enclave compares the ENC-ID$_{service}$ against a repository of approved enclaves (step 2 in Fig. 4.2). If a vulnerability is discovered in a specific service version, the cloud provider can delete the service's ENC-ID$_{service}$ from the repository of white-listed services and deny attestation to it, as we further discuss in §4.3.3.3.

4.3.3 *SovereignTEE* Attestation

The *SovereignTEE* attestation mechanism decouples services' availability from Intel's infrastructure by obviating the need to contact Intel to perform attestation, while maintaining the security guarantees provided by the classical SGX attestation. Although conceptually similar to the current SGX attestation process, *SovereignTEE*'s attestation replaces some key components with alternatives that are controlled by the cloud provider. *SovereignTEE* replaces the architectural Quote Enclave (QE) and the attestation key used by the QE, both of which are currently controlled by Intel, with a *SovereignTEE* QE and *SovereignTEE*-specific attestation key.

To ensure that the cloud provider cannot manipulate the attestation process (either intentionally or inadvertently, due to an ongoing attack on the cloud provider), *SovereignTEE* uses Intel's classical attestation process exactly once. Specifically, the SGX attestation is performed at machine setup time, to attest *SovereignTEE*'s replacement components.

In the next subsections, we explain how the cloud provider sets up a *SovereignTEE* Quote Enclave and its attestation keys (§4.3.3.1) and how trustworthy attestation proofs (*quotes*) are provided to end-users (§4.3.3.2). Finally, we discuss how *SovereignTEE* extends the classical attestation functionality to prevent the attestation of deprecated services (§4.3.3.3).

4.3.3.1 Machine Setup

To maintain the classical SGX attestation's security guarantees, the *SovereignTEE* Quote Enclave (QE) generates a machine-specific private key and proves the authenticity of the key and the QE's code (*i.e.,* its ENC-ID$_{QE}$) using the current Intel attestation process. To prevent the cloud provider from forging attestation, *SovereignTEE* secures the generated private key within the *SovereignTEE* Quote Enclave and never reveals it.

To allow external parties to verify that the *SovereignTEE* Quote Enclave (QE) does not leak the attestation key, cloud providers are expected to release the QE's source code. External parties can review the code, and verify that the QE is not revealing the key.

We now explain how the *SovereignTEE* Quote Enclave generates the attestation key, secures it, and how the key is bound to the specific enclave code represented by ENC-ID$_{QE}$, and to the cloud provider's identity.

Generating the Attestation Key. To provide attestation without requiring interaction with the Intel Attestation Service (IAS), *SovereignTEE* uses a classical asymmetric signature scheme (ECDSA [201]) instead of the EPID group signature scheme [51] that Intel uses. The EPID scheme allows the machine running the attestation to generate a signature without revealing its identity. However, this privacy property comes at the cost of requiring an additional party—IAS.

In a typical cloud scenario, there is no need to anonymize the cloud provider's machine. The cloud provider is either indifferent about having their machines identifiable by their public keys, or they can conceal any physical machine's identity by generating as many private-public key-pairs as needed.

Every *SovereignTEE* Quote Enclave generates local private and public attestation keys, denoted AK$_{priv}$, AK$_{pub}$, respectively. The *SovereignTEE* Quote Enclave uses AK$_{priv}$ to sign attestation quotes and end-users verify the quotes using AK$_{pub}$ (§4.3.3.2).

The private attestation key, AK$_{priv}$, is the source of trust in *SovereignTEE*'s attestation, and therefore must be protected from the cloud provider and the service enclaves running on the same machine as the *SovereignTEE* Quote Enclave. While in memory, the private key AK$_{priv}$ is secured within the *SovereignTEE* Quote Enclave thanks to SGX's secure memory properties. Any attack capable of extracting AK$_{priv}$ is, in principal, also possible on Intel's original Quote Enclave. When stored on persistent storage, AK_{priv} is protected using the classical SGX sealing API [29, 131].

However, a straight-forward use of the sealing API will expose the private attestation key to the service enclaves and the cloud provider. The default sealing key-derivation scheme, as implemented by the SGX SDK, uses the enclave-writer's identity (SIG-ID) to generate the sealing key. In the *SovereignTEE* ecosystem, all service enclaves are re-signed with the cloud provider's public key (§4.3.2). Therefore, they can all share the same SIG-ID_{cloud} and can thus acquire the same sealing key as the *SovereignTEE* Quote Enclave. Moreover, using this default sealing scheme also allows the cloud provider to write a second Quote Enclave, signed with the same SIG-ID_{cloud}, that can unseal and leak the attestation key.

To protect the attestation key from the cloud provider and other service enclaves, the *SovereignTEE* Quote Enclave (QE) stores the attestation key using a sealing key derived from its code's digest, ENC-ID_{QE}. Other enclaves, even if written by the cloud provider, will have a different ENC-ID and will not be able to acquire the same sealing key. Protecting the attestation key in this manner is similar to how SGX QE protects the attestation key, generated by Intel. Therefore, attacks on *SovereignTEE* QE are conceptually also possible on SGX QE.

For remote verifiers to trust the *SovereignTEE* attestation process, they must trust that the private attestation key, AK_{priv}, is only accessible to a trustworthy Quote Enclave, controlled by the cloud provider and protected by SGX security properties. We now explain how *SovereignTEE* binds the public attestation key, AK_{pub}, to a specific cloud provider and proves that the key was generated within an enclave matching ENC-ID_{QE}.

Binding AK_{pub} to the Cloud Provider. To prove to service owners and end-users that the *SovereignTEE* Quote Enclave runs on a machine owned by the cloud provider, the provider certifies AK_{pub} using the cloud provider's public key, as illustrated in Fig. 4.6a. Binding AK_{pub} to the cloud provider is important to ensure storage migrations is only possible within the cloud provider's machines (§4.3.4). Moreover, VC3 [240] discuss the importance of allowing end-users to verify they are interacting with a machine owned by the cloud provider in order to prevent cuckoo attacks [178]. In this attack scenario, the attacker redirects the end-user to communicate with a machine controlled by him, allowing the attacker to conduct physical attack to extract the SGX master secrets.

At setup time, the cloud provider has direct access to the machine and therefore trusts the generated public attestation key (AK_{pub}) belongs to a specific machine.

The cloud provider assures end-users and service owners that the attestation public key (AK_{pub}) belongs to the *SovereignTEE* Quote Enclave, running on a machine within the cloud infrastructure, by certifying AK_{pub} with the provider's public key. We assume that the cloud provider has a Certificate Authority (CA) key that allows it to generate a classical PKI certificate [114, 158] over AK_{pub}. The certificate is verified by end-users during operation (§4.3.3.2), when they verify the attestation *quote* containing the key-exchange key-share.

Binding AK_{pub} to ENC-ID$_{QE}$. The *SovereignTEE* Quote Enclave uses classical SGX attestation to prove that the public attestation key, AK_{pub}, was generated and is protected by an enclave with code matching ENC-ID$_{QE}$. This process is illustrated in Fig. 4.6b. At machine setup time, the *SovereignTEE* Quote Enclave performs a classical SGX attestation with a modified *quote*. Instead of containing a key-share, used to establish a secure session (§4.2 and Fig. 4.3), the *quote* contains AK_{pub}.

For end-users and service owners to verify the *quote* without contacting Intel themselves, the cloud provider contacts Intel Attestation Service (IAS [139]) on their behalf, at machine setup time. When queried with a *quote*, IAS provides a response signed by Intel's public key, serving as the attestation *transcript*. Therefore, the transcript can be verified *offline* by any third party that trusts Intel's public key. The attestation transcript proves that on a given time and date, a *SovereignTEE* Quote Enclave with code matching ENC-ID$_{QE}$, which generated the attestation key AK_{pub}, successfully performed the attestation process with IAS. If the cloud provider wishes to prevent the public attestation key from being a unique machine identifier, it can generate several attestation keys per machine.

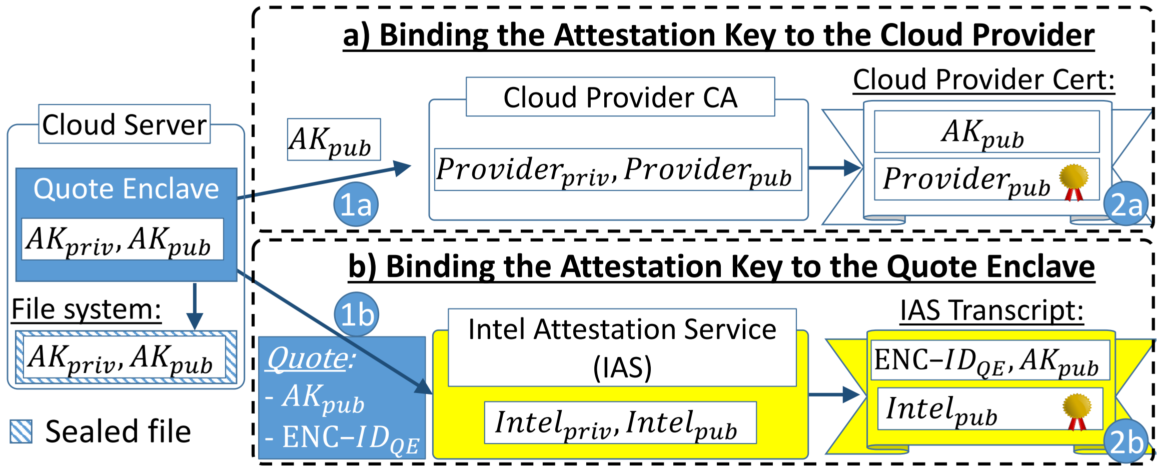


Figure 4.6: Attestation-Key Chain of Trust. (a) At setup time the cloud provider's Certificate Authority (CA) signs the public attestation key (2a), AK_{pub}, proving that the *SovereignTEE* Quote Enclave indeed runs on the provider's infrastructure. (b) IAS verifies the quote and attests that an enclave, running the code that matches ENC-ID_{QE} has generate a public key AK_{pub}. The generated IAS transcript (2b) proves that AK_{pub} was generated by a genuine SGX enclave. At rest, the Attestation Key is protected on the file system via the SGX sealing API.

One-time SGX Attestation. In *SovereignTEE*, the classical SGX attestation is performed only once per machine, instead of once per service enclave per machine per end-user. When end-users attest service enclaves, *SovereignTEE* attestation (§4.3.3.2) is performed instead of classical SGX attestation and there is no need to further interact with Intel.

SovereignTEE repeats the classical SGX attestation in case of a microcode or critical SDK update. Intel may publish microcode or SGX SDK critical security updates [125, 135]. The cloud provider can decide to update the microcode or the SDK version, and then re-execute the attestation process with IAS, receiving a new signed transcript with a fresh time-stamp.

4.3.3.2 Operation time

During operation time, remote end-users connect to the service enclave and request an attestation proof (*quote*) to verify the service is running on a trusted enclave. In the *SovereignTEE* ecosystem, the service enclave uses the same reporting scheme as in classical SGX attestation, however this time, with the *SovereignTEE* Quote Enclave. After verifying the service enclave, *SovereignTEE*'s Quote Enclave generates a *SovereignTEE quote*, signed by the attestation key, AK_{priv}, generated at machine setup (§4.3.3.1).

Verifying the Service Enclave. The *SovereignTEE* Quote Enclave verifies the service enclave code, $\text{ENC-ID}_{service}$, via the SGX reporting mechanism [131, 136], also known as *local attestation* (§4.2). However, since *SovereignTEE* re-signs the enclave's header with SIG-ID_{cloud} at launch time (§4.3.2), the report does not contain $\text{SIG-ID}_{service}$. That is, the report is not enough to convince the *SovereignTEE* Quote Enclave that the service belongs to a specific service owner, with a public key matching $\text{SIG-ID}_{service}$.

For the *SovereignTEE* Quote Enclave to verify the service enclave was signed by $\text{SIG-ID}_{service}$, the service enclave also sends its original enclave header (Fig. 4.2), containing the service owner's signature on $\text{ENC-ID}_{service}$.

The *SovereignTEE* Quote Enclave then inspects the RSA signature [201] in the enclave header and verifies that it was signed by $\text{SIG-ID}_{service}$. The *SovereignTEE* Quote enclave generates the attestation proof (that is, the *quote*) with the correct $\text{SIG-ID}_{service}$. Finally, the quote is signed with the *SovereignTEE* Quote Enclave private key, AK_{priv}, generated at setup time (§4.3.3.1).

Verifying a *SovereignTEE* Quote. End-users can verify *SovereignTEE*'s attestation proof without contacting Intel. In the attestation process, the *SovereignTEE* Quote Enclave provides the service enclave with (*a*) the *quote* and (*b*) the cloud provider's certificate on the public attestation key, AK_{pub} (2a in Fig 4.6a), and the Intel signed *transcript* (2b in Fig. 4.6b), which bind AK_{pub} to the the cloud provider and *SovereignTEE* Quote Enclave.

The *SovereignTEE quote*, the certificate on AK_{pub}, and the IAS *transcript* are then sent to the end-user for verification. The end-user verifies the attestation by (*a*) inspecting the certificate chain in the *quote*, thus trusting that AK_{pub} was generated on a machine owned by the cloud provider; (*b*) inspecting the IAS *transcript*, thus trusting the attestation-key, AK_{pub}, was generated on a genuine *SovereignTEE* Quote Enclave secured by SGX; and (*c*) inspecting the key-share, hence the end-user trusts it is establishing a secure channel with a genuine service enclave. For (*a*), we assume the end-user trusts the cloud provider's CA key. For (*b*), we assume that the end-user trusts Intel's public key.

4.3.3.3 White-listed Enclave Versions

Using *SovereignTEE* to perform attestation does not allow IAS to revoke service owners' identities on one hand, but allows extending the attestation functionality on the other. Because *SovereignTEE* interacts with the IAS once at setup time, IAS cannot subsequently decline attestation to specific service owners or specific enclaves.

To our knowledge, Intel does not maintain an enclave version tracking system to revoke enclaves with known vulnerabilities. Maintaining such a repository for all enclaves on all machines would entail significant cost.

However, declining attestation to enclaves with known vulnerabilities is crucial for services' and end-users' security [226]. Over time, vulnerabilities are expected to be found in existing enclaves.

SovereignTEE enables the cloud provider to manage the revocation of vulnerable enclave versions. During attestation (§4.3.2), the *SovereignTEE* Quote Enclave verifies that the attested enclave's $\text{ENC-ID}_{service}$ is approved to receive attestation quotes, prior to signing the quote. When a vulnerability is discovered in a specific service enclave, the cloud provider removes its $\text{ENC-ID}_{service}$ from the repository of trusted services. When the service owner releases a service enclave that fixes the vulnerability, it will have a new $\text{ENC-ID}_{service'}$.

4.3.4 Storage-Key Management

SovereignTEE enables efficient failure recovery and allows services to share storage and seamlessly migrate data between machines by generating a storage key, using a Key-Factory enclave. *SovereignTEE* seamless migration can be used in conjunction with existing SGX frameworks such as SCONE and Graphene [34, 264] which do not support machine migration. At service setup time (§4.3.4.1), the Key-Factory enclave receives a storage seed from the service owner, which serves as the base entropy for storage keys. The service owner's storage seed is secured in memory and on the local file-system via SGX security guarantees. During operation time (§4.3.4.2), the service requests a key from the Key-Factory instead of using the SGX sealing API. Instances of Key-Factories securely share storage seeds to allow seamless migration of encrypted files between machines.

4.3.4.1 Setup Time

To ensure the storage keys are only known to the service owner, the owner securely delivers a randomly generated storage-seed to the Key-Factory enclave. The storage-seed is used during operation time (§4.3.4.2) to generate storage keys via a Key Derivation Function [62].

The service owner uses the *SovereignTEE* attestation process (§4.3.3) to (1) verify the Key-Factory enclave's integrity, (2) ensure that the Key-Factory enclave is running on the cloud provider's infrastructure, and (3) to establish a secure communication channel with the Key-Factory. The service owner then uses the secure channel to deliver their storage-seed. To allow future key requests from service enclaves belonging to the service owner, the Key-Factory registers the storage seed as belonging to the service owner's identity, namely $\text{SIG-ID}_{service}$.

The Key-Factory protects the storage-seed via the SGX sealing API. Similar to the sealing policy used by the *SovereignTEE* Quote Enclave to protect the private attestation key (§4.3.3), the Key-Factory (KF) derives its sealing key from its ENC-ID_{KF}.

To provide service enclaves with storage keys on new machines, which are yet to receive the storage seed, every Key-Factory is configured with the address of a remote, backup Key-Factory. For simplicity, in the following discussion, we assume that all machines in the cloud infrastructure have a single backup remote Key-Factory, as depicted in Fig. 4.7. All Key-Factories use the backup Key-Factory to back up and retrieve storage-seeds.

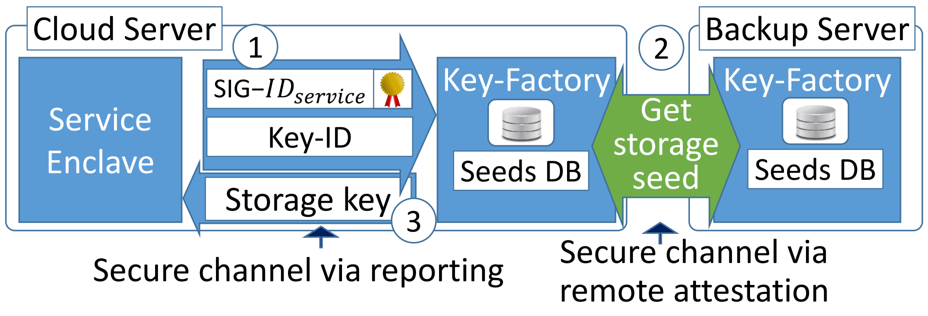


Figure 4.7: *SovereignTEE* Key Factory. (1) The enclave sends its original header containing its signature by SIG-ID$_{service}$, and a KEY-ID. (2) If needed a secure channel is established with a remote backup Key-Factory to retrieve the storage-seed. (3) Finally, the local Key-Factory derives and returns the key.

4.3.4.2 Operation Time

To allow easy integration of *SovereignTEE* into existing services, *SovereignTEE* replaces all file operation APIs (*e.g.,* fopen, fread, fseek, etc.) with wrapper code to use the *SovereignTEE* secure storage mechanism. *SovereignTEE* automatically performs the key management operations, described below, on behalf of the service enclave.

To provide similar security properties as SGX sealing, *SovereignTEE* storage keys are derived from the enclave-writer's identity (SIG-ID), arbitrary 32 bytes of nonce called KEY-ID, and optionally the enclave's code digest (ENC-ID). While SGX sealing keys are derived from a machine-specific secret, fused to the processor at manufacturing time, *SovereignTEE* storage keys are derived from the storage-seed provided by the service owner.

Service enclaves securely retrieve the storage key over a secure channel with the Key-Factory, established via *local attestation* (§4.2). Similar to the *SovereignTEE* attestation process (§4.3.3.2), the service enclave proves its SIG-ID$_{service}$ by sending its enclave header (Fig. 4.2) with the signature signed by SIG-ID$_{service}$ (step 1, Fig. 4.7).

The *SovereignTEE* secure storage mechanism allows enclaves to seamlessly reuse files created on other machines. When a service enclave requests a storage key, the Key-Factory will search its repository for a storage-seed matching the SIG-ID$_{service}$ of the requesting enclave. The absence of a match means that the service enclave has migrated to this machine and that the storage-seed needs to be retrieved. The local Key-Factory will contact the remote backup Key-Factory enclave to securely retrieve the storage-seed by using *SovereignTEE* remote attestation, as depicted in Fig. 4.7.

SovereignTEE remote attestation enables the backup Key-Factory to verify the storage seed is delivered to a trustworthy Key-Factory. That is, a Key-Factory (KF) with the same ENC-ID_{KF} and that runs on a machine within the cloud provider's infrastructure.

4.4 Evaluation

In this section, we evaluate *SovereignTEE*'s ability to provide high availability and failure recovery by answering the following questions: (1) Can *SovereignTEE* enable machines to share data via common secure storage and enable them to be securely migrated/restarted upon failure to improve availability (§4.4.1)? (2) What is the performance impact of using *SovereignTEE*'s secure storage capability compared to the current SGX storage mechanism (§4.4.2)? (3) What is the integration effort required for existing applications to leverage *SovereignTEE* (§4.4.3)?

We evaluate *SovereignTEE* on popular applications with realistic use cases in a cloud computing scenario. We focus on applications that are used to manage private information belonging to service owners and end-users. Specifically, we evaluate *SovereignTEE* on two databases: SQLite [255] and Redis [232]; two web servers: NGINX [200] and Lighttpd [167]; and a secure communication application: OpenVPN [213]. SQLite is an embedded database used in Firefox, iOS, Chrome, and Android; Redis is an in-memory key value store which backs up its database to disk and is used by Twitter, GitHub, Craigslist, StackOverflow and more; the NGINX web server serves around 24% of the web [197]; OpenVPN provides confidential and authenticated secure channels between remote machines, and is supported by Amazon's AWS [24] and Microsoft Azure [186].

Lastly, to compare file access performance of *SovereignTEE* secure storage vs. using SGX sealing keys we evaluate *SovereignTEE* on the Flexible File System Benchmark (FFSB [78]).

Experimental Setup: To test file accesses from multiple machines, we conduct the experiments using two machines. The first machine is a 2.8GHz Intel Core i7-7700HQ with 8 logical cores and 32 GB DDR4 RAM. The second machine is an Intel Core i7-7567U CPU @ 3.50GHz with 4 logical cores and 32 GB DDR4 RAM. Both machines run Ubuntu 16.04. The machines communicate over a 1 Gb/sec link (948 Mbits/sec, effectively, as measured using iperf3 [144]). Average link latency is 0.69 *ms*, as measured using ping.

The *SovereignTEE* implementation consists of 12923 C/C++ lines of code (LOC), of which 8256 LOC belong to the launch utilities (§4.3.2); 3103 LOC to attestation utilities (§4.3.3); 1564 LOC to secure storage (§4.3.4).

4.4.1 *SovereignTEE*'s Effectiveness

In this section, we evaluate the effectiveness of *SovereignTEE* by demonstrating how we integrate it with five different applications. We first explain how applications use *SovereignTEE* during their life-cycle: launch, attestation, and secure storage access. We then show that *SovereignTEE* allows services to seamlessly migrate and be restarted on a different machine, using the same configuration and database files. We also show that services can share common storage without interacting with each other to agree on a storage key.

Launching Enclaves: Launching enclaves with *SovereignTEE* does not require any binary linking (unlike attestation and secure storage). We use the *SovereignTEE* Launch Enclave to re-sign the five tested applications (Fig. 4.5). To verify the correctness of the re-signing process, we perform the classical SGX attestation with the re-signed applications. We then inspect both the attestation quote and the respective IAS *transcript* (2b in Fig. 4.6b) and verify that it contains SIG-ID$_{cloud}$ instead of the original SIG-ID$_{service}$.

We now explain the modifications that existing applications require to leverage *Sovereign-TEE*.

***SovereignTEE* Attestation:** We verified that the IAS *transcript*, generated for

SovereignTEE Quote Enclave (QE) at setup time (Fig. 4.6b), attests that the QE runs on a genuine SGX enclave, and that the QE's public key, AK_{pub}, that appears in the IAS *transcript*, is used to sign the attestation *quotes* (§4.3.3.2).

Secure Storage: For secure persistent storage, enclaves can use the SDK *protected file system* API. This API can be configured to either use SGX sealing keys or manual keys, provided by the enclave code. *SovereignTEE* uses the SDK *protected file system* API in manual mode and supplies the key retrieved from the *Key Factory* (§4.3.4). Any further operations using the SDK *protected file system* API are then seamlessly supported.

4.4.1.1 Failure Recovery: Enabling Database Migration

To evaluate *SovereignTEE*'s usefulness for migrating data between machines, we test Redis [232], and SQLite [255]. For both applications, we run a specific performance evaluation benchmark (detailed below) on one machine, where each benchmark generates a database. We then run the benchmark again on the existing database on another machine and ensure that the records in the database can be read from and written to.

SQLite: We evaluate SQLite using *lsmtest* benchmark [254]. The *lsmtest* benchmark adds new key-value pairs to the database and then reads them back. We use the default key size of 12 bytes and value size of 100 bytes, with 1:1 read-write ratio. SQLite accesses a total of three files: the database file and two temporary files, which are deleted upon process exit. At the end of the benchmark, the database contains 2.5 million records. We then move the database file to another machine and re-run the benchmark with the existing database. We run the benchmark in read-write mode to ensure both reading and writing is possible on the new machine.

Redis: Redis is in-memory key-value store [232], which backs up its database to the file system. Restarting Redis after a crash or a shutdown reloads the database from the file system. We test Redis with Redis's *memtier benchmark* [231].We run the benchmark with 1:1 read-write ratio until it generates a database with 1.5 million keys. We then copy the

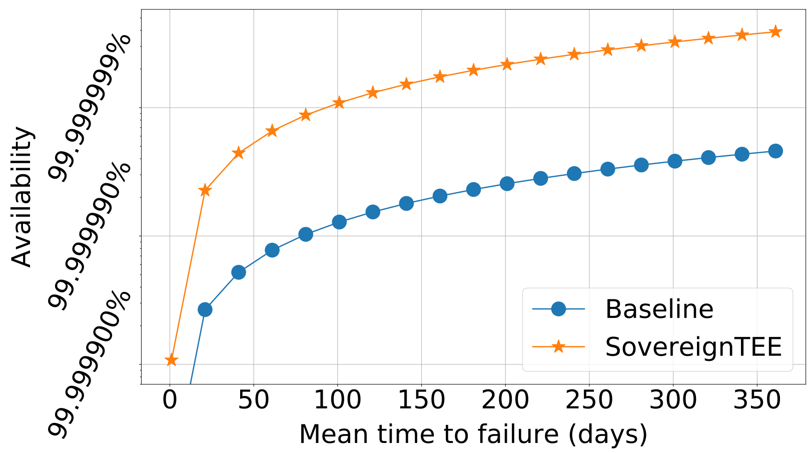


Figure 4.8: Availability of Redis server. Without *SovereignTEE*, the machine owning the data must be restarted upon failure. In contrast, using *SovereignTEE*, the database can be transferred and reloaded on a different machine, without waiting for the original server to recover, which improves availability by an order of magnitude on average.

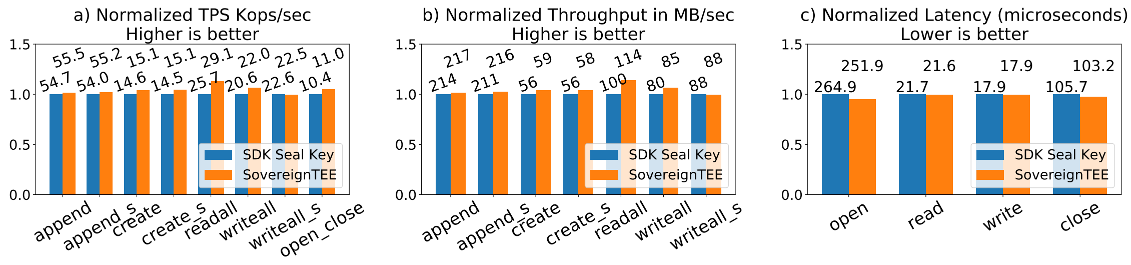


Figure 4.9: FFSB Performance Results. Normalized performance of using SGX *sealing* key managed via the SDK vs. a storage key managed by *SovereignTEE*. All performance values are normalized to the native SDK performance; Numbers are actual values. a) Transactions Per Second (TPS) of different microbenchmarks in FFSB [78]. *_s* is short for *_fsync*. b) Bandwidth in MB/Sec. c) Average latency of basic operations.

encrypted database file to the new machine and restart Redis. Redis successfully loads the database back into memory and it is fully functional; *i.e.,* stored values can be fetched and existing keys can be updated with new data.

Figure 4.8 shows how seamless data migration can improve the availability of a Redis server. We used *memtier* to generate a 3.5 GB data-set (in memory), which is compressed to 96 MB on disk. We compute availability as the ratio of time-to-recover and mean-time-to-failure. Time to recover includes either rebooting the machine (baseline, 60 s[2]) or

[2]The experiments with two server-class Intel Xeons reveled that a reboot always takes longer than 60 seconds. We pick 60 seconds as the baseline to err on the side of caution.

transferring the database using *SovereignTEE* (0.1 s over a 10 Gb link), and then reloading Redis (7.9 s). *SovereignTEE* improves the availability of Redis by an order of magnitude.

4.4.1.2 Enabling Data Sharing Between Machines

We now evaluate the ability of *SovereignTEE* to enable scaling of services by supporting access to encrypted files across two separate machines. Without *SovereignTEE*, each individual program would need to implement a mechanism for sharing their storage key. We evaluate file sharing with Lighttpd [167], NGINX [200], and OpenVPN [213].

Typical web server deployments have many instances on separate machines concurrently serving web pages from common storage. The tests serve pages from a single common storage simultaneously accessed by web servers on two machines. We verify Lighttpd and Nginx are serving web-pages correctly using *http_load* [7]. The benchmark fetched concurrently 1000 different pages with varying sizes between 10 KB to 10 MB. The web servers operated correctly, fetching pages from the common encrypted storage.

For OpenVPN, multiple service instances may need to communicate with each other in a public cloud infrastructure. Whereas the software instances run on separate machines, it is desirable for them to retrieve a common encrypted configuration and encryption keys. To evaluate OpenVPN with *SovereignTEE*, we create a tunnel between two machines using configuration and encryption keys on common storage. We then test the encrypted tunnel link using *iperf3* [144]. Both machines were able to send and receive data over the encrypted tunnel.

4.4.2 *SovereignTEE*'s Efficiency

In this section, we evaluate the performance impact of managing storage keys with *SovereignTEE* and show that it has no adverse effects, compared to the baseline of using SGX sealing keys.

If the key is managed automatically, the SGX SDK derives it from the machine-specific

SGX *sealing* key [29, 131], which means the file cannot be used on any other machine. If the key is managed manually by the software, the SGX software has to receive the key from an external party any time the service is restarted on a different machine.

We evaluate the performance of different types of file accesses using the Flexible File System Benchmark [78]. The benchmark evaluates the latency of basic operations such as open, close, read, write, and measures the throughput of reading and writing to new and existing files using various POSIX API functions.

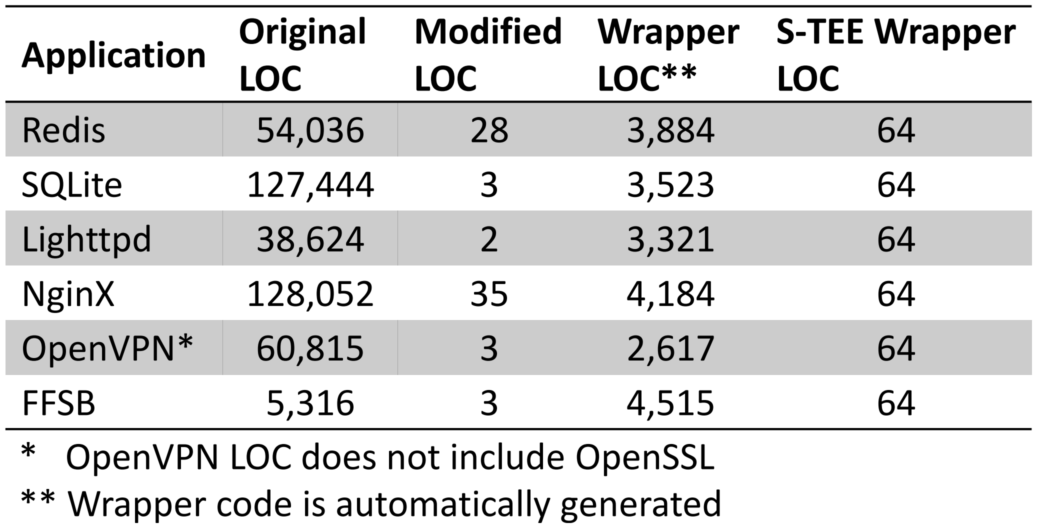
In all the experiments, we compare the performance of accessing files with an SDK-managed sealing key vs. file accesses using *SovereignTEE* to manage the storage keys. As shown in Fig. 4.9, managing storage keys with *SovereignTEE* is at least as efficient as using *sealing* keys that are automatically generated by the SDK. Fig. 4.9a shows that *SovereignTEE* enables similar or higher rate of Transactions Per Second (by 2-7%) for various file operations. Fig. 4.9b shows that the data bandwidth associated with the various file operations is either comparable or higher (by 3.5–6.5%) when using *SovereignTEE*. Finally, Fig. 4.9c demonstrates that the latency associated with basic file operations is the same or lower (down to 4.5% less) when using *SovereignTEE*.

The automatic key management performed by the SGX SDK is slightly slower because the SDK needs to call `EGETKEY` to retrieve the *sealing* key. In contrast, a *SovereignTEE* storage key is retrieved only once, when the service enclave is loaded.

Memory overhead of *SovereignTEE*: *SovereignTEE* attestation and secure storage requires linking with additional functionality provided by *SovereignTEE*. *SovereignTEE* static library adds 792 KB of code to the enclave. The SGX SDK static libraries add 683 KB, for a total overhead of 1475 KB in the resulting service-enclave object size.

4.4.3 Integration Effort

SovereignTEE is intended to work with SGX applications. However, the applications we test were not originally designed to use SGX. We therefore port each application to run



Application	Original LOC	Modified LOC	Wrapper LOC**	S-TEE Wrapper LOC
Redis	54,036	28	3,884	64
SQLite	127,444	3	3,523	64
Lighttpd	38,624	2	3,321	64
NginX	128,052	35	4,184	64
OpenVPN*	60,815	3	2,617	64
FFSB	5,316	3	4,515	64

* OpenVPN LOC does not include OpenSSL

** Wrapper code is automatically generated

Figure 4.10: Effort of porting applications to run in SGX Enclaves, in Lines of Code (LOC). SGX enclaves can not access the OS API directly and therefore wrapper code is added to perform out-calls (*ocalls*) to regular code. The automatically generated wrapper code is then modified to use *SovereignTEE* secure storage operations requireing 64 LOC changes in each case.

inside an SGX enclave. The SGX porting process requires linking applications with the SGX SDK replacements for standard libraries. Any POSIX APIs that require interaction with the OS (*e.g.,* for network accesses) are replaced with wrapper code to interact with non-enclave code via *ocalls* (§4.2). We generate most of the wrapper code using an automatic tool we built.

Fig. 4.10 depicts the porting effort in lines of code (LOC). Applications require modifications to run properly within SGX enclaves. Porting an application to run inside an SGX enclave is orthogonal to the effort required to integrate the application with *SovereignTEE* and requires on average 2,600–4,300 LOC. Integrating an SGX-ready application with *SovereignTEE*, however, requires a relatively small effort (64 LOC) and is fully automated.

4.5 Related Work

Recent work proposes useful frameworks for porting existing applications to run under SGX protection [34, 40, 264] and preventing data leakage [115]. However, these works do not address the key practical deployment challenges of Intel SGX such as continuous dependence on the hardware vendor and inflexible deployment mechanisms. Moreover, these works also do not provide failure recovery features when their enclave hosting machine has crashed.

LibSEAL, Pesos, EnclaveDB, and Attestation Transparency (AT) [39, 41, 157, 225] suggest SGX-based mechanisms to augment trust in various cloud storage services and improve the security of trusted communication with them.

The *SovereignTEE* Quote-Enclave functionality of monitoring enclave versions (§4.3.3.3) can benefit from the elaborate database proposed by AT. Pesos and EnclaveDB [157, 225] are mechanisms for providing trusted storage. Pesos delegates secure storage to external secure hardware, with which it communicates over a secure TLS connection. *SovereignTEE* does not assume any security guarantees are provided by the storage hardware. EnclaveDB does not rely on secure external storage, however, it cannot recover the contents it manages if the machine it runs on crashes. While AT implements manual key management, its protocol requires interactions between two functional AT machines and cannot recover the key if a machine crashes.

The enclaves implementing *SovereignTEE* should be cognizant of existing attacks on SGX enclaves, such as controlled channels, cache side channels, branch shadowing, etc. [53, 280, 100, 162, 164, 286]. Future implementations of *SovereignTEE* can benefit from employing known defense mechanisms, such as SGX-Shield, T-SGX, SGXBOUNDS, etc. [61, 160, 246, 250].

4.6 Conclusion

Trusted Execution Environments (TEEs) promise to revolutionize cloud security. However, the only widely available TEE for servers, Intel SGX, presents challenges for supporting high availability and failure recovery for deployed services. In this chapter, I analyze these challenges and introduce *SovereignTEE*—a new model for managing secure enclaves in a cloud infrastructure. *SovereignTEE* enables cloud providers to be independent of Intel's infrastructure availability and securely manage the attestation process. *SovereignTEE* enables scaling of services by allowing cloud seamless migration data between machines in the data center. Finally, I demonstrate that *SovereignTEE*'s effectiveness and efficiency on five real-world programs, showing it improves throughput by 2–7% and increases services availability by an order of magnitude.

CHAPTER V

Foreshadow: Breaking SGX using Speculative Execution

5.1 Introduction

In previous chapters I analyzed SGX impact of applications' performance and proposed frameworks to enable practical deployment of distributed trusted services. In this chapter, I analyze the SGX's resilience to micro-architectural attacks. Specifically, I present a micro-architectural side channel attack that breaks SGX's confidentiality, integrity and trust. A different variant of the vulnerability was independently and concurrently discovered by a team at KU Leuven. Intel's subsequent investigation of the attack uncovered two closely related variants, which Intel refers to as L1 Terminal Fault (L1TF [282]).

Notwithstanding its strong security guarantees, SGX does not protect against micro-architectural side channel attacks. Such side channel attacks exploit subtle timing variations resulting from contention on CPU micro-architectural resources to extract otherwise-unavailable secret information. Since their introduction over a decade ago [221, 42, 217, 265], micro-architectural attacks have been used to break numerous cryptographic implementations [293, 87, 145], track user behaviors [215, 171, 104], and create covert channels [103, 277]. Moreover, recent works combine micro-architectural attacks with speculative execution [155, 124, 172], allowing the attacker to read the entire address space of victim processes or of the operating system.

In terms of protection against side channel attacks, Intel acknowledges that "SGX does not defend against this adversary" [129, Page 115] arguing that "preventing side channel attacks is a matter for the enclave developer" [127]. Indeed, starting with the controlled channel attack [286], numerous works have demonstrated side channel attacks on or from SGX enclaves (See Section 5.1.4). Crucially, all the previously published attacks on SGX exploit *existing* side channel vulnerabilities [286], coding bugs [160], or speculative execution gadgets [208, 60] in enclaves' code to leak sensitive data.

Much less is known, however, about the side channel security of SGX enclaves that do not contain existing side channel vulnerabilities or other coding bugs. Thus, in this chapter, I ask:

Can an adversary extract secret data from an enclave's address space when the code running in that enclave does not itself have any security vulnerabilities? If so, can this be done cheaply and unobtrusively?

Next, I observe that whereas SGX's confidentiality guarantees in the presence of side channels have been studied before [286, 271, 163, 279], SGX's integrity guarantees in the presence side channels have received much less research attention. Thus, I ask:

What are the implications of side channel attacks on the SGX integrity guarantees? Can an adversary make an enclave operate on corrupted input data or corrupted state?

Finally, given the importance of SGX *remote attestation* [149] in establishing trust in the SGX ecosystem, I finally ask:

Can a side channel adversary erode the trust in SGX remote attestation? If so, what will it take to mount such an attack?

5.1.1 My Contribution

I answer all three questions in the affirmative. I answer the first question by presenting several new attacks that compromise SGX's confidentiality guarantees. I then use the attacks on SGX's confidentiality properties to break SGX's integrity guarantees, thereby answering the second question. Finally, I use these attacks to recover the machine's private attestation keys, thereby breaking SGX's attestation protocol and answering the third question. *As such, without mitigation*[1]*, the attacks described in this chapter fully compromise the basis of trust in the SGX ecosystem, both in terms of confidentiality and integrity.*

Breaking SGX's Confidentiality The first attack exploits the speculative execution features present in all SGX-enabled Intel CPUs to read the entire address space of victim enclaves. Crucially, unlike previous Spectre-style speculative execution attacks on SGX [59, 208], the attack *does not require* any code gadget or any other form of cooperation from the victim enclave. In fact, the attack reads all the secrets of the victim enclave without requiring that enclave to execute any instruction. In particular, the attack bypasses all currently proposed side-channel mitigations for SGX [246, 250, 61, 209], as well as proposed countermeasures for speculative execution attacks [124, 134].

At a high level, an attacker can maliciously retrieve memory contents of the victim enclave by remapping the victim memory into the address space of the attacker enclave, which allows bypassing SGX's default enclave-isolation mechanism of abort page semantics. The attacker then prefetches the victim's data into the L1 cache without the victim's involvement by leveraging the cache behavior of SGX paging instructions, thus dramatically improving the attack effectiveness. Finally, the attacker uses speculative execution to perform segmentation-fault-free access to the victim's memory. The interested reader is referred to Section 5.4 for additional details.

Breaking the Integrity of Sealed Data Going beyond attacks on the SGX confidentiality properties, I show the first attack that compromises SGX's *long-term storage integrity* guarantees. More specifically, in addition to secure computation, SGX also aims to provide private and authenticated long-term storage, which is implemented via a special *sealing* Application Programming Interface (API) [29]. This storage mechanism allows enclaves to encrypt and verify data stored by the (untrusted) operating system.

[1] Intel released a micro-code patch mitigating the attack described in this chapter on Aug. 14th 2018.

As I show in Section 5.5, I can use the attack on SGX's confidentiality to extract the sealing key from a victim enclave that uses the SGX sealing mechanism. We note that extracting this key from the address space of an enclave is challenging as the SGX Software Development Kit (SDK) implementation of the sealing API [133] zeros out the sealing key from memory immediately after using it, thereby requiring the attack to intercept the key before it is destroyed. After recovering the sealing key, I use it to unseal and read the sealed information, then modify and reseal it. As SGX provides no means to detect such a change, the victim might now operate on data corrupted by the attacker.

Breaking Remote Attestation Finally, I turn the attention to SGX's remote attestation mechanism, which allows an enclave to prove to a remote party that it has been initialized correctly and is executing on a genuine (presumably secure) Intel processor.

As we show in Section 5.6, we can mount the aforementioned attacks on the SGX *Quoting Enclave*, dump its entire address space, and retrieve its sealing key. Besides being the first documented attack on a production enclave, this attack is particularity devastating as we use the sealing key to unseal the persistent storage of the Quoting Enclave, which contains the machine's private attestation key. With this key in hand, we can construct malicious SGX simulators that pass the entire attestation process, masquerading as enclaves that are allegedly running on genuine Intel processors with the SGX security guarantees. As the simulated enclaves do not offer any security guarantees, this attack undermines the trustworthiness of SGX's attestation mechanism.

Exploiting SGX's Privacy Guarantees The SGX attestation protocol is designed with privacy in mind, and does not reveal the identity of the attesting machine to the remote verifying party. As such, the remote party has no way of telling *which* keys were used for the attestation. Consequently, until revoked by Intel, even a single leaked attestation key can be used for *all* malicious simulators, without the remote parties being able to distinguish them from genuine SGX machines. Thus the leak of even a single key jeopardizes the trustworthiness of the entire SGX ecosystem.

Brittleness of the SGX Ecosystem To the best of our knowledge, this attack is the first direct attack on the confidentiality of the SGX enclaves that makes no assumptions about code running in a victim enclave. By leveraging this attack, the adversary may break the integrity of the SGX long-term storage and the trustworthiness of the remote attestation protocol. As such, this chapter highlights the brittleness of the current SGX design, because a flaw in confidentiality leads to a cascading set of compromises that undermine the root of trust in the ecosystem.

5.1.2 Targeted Hardware

Experimental Setup The attacks described in this chapter are applicable to any presently shipping SGX-enabled Intel CPU. The experiments were conducted experiments on a NUC7i7BNH machine equipped with an Intel Kaby Lake Core i7-7567U processor featuring 2 physical cores and 64 KB of 8-way set associative L1 data cache. The machine was running a fully updated Ubuntu server 16.04, which includes microcode and operating system countermeasures against previous speculative execution attacks (e.g., Spectre and Meltdown). The attack was verified to successfully read enclave contents on machines featuring Core i7-6770HQ, i7-6700K, i7-6700, i7-6500U.

More information regarding the L1TF vulnerability is available in [269, 282, 120, 266, 95, 187].

5.1.3 Threat Model

The Attacker Controls the Operating System. We assume that the attacker controls the operating system and, in particular, can install kernel modules or otherwise execute code with supervisor (ring-0) privileges. The attacker is therefore capable of controlling the virtual-to-physical memory mapping of processes and execute SGX instructions. We note that while we assume a very strong adversary, such a malicious attacker is well within the SGX threat model. Specifically, an SGX enclave is designed to be “protected even when the BIOS, VMM, OS, and drivers are compromised” [123].

No Physical Access While the attacker requires supervisor privileges, the attacks presented in this chapter can be conducted remotely. We do not assume any physical access to the attacked machine, including its memory, memory bus, motherboard, etc.

The Attacker Can Observe *When* a Secret is in Memory We assume the attacker is capable of launching the victim enclave, and estimating *when* the enclave contains secrets. In a simple scenario, the observation is made by the attacker invoking specific enclave functions that generate a secret locally or request a secret from an external party. In section 5.5, we explain how an attacker can pause an enclave to retrieve a secret in memory before it is erased.

We stress that the interaction with the victim enclave is made for *observing* when the secret is in memory. No interaction with the victim enclave is required for *extracting* the secret. At any time, the attack allows the entire victim memory space to be extracted.

No Assumptions on the Victim’s Code The attack makes no assumptions on the victim’s code or on its layout in memory. Unlike Spectre [155, 124], the attack does not require any special code gadgets [208]. Even if the code is encrypted or Address Space Layout Randomization (ASLR) is used [246], the attack is capable of reading the contents of the enclave *after* it is decrypted and after ASLR code placement randomization is completed.

5.1.4 Related Work

Exploiting Operating System Control. Excluding the OS from the TCB gives potential attackers powers that do not exist in more traditional attack models. The controlled channel attack [286] uses the OS’s control over the enclave’s memory mapping to leak information about the enclave's operation. Under this attack, the OS protects the enclave pages by manipulating virtual memory permissions, preventing access. When the enclave attempts to access a protected page, the processor traps to the OS, which records the access before enabling permission and allowing the enclave to continue. The attack can recover high-resolution data, including outlines of JPEG images [286] and cryptographic keys [251, 280].

In addition to page faults, the operating system can also monitor other effects of page table access, including whether a page is dirty or has been accessed, and the caching status of the page table or the translation lookaside buffer [276, 271]. This approach reduces the overhead and side effects of the controlled channel attack.

micro-architectural Attacks. micro-architectural side channel attacks are considered outside the scope for SGX [129, 127], hence it is not surprising that SGX is vulnerable to such attacks. However, operating system control gives attackers additional powers when deploying micro-architectural attacks.

One such power is the ability to interrupt the victim enclave frequently, allowing the attacker to monitor the cache [194, 193] or the branch predictor unit [163, 77] after almost every victim instruction. Attackers can also use the operating system control to reduce system activity and the noise it induces on micro-architectural attacks. For example, the operating system can block interrupts and ensure that the attacker thread and the enclave execute on the two hyperthreads of the same core [49]. Furthermore, the operating system has access to performance information that is not normally available to, or is very noisy when used from user processes [163, 49, 99].

Speculative Execution Attacks on Enclaves. Both [208] and [60] demonstrate that the Spectre attack [155] works on SGX enclaves. Both attacks are demonstrated only against specially crafted enclaves, and as such are more at the proof-of-concept stage rather than being practical attacks.

Both attacks rely on executing vulnerable code within the victim enclave. The attack described here, in contrast, does not require any specific code in the victim enclave and can extract all of the memory contents of the enclave without executing any of the enclave code. While existing work shows vulnerable gadgets exist in the SGX SDK [208], these attacks can be mitigated by patching the SDK and removing these gadgets. The attack does not rely on vulnerabilities in SDK.

Denial of Service Attacks. The use of encrypted and authenticated memory protects enclaves from subversion via attacks, such as Rowhammer [151], that modify the contents of memory. However, the pitfall of this protection is that Rowhammer attacks can now be used to mount a denial-of-service attack on the entire machine, as the unauthorized memory changes lock the processor, requiring a power cycle for execution to resume [147, 102].

Other Attacks on Enclaves. SGX offers only limited protection to vulnerabilities in the code running within enclaves, which can compromise enclave security. One example is Edger8r [143], a timing leakage in the Intel SGX SDK that allows attackers to retrieve some contents of the attacked enclave. [161] show that memory corruption vulnerabilities in SGX enclaves can be exploited and explain how to mount Return Oriented Programming [247] attacks on such vulnerabilities. Finally, AyncShock [279] shows how to exploit synchronization bugs to hijack the enclave control flow.

Attacks from Enclaves. Because the contents of enclaves cannot be observed by the operating system, malicious code may run undetectably in enclaves. SGX provides some protections against malicious enclaves. In particular, enclaves execute in user space, and thus cannot invoke privileged instructions. Furthermore, several non-privileged instructions are disabled in enclaves, for example, `IN` or `OUT`, which perform input/output operations and `SMSW`, which may leak kernel information. However, this protection does not extend to micro-architectural attacks. Consequently, enclaves can leak information through cache attacks [243] and modify the contents of memory using the Rowhammer attack [102].

Defenses. Several mitigation techniques for SGX attacks have been proposed. T-SGX [250] uses a Transactional Synchronization Extensions transaction to catch interrupts, so the enclave can identify the page faults used in the controlled channel attack [286] and the timer interrupts used in other attacks [194, 163, 77]. [251] defend against the controlled channel attack by forcing a deterministic access pattern to enclave pages. HyperRace [61] aims to protect against attacks that rely on hyperthreading, such as [49]. HyperRace uses a data race to implement a co-location test with which an enclave thread verifies that it is co-located on the same core as a dedicated *shadow* thread, thereby ensuring that no attacker concurrently executes on the same core. Similar techniques are used in Varys[209]. SGX-Shield [246] randomizes enclaves' address space layout to protect against memory-based attacks, including the controlled channel attack. DR.SGX [48] protects from some cache-based attacks using fine-grained randomization of enclaves' data locations. SGXBounds [160] provides memory safety for SGX enclaves by tagging pointers with bounds information.

5.2 Background

5.2.1 The Flush+Reload Attack

Flush+Reload [292, 109] is a cache-based micro-architectural attack technique that detects access to a shared memory location. The technique consists of two main operations. The *flush* operation evicts the contents of a monitored memory location from the cache. Typically, this is done using a dedicated instruction, such as the `clflush` instruction on x86 architecture. The *reload* operation then measures the time it takes to access the monitored location. Based on this time, it determines whether the contents of the monitored location was cached prior to its execution.

In a typical attack scenario, an attacker flushes one or more monitored locations. It then either executes an operation it wants to analyze or waits until it is naturally executed by the victim. The attacker then reloads the monitored locations, while recording the amount of time required to perform the reload. As the analyzed operation accesses (and thereby caches) some of the monitored locations but not others, the attacker is then able to learn information about the memory access pattern of the analyzed operation. Finally, as memory access patterns often reveal information about the *inputs* of the analyzed operation, the attacker is often capable of completely recovering these inputs. Flush+Reload has been extensively used for side channel attacks [18, 84, 292, 146, 85, 222, 104, 172, 155, 296].

5.2.2 Speculative Execution

To improve performance and utilization, modern processors execute multiple instructions concurrently. In a nutshell, the processor tries to predict the future instruction stream. By executing multiple instructions from the predicted stream in parallel, the processor is able, for example, to use the time it waits for data to arrive from memory to execute future instructions. For linear code, i.e. code that does not branch, prediction of the future instruction stream is straightforward. For non-linear code, processors employ multiple strategies for predicting the outcome of branches.

Execution of future instructions is inherently *speculative*. The actual instruction stream may differ from the predicted one. Two scenarios that may result in prediction errors are branch *misprediction*, where the branch predictor incorrectly guesses the outcome of a branch, and the occurrence of traps that interrupt the instruction stream. To maintain correct program behavior, the processor does not commit the results of speculatively executed instructions. Instead, completed instructions are kept in a *reorder buffer* until all preceding instructions have successfully completed. When the processor discovers it erroneously speculated an instruction stream, these instructions are *abandoned* and the results of their execution do not affect the state of the program.

5.2.3 Spectre and Meltdown

When the CPU abandons speculatively executed instructions, it does not fully revert the side-effects these instructions have on its micro-architectural state. Spectre and Meltdown [155, 172] exploit these side-effects to leak information across protection domains. The attacks cause the CPU to speculatively execute a *gadget* that implements the transmitting side of a covert channel, sending information that would otherwise be unavailable to the receiver.

More specifically, the Meltdown attack exploits a race condition in vulnerable processors that allows user processes to read kernel data. Specifically, when a user program attempts to read from a kernel address, the processor validates the access while at the same time it speculatively executes the instructions that follow the kernel memory read. By placing a gadget that sends the contents of the kernel memory address through a covert channel, an attacker can retrieve the contents of that address.

Similarly, the Spectre attack leaks memory using a speculatively executed gadget, however instead of bypassing memory protection, it exploits branch misprediction. More specifically, the attacker first trains the branch predictor to predict a desired outcome of a branch, resulting in the execution of the gadget. It then executes the branch with malicious data that produces a different outcome. Due to the prior training, the new outcome of the branch is mispredicted. The gadget is speculatively executed with the malicious data, leaking information through a micro-architectural channel, which the attacker observes to retrieve the information.

5.3 Intel Software Guard Extensions and its Threat Model

Intel Software Guard Extensions (SGX) [183, 113, 29] is an extension of the x86_64 instruction set, supporting secure execution of code in untrusted environments. SGX creates secure execution environments, called *enclaves*, which protect the code and data residing inside them from being maliciously inspected or modified. Additionally, SGX provides an ecosystem for remote attestation of enclaves' software and the hardware on which they run.

The SGX threat model assumes that the only trusted system components are the processor and Intel-provided and Intel-signed *architectural enclaves*. After booting, only the processor is trusted. The trust is extended to the *architectural enclaves* by hard-coding the public key used to sign them into the processor. Other than the *architectural enclaves*, SGX does not trust any software executing on the processor, including the operating system, the hypervisor, and the firmware (BIOS). The processor's microcode, however, is considered part of the processor and hence trusted.

Furthermore, SGX does not trust external hardware components, such as the system's memory. The operating system can manage enclaves, but it is unable to view or modify their contents. We note that the SGX threat model does not protect against denial-of-service attacks, i.e. the operating system can prevent an enclave from executing. Furthermore, the threat model does not protect against micro-architectural side channel attacks [130, 163, 286, 271].

We now describe the aspects of the SGX implementation relevant to this chapter. Further information can be found in [131, 136, 69].

5.3.1 Memory Encryption

To protect enclaves' data from the operating system, the firmware of the machine reserves a range of memory called the *Processor Reserved Memory* (PRM), which contains a region encrypted using the Intel Memory Encryption Engine (MEE) [107], as well as metadata used for SGX and for MEE.

The main aim of MEE is to protect against an adversary that has physical access to the memory of the host machine. To provide confidentiality of the data, MEE encrypts the data in the PRM. To protect the data integrity, MEE maintains a Merkle Tree [185] of stateful Message Authentication Codes (MACs), which ensure unauthorized modifications, including rollbacks, of the memory are detected. MEE operates between the LLC and the system's memory. Cache lines are encrypted when written to memory and decrypted and validated when read from memory.

5.3.2 Enclave Creation

SGX extends the x86_64 instruction set with a variety of instructions for the operating system, user code, and hypervisors to manage enclaves. Launching an enclave requires a three-step sequence. First, the operating system populates initial data structures that describe the enclave and assigns a contiguous range of virtual addresses, called *ELRANGE*, to the new enclave. The contents of the ELRANGE is private to the enclave and can only be accessed by code running within the enclave. Next, the operating system adds the initial (non-secret) code and data to the enclave by using the `EADD` SGX instruction. Finally, the enclave is initialized; the operating system may not add more code or data after initialization.

For each enclave, SGX keeps an enclave-identity comprised of the enclave developer's identifier and a *measurement* representing the enclave's initial state. The developer's identifier, referred as MRSIGNER in SGX literature, is a cryptographic hash of the public RSA key the enclave developer used to sign the enclave's *measurement*. The enclave *measurement*, representing the enclave's initial state, is a cryptographic hash of those parts of the enclave's contents (code and data) that its developer chose to include in the measurement. The SDK implementation includes in the *measurement* all contents added to the enclave via `EADD`. Following the SGX nomenclature, we refer to this measurement as MRENCLAVE.

5.3.3 Memory Management

To protect the contents of the ELRANGE, pages within this range must map to the PRM. More specifically, part of the PRM is used for the *Enclave Page Cache* (EPC), in which enclaves' pages are stored. Each page within the ELRANGE of an enclave can be either

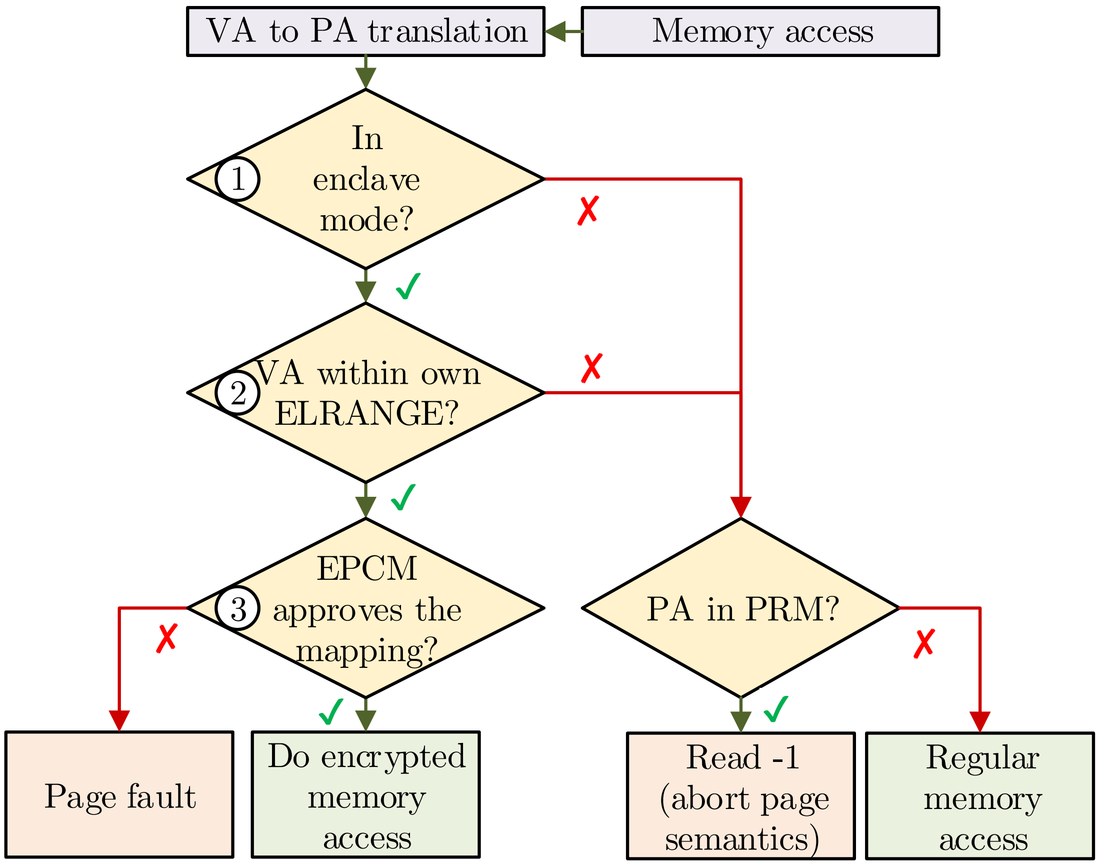


Figure 5.1: SGX's memory access flow, as described in [183].

loaded in the EPC, where the enclave can use it, or (securely) stored outside the EPC, where the OS must load it into the enclave before it may be accessed. The operating system is given the primitives required for managing the EPC, without being able to observe or to modify the contents of the EPC pages. These mechanisms enable the operating system to implement a secure paging facility for enclaves whose footprint exceeds the available capacity of the EPC.

The `ENCLS` instruction supports several functions for loading and unloading pages to and from the EPC and for managing the metadata associated with these pages. Specifically, the `EWB` leaf instruction encrypts the contents of an EPC page and copies the encrypted contents to the unprotected memory. `EWB` also maintains (in the EPC) the required metadata that identifies the page's version and virtual address in ELRANGE, to protect the evicted page's contents. Similarly, the `ELDU` instruction loads the encrypted contents of an EPC page from the unprotected memory.

Because SGX enclaves execute within the virtual address space of a process, the translation of enclave addresses to physical addresses must be trusted. However, the operating system controls the mapping of virtual to physical addresses and can change this mapping at will. Instead of ensuring the correctness of the operating system's page table, SGX maintains an internal data structure called the *Enclave Page Cache Map* (EPCM), which tracks the mapping and the identities of frames within the EPC. The EPCM provides the reverse mapping of the physical-to-virtual address mapping encoded in the page tables.

The purpose of the EPCM is to protect against attempts to bypass the SGX protection by mapping an EPC page at a different virtual address. This protection is achieved by adding several validation tests following a virtual to physical address translation. Figure 5.1 shows a flow chart of the validation process. First (Step ①), the processor checks whether the access is from within an enclave. Non-enclave code is blocked from any access to the PRM by providing *abort page semantics* for such accesses. That is, reads from the PRM return an all-one data (`0xFF`) and writes to the PRM are ignored.

Code executing within an enclave can access both the unprotected (normal) memory and its own ELRANGE. Thus, in Step ②, the processor checks that the accessed virtual address is within the ELRANGE of the accessing enclave. Failing this test, the processor reverts to the default behavior, i.e. normal access for user memory and abort page semantics for PRM access.

Finally, in Step ③, the processor verifies that the data in the EPCM matches the attempted access. If the verification fails, the processor issues a page fault to abort the access.

5.4 Reading Enclave Data

In this section, we describe the first attack, which allows us to read data located within the address space of some victim SGX enclave. At a high level, the attack is constructed to coerce Steps ①, ②, and ③ in Figure 5.1 to result in a page fault due to a failure of

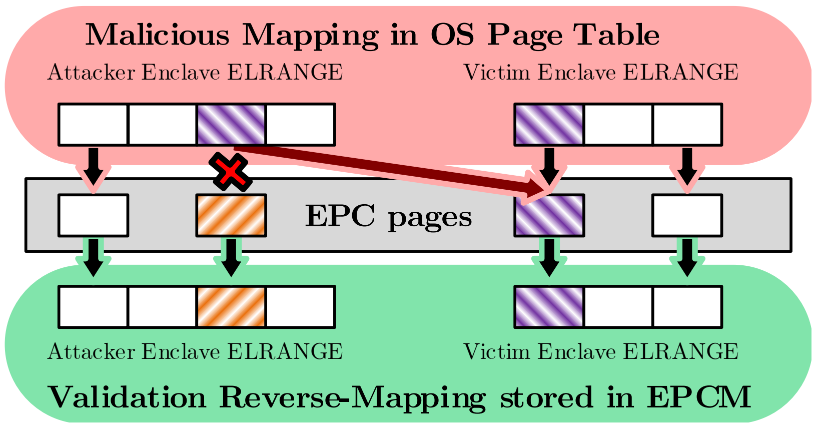


Figure 5.2: Malicious mapping. At creation time, the OS assigns the enclave a contiguous virtual address range (ELRANGE). The Virtual Address of an enclave page may be assigned a physical address of a page in the EPC. In SGX-ray, the attacker maps a virtual page within *its own* virtual range to the victim's physical EPC page.

verification of an EPCM test. We then use a variant of the Meltdown attack to subvert the page fault and read the victim's data.

Before describing each of the components in the attack we first explain how we establish a cache-based covert channel, which is used by the speculative execution component of the attack.

5.4.1 Establishing a Cache-Based Covert Channel

Similar to prior work [155, 59], we leverage the Flush+Reload covert channel. The channel consists of three operations, abstracted as functions in Listing V.1.

Channel Initialization The `prepare` function is used to initialize the covert channel for sending a byte, encoding the byte's value via the cache state of a corresponding element of `probeArray`. As such, to initialize the channel, the `prepare` function flushes all the elements of `probeArray` from the CPU's cache. To simplify the attack's description, we assume that `probeArray` is a global shared buffer, which is accessible to all the subroutines used by the attack. We, therefore, omit its passing via function parameters and access it as a shared global variable.

Transmitting a Byte The `send` function takes a one byte argument `data`, and transmits it via the covert channel by accessing a corresponding element from `probeArray` (Line 7), thereby bringing it to the CPU's cache. As we can only distinguish accesses at a cache-line granularity, we multiply (`data`) by 256 before accessing `probeArray`. With a cache-line size of 64 bytes, spreading indices by multiples of 256 ensures that different values of `data` index to different cache lines. We find that using lower multipliers increases noise level due to prefetcher activity.

Receiving a Byte The `recv` function reads the channel and returns a boolean array indicating possible values of the transmitted byte. To receive the data transmitted by `send`, `recv` reloads each of the addresses that `send` might have accessed, while measuring the time required to perform the load (Lines 15-18). If the required time is below an (empirically set) cache-hit threshold, this indicates that the `send` function might have transmitted the value of `guess` via the cache-based covert channel. That is, the `send` function accessed `probeArray[guess*256]`, thereby bringing it into the CPU's cache, accelerating the subsequent probe access.

We note here that while the Flush+Reload channel is usually clean, some measurement errors remain possible, typically due to unrelated system activity. Consequently, instead of returning a single guess for the transmitted data, `recv` returns a *vector* of results that indicates which values where possibly transmitted by `send`. In Section 5.4.5 we show how to avoid these errors and correctly recover the transmitted data. Finally, to avoid the CPU's prefetcher modifying the cache state, `recv` does not access the indices of `probeArray` sequentially and follows the approach of previous works [155, 59] by applying a simple linear permutation to the order of accesses (Line 13).

```
prepare() {
   for (i=0; i < 65536; i++)
      flush(probeArray[i]);
}

send(data) {
   t = probeArray[data * 256];
}

recieve() {
   for (i = 0; i < 256; i++) {
      // mix guess to avoid prefetching probeArray
      guess = ((i * 167) + 13) % 256;
      // compute address to probe
      addr = &probeArray[guess*256];
      t1 = rdtscp(); // read timer
      temp = *addr;  // access probing array
      t2 = rdtscp(); // read timer
      if (t2-t1 <= CACHE_HIT_THRESHOLD){
         results[guess]=1;
      } else results[guess]=0;
   }
   return results;
}
```

Listing V.1: Pseudocode of the Flush+Reload covert channel.

5.4.2 The Malicious Hosting Process

The attacker launches the victim enclave in a process and identifies the virtual address range (ELRANGE) of the victim enclave inside the process's address space. The range can be located either via a malicious driver or by inspecting the contents of `/proc/pid/maps`. At this point, a naive attacker might attempt to read directly the enclave address space. However, the virtual addresses of the victim enclave are mapped to physical addresses residing in the EPC, which is part of the PRM. Consequently, as Figure 5.1 shows, such access would result in abort page semantics, i.e., the read returns a value with all bits set, regardless of the actual memory content. Alternatively, the attacker may attempt to read the enclave's data by mounting a speculative execution attack (e.g., Meltdown). However, we found that such attempts also result in abort page semantics.

5.4.3 The Attacker Enclave

To cause a page fault, we first need to pass Step ① of the memory access flow (Figure 5.1). We achieve this by spawning a malicious *attacker enclave*, which performs the read access. We note that there is no need for the attacker enclave to be vetted by Intel, as we can execute the attack enclave in SGX debug mode. However, a naive use of an enclave is not sufficient to coerce a page fault, because the ELRANGE of the attack enclave is different than that of the victim enclave. Consequently, the validation in Step ② of Figure 5.1 fails and the access still results in abort page semantics.

5.4.4 Malicious SGX Driver

To overcome the ELRANGE check deeming the victim's memory address outside of the attacker enclave (Step ② in Figure 5.1), we use a malicious SGX driver that introduces an incorrect and malicious mapping from the *virtual* address space of the attacker's enclave to the *physical* EPC page of the victim enclave (see Figure 5.2). With the malicious mapping in place, the attacker enclave can pass the ELRANGE check when attempting to read EPC pages located in physical memory and belonging to the victim enclave.

However, as mentioned in Section 5.3.3, SGX prevents operating systems from maliciously manipulating the virtual to physical mapping of enclave pages by keeping the reverse mapping inside the EPCM (Figure 5.2). The EPCM contains the reverse mapping from the physical addresses in the EPC to the virtual addresses in the enclave's ELRANGE. When memory is accessed from inside an enclave, the CPU checks the OS-controlled page-table against the EPCM to verify that the OS-managed virtual to physical mapping matches the page's EPCM entry. As shown in Figure 5.1, memory is only accessed in case both mappings match, and an EPCM page fault occurs if a mismatch is detected.

We observe that the EPCM is indexed according to the physical address of the EPC page, meaning that the OS-managed virtual to physical mapping can be verified only *after* the virtual to physical address translation completes. We conjecture that Intel implemented the memory access in parallel to the EPCM verification with both operations performed after resolving the virtual to physical mapping. We now show how to exploit this fact to bypass the EPCM page fault, thereby recovering data from the victim enclave.

5.4.5 Reading Cached Enclave Data

We begin the attack by assuming that the address space of the victim enclave contains some secret that the attacker wants to read and that this secret happens to reside in the CPU's L1 cache. Later, in Section 5.4.7, we show how to remove this assumption, allowing the attacker to read the entire address space of the victim enclave. The attack thus proceeds as follows.

A Cross-Enclave Speculative Execution Attack We begin the attack by setting up a malicious hosting process (Section 5.4.2) which initializes both the victim enclave and the attacker enclave (Section 5.4.3). The process then uses the malicious driver to set up a malicious mapping of the victim enclave's page that contains the data we want to read (Section 5.4.4). Finally, as explained above, in this section we assume that the data that the attacker wants to read resides in the CPU's L1 cache.

Next, the attacker enclave exploits the CPU's branch predictor in order to mount a speculative execution attack on the victim enclave. Unlike Spectre attacks [155], which exploit the branch predictor to read (and subsequently leak) data from within the same address space, the attack exploits the branch predictor to leak information across enclave boundaries while eluding the page fault issued for the illegal access. At a high level, the speculative execution attack consists of three phases, which we now describe.

Step 1: Branch Predictor Training Consider the pseudocode presented in Listing V.2, which is executed by the *attacker enclave* using some `dummyValue` that is provided by the malicious hosting process. During the first five iterations of the for loop in Line 3, the selected address is `dummyAddress` (which is the address of the variable `dummyValue`)

```
 1 speculative_read(dummyValue, addressToRead ){
 2    dummyAddress = &dummyValue;
 3   for (i = 5; i >= 0; i--) {
 4     selectedAddress =
 5         ct_select(i, dummyAddress, addressToRead);
 6     flush(&dummyAddress);
 7     if (selectedAddress == dummyAddress){
 8       // read value from selected address
 9       value = *selectedAddress;
10       // send value via a Flush+Reload covert channel
11       // using the code in Listing V.1
12       send(value);
13     }
14   }
15 }
```

Listing V.2: Pseudo code of the attacker enclave. The function `ct_select` outputs addressToRead when $i = 0$ and dummyAddress otherwise.

and it is the case that the branch in Line 7 evaluates to true. Next, as it is the case that `selectedAddress` equals `dummyAddress`, Lines 9 and 12 actually send the attacker-provided `dummyValue` through the Flush+Reload cache covert channel. We note that while this does not provide any additional information to the attacker, it does train the CPU's branch predictor that the branch in Line 7 typically evaluates to true.

Step 2: Attack Phase Consider the final iteration $(i = 0)$ of the loop in Line 3. As $i = 0$, the selected address in Line 5 is the address from which to read in the victim enclave, as provided by the hosting process. Moreover, as Line 6 flushes the value of `dummyAddress`, it is impossible to evaluate the branch in Line 7 until `dummyAddress` is fetched from memory. Rather than waiting, the CPU consults the branch predictor and speculatively executes the branch in Line 7, predicting the condition to be true. Next, as `selectedAddress` equals `addressToRead` (since $i = 0$), both values actually point (via the malicious mapping described in Section 5.4.4) into a physical page belonging to the victim enclave. As the data located in `addressToRead` already resides in the L1 cache, the CPU proceeds to speculatively execute Line 9, setting `value` to be the value located in `addressToRead`. Finally, while `dummyAddress` is still fetched from memory, the

CPU also proceeds to speculatively execute Line 12, thereby leaking `value` through a cache-based Flush+Reload covert channel.

Step 3: Fault Suppression We note that executing Line 9 when $i = 0$ actually performs a memory access to `addressToRead`, which points (via the malicious memory mapping) to a physical page of the victim enclave. Next, as discussed in Section 5.4.4, SGX holds a redundant copy of relevant page table entries in the EPCM, to verify the virtual to physical mapping, as managed by the (potentially malicious) OS. Thus, as shown in Figure 5.1, executing Line 9 for the case of $i = 0$ where the EPCM does not match the page table entry should have resulted in an EPCM page fault. However, recall that the attack actually executes Line 9 speculatively, while waiting for `dummyAddress` to arrive from memory (after being flushed in Line 6). When the value of `dummyAddress` eventually does arrive from memory, the CPU realizes the branch in Line 7 was mispredicted, for the case of $i = 0$, and rolls back the execution of Lines 9—12 without emitting an EPCM page fault. However, as the cache state is not rolled back, the host process can receive the value obtained from the victim enclave, sent through the cache-based covert channel.

Relation to Meltdown and Spectre The attack describe in this chapter is, in fact, a hybrid between the techniques proposed in the Spectre [155] and Meltdown [172] papers. More specifically, it uses a technique similar to Meltdown where the attacker abuses speculative execution to dereference a pointer to a privileged address space, and subsequently leaking a result through a cache-based side channel. On the other hand, the attack also uses the fault suppression technique from Spectre, as it mistrains the CPU's branch predictor regarding Line 7 of Listing V.2. While such a combination was theoretically considered in the Meltdown paper [172], to the best of my knowledge, the work described in this chapter is the first to provide an explicit implementation and empirical evaluation of this technique.

5.4.6 Recovering the Leaked Data

So far we have focused on describing how the attack reads data from the victim enclave and transmits it using a cache-based covert channel. In this section, I focus on describing the receiving side of the attack, which recovers the data of the victim enclave from the cache-based covert channel. First, we notice that the code in Listing V.2 actually transmits two values using the Flush+Reload based covert channel. Indeed, during the first five iterations of the loop in Line 3 of Listing V.2, the code sends the value of `dummyValue` (as provided by the attacker). Next, during the final iteration ($i = 0$), the code in Listing V.2 sends the value located in `addressToRead`. Thus, to learn the value located in `addressToRead`, the attacker must somehow distinguish the transmission of `dummyValue` from the transmission of other values (presumably obtained during the last iteration of the loop in Line 3). The problem is further compounded by the existence of sporadic channel noise that sometimes corrupts some of the transmissions.

At a high level, we solve both issues using an approach similar to that of [155]. That is, we transmit the value located in `addressToRead` multiple times, each time providing a different `dummyValue` to be transmitted along with it. We then aggregate the results across all the multiple transmission attempts, returning the most common value as the value located in `addressToRead`. As the correct value located in `addressToRead` remains the same while `dummyValue` is constantly changed, we expect that the most common value transmitted via the cache-based covert channel will be the value located in `addressToRead`.

More specifically, after creating the incorrect memory mapping between the attacker's enclave and the victim enclave (as described in Section 5.4.4), the attacker proceeds to execute the pseudocode presented in Listing V.3, setting `addressToRead` to be a virtual address mapped to the victim enclave (via the incorrect mapping). At a high level, the pseudocode presented in Listing V.3 performs the following for each try index $i = \mathsf{maxTries}, \cdots, 1$.

- **Step 1: Preparing the Covert Channel** The attacker starts every attempt to read the data

```
1  read_value(addressToRead, maxTries){
2     int byteScores[256];
3     for (i = maxTries; i > 0; i = i - 1){
4      // prepare a cache-based coveret channel
5      // using the code in Listing V.1
6      prepare();
7      dummyValue = i % 256;
8      // read the value from the victim enclave and
9      // send it via the cache-based coveret channel
10     // using the code in Listing V.2
11     speculative_read(dummyValue, probingArray, addressToRead)
12     //receive scores for each possible value sent
13     //via the  cache-based coveret channel
14     //using the code in Listing V.1
15     results = receive()
16     //aggrage the scores across many tries
17     for (j = 0; j < 256; j++){
18      byteScores[i] = byteScores[i] + results[i];
19     }
20   }
21   // return the byte value with the highest score:
22   return argmax(byteScores);
23 }
```

Listing V.3: Pseudocode of reading a single byte.

of the victim enclave by preparing a cache-based covert channel. This is achieved in Line 6 of Listing V.3 by calling the `prepare()` function of Listing V.1.

- **Step 2: Leaking the Victim's Memory via Speculative Execution** The attacker then mounts a speculative execution attack on the victim's enclave by invoking (Line 11) the code presented in Listing V.2, supplying it with addressToRead and using the try index *i* as the dummy value.

- **Step 3: Receiving Data via the Covert Channel** After mounting the speculative execution attack, the attacker calls the receive function (Line 15) of the cache-based covert channel to receive the data transmitted during the speculative execution attack. As the cache-based covert channel has been prepared, we expect the receive function (Line 15) to return high scores for only two specific values: *i* % 256 (which is used as `dummyValue` and sent during the branch predictor training phase in Section 5.4.5) and the value located at `addressToRead` (which is sent during the attack phase in Section 5.4.5). The attacker then sums the scores for each value over all tries into the `byteScores` buffer, where `byteScores[j]` corresponds to the score of a j-valued byte.

Recovering the Byte's Value With the above steps performed for each try index $i = \mathsf{maxTries}, \cdots, 1$, the attacker has collected statistics for each try index about which values were received via the covert channel. As mentioned above, for every try index $i = \mathsf{maxTries}, \cdots, 1$ we increase the score of `bytesScores[i]` and `bytesScores[value]` where value is the data located at `addressToRead`. Thus, after performing the above steps for all $i = \mathtt{maxTries}, \cdots, 1$, we expect that `bytesScores[value]` will equal `maxTries` while all other values in `bytesScores` are significantly lower. Thus, returning `argmax(byteScores)` successfully recovers value (Line 22 of Listing V.3).

5.4.7 Reading Uncached Enclave Data

The attack described thus far explicitly assumes that that the value located in `addressToRead` is also present in the L1 cache. We now describe a method to remove this assumption, allowing the attacker to read any data located inside the victim's virtual memory, including data that is *never* accessed by the victim enclave.

Managing the Enclave Page Cache (EPC) Although SGX assumes an untrusted OS, SGX nevertheless *does* rely on the host's operating system for managing the limited space allocated for the EPC in the machine's physical memory. As explained in Section 5.3.3, the operating system uses the `EWB` and `ELDU` leaf instructions to securely copy enclave pages out of and back into the EPC. We observe that while decrypting and verifying an encrypted enclave page, the `ELDU` instruction loads the page's contents into the CPU's L1 cache. Crucially, `ELDU` never evicts the page from the L1 cache, leaving the page's contents cached after the instruction terminates.

Exploiting ELDU Thus, the attack performs the following. Going over the pages of the victim enclave, the malicious SGX driver described in Section 5.4.4 first uses `EWB` to evict the page from the EPC only to immediately load it to the EPC using the `ELDU` instruction. As the `ELDU` instruction loads the page into the L1 cache and does not evict it afterwards, we can use the attack presented in Section 5.4.5 to extract its content. Finally, the entire attack process is repeated for the next page of the victim enclave.

5.4.8 Overall Attack Performance

In this section, we empirically evaluate the performance of the attack in retrieving data from the address space of the victim enclave.

Reading Specific Memory Areas Using the setup described in Section 5.1.2, we begin the evaluation with the assumption that the attacker desires to recover the contents of a specific memory region inside the victim enclave. For this experiment, we launch a victim enclave and filled 128 consecutive pages (512 KiB) of its memory with random data. Initially, using the attack described in Section 5.4.7, we can read all of these pages, achieving a reading speed of 1.63 KiB/sec and 56.6% accuracy.

Reducing Prefetching and Cache Noise Next, to minimize cache pollution and prevent eviction of cache lines containing secret data, we disabled several memory prefetchers [273]. While this operation requires elevated privileges, recall that the SGX threat model assumes a malicious OS, thereby giving the attacker elevated privileges on the target machine. Lastly, to improve the attack accuracy, we gradually increased the value of `maxTries` in Line 3 of Listing V.3 if the extracted value is 0x00 or 0xFF. Employing these two optimizations improves the read accuracy to 85.68%, with a read speed of 1.58 KiB/sec.

Eliminating Noise Caused by Entering and Leaving the Enclave Inspecting the errors in the recovered data, we identify that these typically occur in bursts of cache line size (i.e., 64 bytes). Observing that entering an enclave and exiting it are complex operations, we hypothesize that these operations generate sufficient noise in the cache to evict the victim enclave data during the call to `speculative_read` in Line 11 of Listing V.3. We thus follow the approach proposed by [281, 216] of having a thread continuously running inside the enclave calling `speculative_read`, avoiding the enclave enter and exit operations. Using this optimization we were able to successfully read 99.93% of the bytes at 0.88 KiB/Sec.

Reading the Entire Enclave Contents To read the contents of an entire victim enclave, without knowing the specific virtual address of interest, we inspect the contents of `/proc/pid/maps` to find the physical addresses that match each of the enclave pages. While the entire range of the tested victim enclave is 16 MiB, only 4 MiB are allocated. Due to the optimization of dynamically adjusting the number of repetitions in Listing V.3 if the extracted value is 0x00 or 0xFF, reading pages containing only zeros is significantly slower, yielding a read speed of 13.5 bytes/sec, compared with 880 bytes/sec for other pages. Overall, reading the entire enclave took 3:42 hours with 99.77% of the bytes successfully read.

Attacks on Other Intel Processors The attack is not specific to the Kaby-Lake i7-7567U processor, used in the above-described performance evolution and in principle can be applied to any SGX-equipped CPU. Indeed, similar results were also obtained when attacking a previous generation of Intel CPUs, namely Skylake i7-6770HQ, i7-6700K, i7-6700, and i7-6500U.

5.5 Attacking SGX's Sealing Mechanism

The attack described in Section 5.4 is capable of breaching SGX's confidentiality guarantees, by reading the virtual address space dedicated to any SGX enclave available on the target machine. It cannot, however, breach SGX's integrity guarantees as it is unable to modify the contents of enclave memory.

In this section, we show an attack against SGX's sealing mechanism, which is a mechanism designed to provide enclaves with encrypted and authenticated persistent storage. In a nutshell, we use the attack from Section 5.4 to recover the sealing keys from within the address space of the victim enclave. We then use the recovered sealing keys to unseal the victim's persistent storage and replace its content. Finally, we use the recovered sealing keys again, this time to seal the new contents, by encrypting it and calculating the new authentication information. The victim enclave can now successfully unseal the (malicious) sealed data and, since the authentication information was correctly computed, believes the data is genuine and has not been tampered with. Before presenting the attack, we now provide further background information about the SGX sealing mechanism.

5.5.1 SGX's Sealing Mechanism

SGX provides enclaves with a mechanism for an encrypted and authenticated persistent storage. During CPU production, a randomly generated *Root Seal Key*, which is not kept in Intel's records, is fused in every SGX-enabled CPU. Using this key, the CPU can derive a sealing key, which can be used to encrypt and authenticate information from within the enclave's address space. Data that is *sealed* with this key, i.e., encrypted and authenticated, can be safely passed to the operating system for long-term storage, for example, on the computer's disk. SGX provides two types of sealing mechanisms, which we now describe (See [29, 131] for additional details).

Sealing Using the Enclave's Identity As described in Section 5.3.2, each enclave has a unique field, called MRENCLAVE, which is a cryptographic hash of the contents used to initialize the enclave code as well as some additional properties. Using the values of the Root Seal Key, MRENCLAVE, and the CPU Security Version Number (SVN) an enclave can use the EGETKEY instruction to derive a unique sealing key for sealing data before passing it to the operating system for long term storage. Note that as a consequence of this approach, for the same Root Sealing Key (i.e., on the same CPU), different software versions of the same enclave have different sealing keys. This prohibits both data migration between different versions of enclaves as well as using these sealed keys for intra-enclave communication.

Sealing Using the Developer's Identity An alternative option to the one discussed above is to generate the sealing key using the Root Seal Key, the SVN, and MRSIGNER (instead of MRENCLAVE). As we explain in Section 5.3.2, MRSIGNER is a cryptographic hash of the public RSA key of the enclave developer and is the same for all enclaves developed by the same vendor. Thus, data sealed in this way is accessible by different versions of the same enclave, as well as by different enclaves belonging to the same vendor.

5.5.2 Extracting SGX Sealing Keys

Key derivation using EGETKEY leaves the sealing key in the memory of the victim enclave. Thus, in principle, it is possible to read this key using the attack, described in Section 5.4 above. However, immediately after using it to encrypt or decrypt the sealed data, the implementation of SGX's sealing API erases the sealing key from memory. Hence, to extract the key we need a method for launching the attack described in Section 5.4 during the data sealing or unsealing process, before these keys are erased from the memory.

A Control Channel Attack We time the attack using a variant of the controlled channel attack [286], inducing an Asynchronous Enclave Exit (AEX) when the enclave calls the encryption function. More specifically, we first examine the shared object file of the victim enclave and find the virtual addresses of the sealing and encryption functions inside the address space of the victim enclave. Next, we use the malicious driver (subsection 5.4.4) to evict from the EPC the page(s) containing the encryption and sealing functions. This induces a page fault and an asynchronous enclave exit whenever the victim enclave attempts to call the encryption functions. However, as the pages containing the monitored functions also contain code of other functions, the controlled channel attack will trigger an enclave exit on accesses to these other functions and does not reveal *exactly* which function caused the exit.

Determining the Precise Function Called We note that during an AEX, the contents of all registers are saved in a dedicated State Save Area (SSA), including the instruction pointer register, which points to the next instruction to be executed by the enclave upon return from the AEX. The SSA is located in the enclave's virtual address space, and therefore its contents can be extracted using the attack from Section 5.4. To find the SSA's address, we follow the pointer to it found in a special enclave page called Thread Control Structure (TCS). While the TCS is not readable even to the enclave, it is readable to the attack described in Section 5.4.

The attack thus proceeds as follows. Upon a page fault due to an access to the *target page* containing the code for the SGX sealing and encryption functions, we use the attack from Section 5.4 to read the contents of the TCS and SSA and recover the value of the instruction pointer. In the case that the instruction pointer points to a function other than the SGX encryption or decryption functions, we load the evicted *target page* back into the EPC, evict a *benign page* we anticipate will be accessed next, and resume normal enclave execution. On the next page fault, caused by access to the *benign page*, we evict the *target page* again.

However, if the instruction pointer points to the beginning of the encrypt or decrypt functions, we know that the victim's seal key is present in the victim's memory, at an address pointed by the `RDI` register, which is used by the compiler for passing function parameters. The attack then proceeds to extract the contents of `RDI` from the SSA. The victim's seal key is then recovered from the address pointed by `RDI` using the attack from Section 5.4.

5.5.3 Empirical Evaluation

We empirically evaluate the attack presented in this section, using the experimental setup described in Section 5.1.2. We implemented a victim enclave which seals and unseals data. We successfully launch the attack as described in this section and extract the sealing key from within the victim enclave. Next, to validate we have the correct key, we implemented custom seal and unseal functions that operate on a seal key, instead of calling SGX key derivation instruction (`EGETKEY`). Using these functions, we can unseal the data sealed by the victim enclave as well as to seal new (malicious) data, outside the victim's enclave. Running the victim enclave again, the new data was successfully unsealed without errors.

5.6 Attacking SGX Attestation

One of most compelling integrity properties provided by SGX is the ability of an enclave to attest to a remote verifying party that it is running on genuine Intel hardware and not on an SGX simulator. This attestation process proves to the remote party that the enclave leverages the data confidentiality and execution integrity properties provided by SGX and, therefore, the remote party can transfer secret data to the enclave. Specifically, the remote party trusts the enclave will not intentionally leak the secret data provided by the remote party and that any data sent by the enclave is a result of a trustworthy execution.

While the attacks described in Section 5.4 and 5.5 are capable of violating the confidentiality of the entire address space of the victim enclave and the integrity of its sealed data, they cannot make the victim violate program semantics, designed by the enclave writer.

In this section, we show that the attack described in Section 5.4 and 5.5, which violate the confidentiality of the enclave address space and sealed inputs actually have devastating consequences for the soundness property of SGX's attestation protocol. More specifically, by mounting the attack described above on the SGX's Quoting Enclave, we are able to recover the private attestation keys, used by the target machine for proving its genuineness. With these keys at hand, we are able to construct a malicious simulator which passes attestation as if it was an SGX enclave running on a genuine Intel processor, while executing the simulator code outside of an actual enclave. As the private attestation keys are all that distinguish genuine Intel hardware from potentially malicious simulators, the remote verifying party has no way of distinguishing between the two and thus cannot trust the computation's output to be correct.

Before describing the attack, we now provide some background about SGX's provisioning, quoting, and attestation processes.

5.6.1 SGX Remote Attestation

The remote attestation process allows a *remote* verifying party to verify that a *specific* software is correctly initialized and executes within an enclave, on a genuine Intel CPU. At a high level, this is performed as follows (see [149] for an extended discussion).

In addition to the Root Sealing Key (Section 5.5.1), every SGX-enabled CPU is also shipped with a randomly-generated Root Provisioning Key (Step 1, Figure 5.3). However, unlike the Root Sealing key, Intel does retain a copy of the Root Provisioning Key, as it acts as a shared secret between Intel and every individual CPU. Next, Intel provides two special enclaves, called the *Quoting Enclave* and the *Provisioning Enclave* which are used in the attestation process.

Attestation Key Provisioning The initialization phase of the SGX attestation protocol consists of the Provisioning Enclave contacting Intel's provisioning server, transmitting the CPU's provisioning ID, and claimed security version (SVN). As the provisioning ID uniquely identifies a specific CPU, it is only accessible to the Intel-signed Provisioning Enclave and is sent encrypted to the provisioning server under Intel's public key. After recovering the root provisioning key, corresponding to the CPU's provisioning ID, the provisioning server and Provisioning Enclave proceed to execute the *Join* phase of Intel's

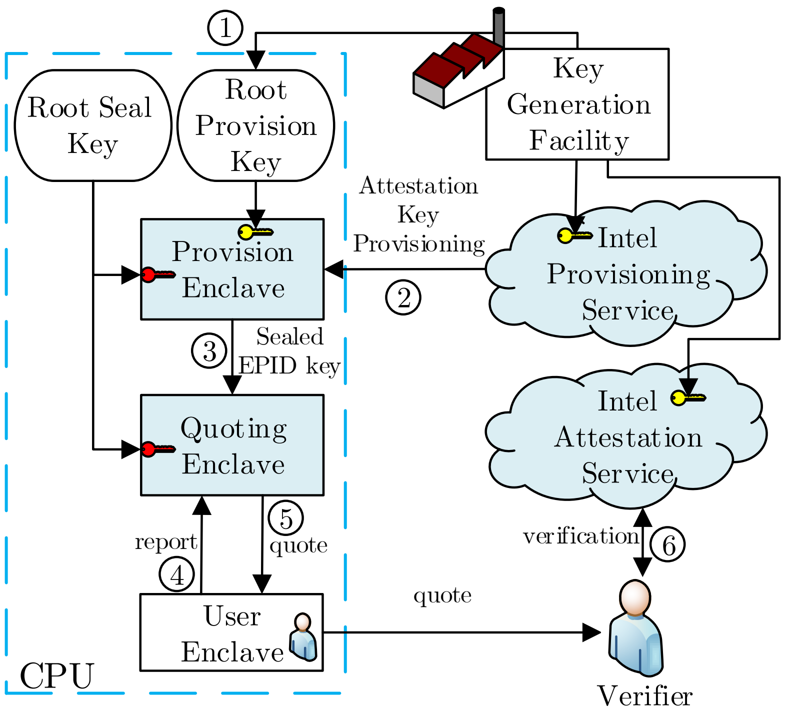


Figure 5.3: SGX's Attestation Process.

Enhanced Privacy ID (EPID) protocol [51], using the root provisioning key as a shared secret (Step 2, Figure 5.3).

At a high level, Intel's EPID protocol is a type of group signature that allows a CPU to sign messages (using its private signing keys) without uniquely disclosing its identity. All that an external observer (e.g., Intel) can do is to verify the signature (thereby becoming convinced that it was signed by a genuine Intel CPU belonging to the group), without being able to link it to any specific Intel CPU or to other signatures it previously signed. See [51] for additional discussion.

Sealing the EPID Key The *Join* phase of the EPID protocol results in the Provisioning Enclave obtaining a private EPID signing key, which is not known to Intel. The Provisioning Enclave then generates a sealing key for sealing the EPID signing key, using the CPU's Root Sealing key, its SVN and the MRSIGNER value of the Provisioning Enclave. It then seals the private EPID key using this sealing key and outputs it to the OS for long term storage

(Step 3, Figure 5.3). Notice that as the Provisioning Enclave is provided and signed by Intel, its MRSIGNER value is a hash of Intel's public key. Consequently, any Intel-signed enclave can unseal the CPU's private EPID key by regenerating the sealing key used to seal it. While this design feature is indeed useful, as it allows the Quoting Enclave (also signed by Intel) to unseal the private EPID key, it is also dangerous as the OS actually has an encrypted copy of the CPU's private EPID keys.

Local Attestation When an enclave wants to prove to a remote verifier that it is running on genuine Intel hardware with a specific security version, it first needs to prove its identity to the *Quoting Enclave*, which is another special enclave provided and signed by Intel, via a processes referred to by Intel as *local attestation* [29, 131]. At a high level, this is done by having the proving enclave use the EREPORT instruction, which prepares a report containing the MRENCLAVE and MRSIGNER values of the proving enclave. The report is also signed using a key that is only accessible to the Quoting Enclave. The proving enclave then passes the report to the Quoting Enclave, which proceeds with the remote attestation process (Step 4, Figure 5.3).

Remote Attestation Upon receiving the report from the proving enclave, the Quoting Enclave performs the remote attestation process, which we now describe. Indeed, after verifying that the report was correctly signed by the EREPORT instruction, the Quoting Enclave proceeds with unsealing the EPID private key that was originally sealed by the Provisioning Enclave. Recall that the EPID private key was sealed using a sealing key derived from the CPU's SVN version, Root Sealing Key, and the MRSIGNER value of the Provisioning Enclave. As both the Quoting Enclave and the Provisioning Enclave are signed by Intel (and thus have the same MRSIGNER value), the Quoting Enclave can regenerate this sealing key and subsequently unseal the private EPID key. Next, using the unsealed private EPID signing key, the Quoting Enclave executes the *Sign* phase of the EPID protocol and signs the report given to it by the proving enclave, creating an *attestation quote*. Finally, the Quoting Enclave returns the quote to the proving enclave, which in turn forwards it to the remote verifying party (Step 5, Figure 5.3).

Attestation Verification After the proving enclave sends the signed quote to the remote verifying party, the remote party interacts with Intel's Attestation Server (IAS [139]) and provides it with the *quote* it obtained from the Quoting Enclave (Step 6, Figure 5.3). Next, IAS performs the *Verify* phase of the EPID protocol while ensuring that the signer's private EPID key has not been revoked by Intel. Intel's server completes the attestation by sending its response (OK, SIGNATURE_INVALID, etc.) to the remote party. The server's response also contains the quote itself and is signed with Intel's signing key, generating a signed attestation transcript which can later be verified by any party that trusts Intel's public key.

5.6.2 Extracting SGX Attestation Keys

In this Section, we describe the attack on SGX's attestation protocol. As explained above, the Quoting Enclave, which can access the EPID signing keys, will not sign a local attestation report without first verifying it. Moreover, as mentioned in Section 5.6.1 above, the operating system actually has a copy of the EPID private keys, which are sealed by a key derived from the CPU's Root Sealing Key. The attack thus proceeds as follows.

Step 1: Recovering the Sealing Keys Using the attack described in Section 5.5 on the Quoting Enclave, the attack recovers the sealing keys used for sealing the EPID signing keys.

Step 2: Unsealing the EPID Signing Keys With the above sealing keys, the attack proceeds to unseal the private EPID keys, originally sealed by the Provisioning Enclave.

Step 3: A Malicious Quote Enclave Using the source code of Intel's Quoting Enclave [133], we have constructed a malicious Quoting Enclave that signs *any* local attestation report with the EPID keys, obtained in Step 2 above, without first verifying it.

Step 4: Breaking Attestation Consider a malicious software that would like to masquerade as a specific enclave and prove its "authenticity" and SGX security properties via remote attestation. Given an enclave to masquerade, the malicious software first generates a *false* local attestation report with the values of MRENCLAVE and MRSIGNER corresponding to the enclave it wants to masquerade, as well as other metadata required for generating the local attestation report. It then sends this report to the malicious Quoting Enclave.

We notice here that the malicious software is unable to sign the *local* attestation report, as it doesn't have access to the appropriate signing key. However, as the malicious Quoting Enclave does not verify the report, the report is not required to be signed. Next, using the unsealed EPID keys, the malicious Quoting Enclave generates a malicious attestation quote by signing the local (false) attestation report. This malicious quote is then sent to the remote party.

Finally, the remote verifying party attempts to verify the malicious quote using Intel's Attestation Server (IAS). As the quote was indeed correctly signed by the malicious Quoting Enclave, assuming that the EPID keys used are valid and have not been revoked, Intel's attestation server will accept the malicious quote and generate a signed transcript of the response. The transcript falsely convinces the remote party that the enclave is running on a genuine Intel CPU (which is designed to provide confidentiality and integrity), while it is actually running on the malicious, non-SGX software, and does not offer any security guarantees.

5.6.3 Empirical Evaluation

In this section, we empirically demonstrate the feasibility of the attack on SGX's attestation mechanism.

Extracting EPID Keys Using the setup from Section 5.1.2, we have successfully extracted the EPID sealing keys from a genuine SGX Quoting Enclave and subsequently unsealed the machine's private EPID keys.

Signing Fake Attestation Quotes Demonstrating the ability to sign arbitrary attestation quotes, we created a local attestation report setting the MRENCLAVE field, "representing" the SHA-256 of the enclave's initial state, to be the string "*Is your enclave cheating on you?*", the MRSIGNER, "representing" the SHA-256 of the public key of the enclave writer, to be "*SGX-ray: Trustworthy Speculation*", and the report's debug flag to 0, thereby indicating that the enclave is a production enclave. We have also populated the report's body (commonly used for establishing a Diffie-Hellman key exchange with the enclave corresponding to the report) to be "*Mary had a little lamb, Little lamb, little lamb, Mary had a....*". Finally, we signed the report via the malicious Quoting Enclave using the above-described unsealed EPID signing keys, thereby producing an attestation quote.

Quote Verification Verifying the validity of the quote, we have contacted Intel's Attestation Server (IAS) and provided it with the above generated quote. As explained in [51, 149], the attestation server will only approve the quote if it can verify that the quote's EPID signature is correct. Since we have correctly extracted a non-revoked EPID private signing key, the attestation server deemed the forged quote as correct and replied with "isvEnclave-QuoteStatus:OK", signing its response with Intel's private key and accompanied it with the appropriate certificate chain leading to Intel's CA certificate.

5.7 Conclusions

In this chapter, I show that the memory protection of SGX enclaves does not protect against a Meltdown-like attack. I build a generic read primitive that allows us to easily read the memory of victim enclaves, including pages that are not accessible to the enclaves themselves. Thus, the attack breaks all of the confidentiality guarantees of SGX. We show how the read primitives can be used to read secrets from an enclave, with a specific example of retrieving the seal key from the Intel Quoting Enclave. Retrieving the seal key of an enclave allows us to read and modify the persistent storage of the enclave. Thus, the attack breaks the integrity guarantees of the sealing mechanism. The seal key of the Quoting Enclave gives us access to the host's attestation key. With access to this key, I demonstrate that we can sign arbitrary attestation quotes, eroding the trust in the SGX ecosystem.

The Foreshadow attack exposes the fragility of the SGX ecosystem, where a single vulnerability can result in cascading compromises that erode the security and trust properties of SGX. Intel has deployed mitigations for the specific vulnerability [121]. Zooming out of SGX-specific attacks, speculative execution attacks are proved to have devastating security implications [56]. In the next chapter I explore a fundamental approach for redesigning modern processors to defeat these types of attacks.

CHAPTER VI

NDA: Preventing Speculative Execution Attacks at Their Source

6.1 Introduction

Speculative execution attacks [173, 154, 124, 269, 282, 120, 256, 272, 55, 241, 153, 242, 176, 156, 43, 59, 177] exploit micro-architectural behavior and side channels to exfiltrate sensitive information from a system. Unlike classical software exploits that modify and observe only architectural state (such as registers and memory), speculative execution attacks have demonstrated that attackers can retrieve secrets by controlling and observing micro-architectural state (e.g., the cache) during speculative wrong-path execution.

Speculative execution attacks can be classified into two main categories. One class (e.g., Spectre [154], Spectre 1.1 [153], and others [242, 176, 43, 156, 59]) allows malicious code to mis-steer a victim program's control flow (e.g., by carefully mis-training branch predictors) to execute specific instructions on the speculative wrong path. Although wrong-path instructions are ultimately squashed (with no effect on architectural state), the victim program is coerced into leaking its own memory contents through a micro-architectural channel. For instance, Chen et al. [59] show how control-flow in an SGX secure enclave [183] can be steered to leak its own protected memory. I classify these attacks as *control-steering* attacks (Figure 6.1a).

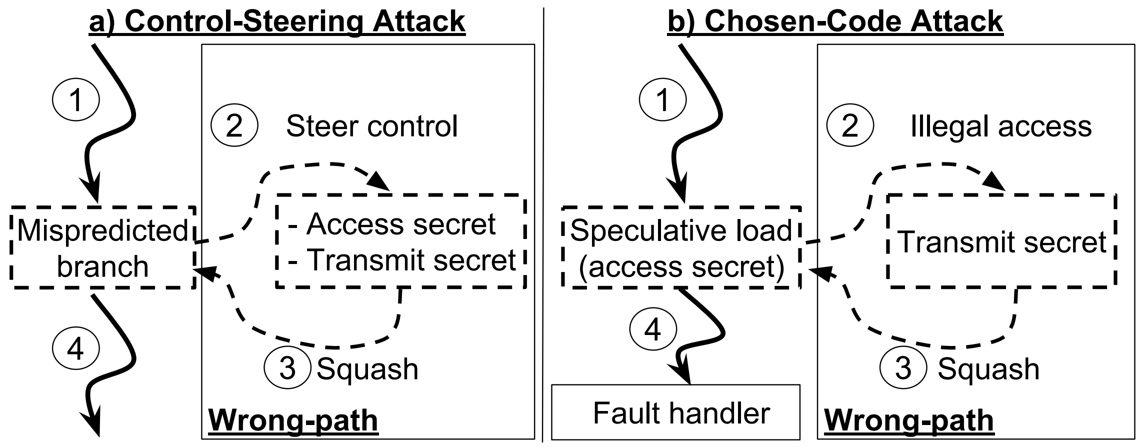


Figure 6.1: Control-steering vs. chosen-code attacks. In control-steering, the attacker steers control flow in existing victim code, inducing unwanted access to the victim's memory space. In chosen-code, the attacker generates code that accesses privileged data or data that belongs to another context.

Another class of attacks [173, 269, 282, 120, 256, 272, 55, 241] enables unprivileged attacker code to access privileged memory that is temporarily exposed during wrong-path execution. For instance, Meltdown [173] allows reading kernel memory; Foreshadow [269, 282, 120] allows reading hypervisor, OS, SMM, or SGX memory; and LazyFP [256] allows reading AES keys from AVX registers used by another process. MDS attacks [272, 55, 241] allow reading recently accessed memory belonging to other processes. Since the attacker generates the code, they can select arbitrary instruction sequences in both correct-path and wrong-path execution. I classify these attacks as *chosen-code* attacks (Figure 6.1b). These two classes of attacks are fundamentally different and therefore require different approaches for mitigation.

Existing software defenses against speculative execution attacks work by modifying a program's source code to block attack-specific mechanisms. Current software defenses for control-steering attacks—such as Retpoline [134, 96], IBPB [141], and improved `lfence` [73] instructions—focus on preventing the attacker from steering the execution of victim code. Unfortunately, these defenses are not immediately applicable to existing binaries. Specifically, software mitigations against chosen-code attacks involve modifying the OS, hypervisor, and SMM code [175, 188, 101, 120]. A recent study by Google [182] discusses why software approaches aimed at mitigating timing channels by manipulating timers are insufficient. The authors show that any optimizations performed by micro-architecture, no matter how negligible, can become observable using an amplification technique. Even if code modifications are made, these defenses can be bypassed. For instance, attackers can redirect control flow to evade fence instructions (e.g., by mis-training the branch target buffer (BTB) [124, 154] or the return stack buffer (RSB) [176, 156, 153]).

Hardware defenses, on the other hand, have the potential to obviate the need to modify existing software [288, 150, 220, 152, 228, 141, 260, 237]. The first disclosed speculative execution attacks [173, 154, 124] use caches as a covert channel to leak data from wrong-path execution. Consequently, initial hardware defenses—such as InvisiSpec [288], SafeSpec [150], and others [220, 152, 228, 260, 237]—seek to prevent wrong-path execution from leaving secrets in the cache that can later be recovered. Taram et al. [260] suggests a hardware modification to automatically insert `lfence` micro-ops where needed. However, the authors claim mainly to address Spectre v1 attacks that use the data cache as a covert channel.

While these techniques are effective, a recent study [56] noted that closing only the cache covert channel is insufficient to stop speculative execution attacks, since the cache is only one of many potential covert channels. Netspectre [242] and SMoTher Spectre [43] have already shown that secrets can be transmitted via the FPU or via port contention [16]. In Section 6.3, I further show how to transmit secrets via the BTB.

Rather than isolating predictive structures [141] or sealing individual covert channels [288, 150, 237]—a ceaseless arms race—I instead seek to close off speculative execution attacks at their source. In this chapter I propose to treat potentially wrong-path values as secret and prevent these secret values from propagating through the micro-architecture. The key observation is that speculative execution attacks require a *chain of dependent wrong-path instructions* to access and transmit data into a covert channel. By preventing potentially wrong-path values from propagating, we break these dependency chains, thwarting the code sequences required to mount attacks.

I propose *NDA*—Non-speculative Data Access—a technique to restrict speculative data propagation in out-of-order (OoO) processors. *NDA* only allows instruction outputs to flow to dependents if the source instruction is considered *safe*. *NDA* restricts data propagation by preventing tag broadcast for unsafe instructions, delaying wake-up of their dependants in the issue queue until the source instruction becomes safe.

I present a taxonomy of the building blocks of speculative execution attacks, show how each class of attack depends upon data propagation in wrong-path execution, and demonstrate how we can define *safe* vs. *unsafe* to prevent the data flow required by the attack. By composing various restrictions on when an instruction becomes safe, I create a design space of *NDA* variants. The variants differ in (1) the constraints they place on the dynamic execution schedule (and therefore, performance), (2) the locations from which secret data might be extracted (e.g., whether general purpose registers are protected), and (3) the kind of speculation attacks they prevent (e.g., control-steering vs. chosen-code).

NDA defeats all 25 documented [56, 43, 272, 55, 241] speculative execution attacks without the need to modify any existing code. Importantly, however, *NDA* does not preclude all speculation or OoO execution. For example, one *NDA* policy treats all instructions after an unresolved branch as unsafe. These instructions may still execute speculatively OoO, but they are restricted from propagating their output to dependents until all preceding branches resolve. As my evaluation demonstrates, despite delayed wake-ups, the vast majority of the performance gap between in-order (the only other model known to eliminate all known speculative execution attacks) and unconstrained OoO execution is recovered.

I simulate *NDA* designs on the SPEC CPU 2017 benchmark suite and compare its performance to InvisiSpec [288] on the same setup. InvisiSpec blocks data-cache-based attacks and introduces 7.6-32.7% overhead in the evaluation setup. In comparison, *NDA* blocks *all* covert channels. I show that an *NDA* policy that mitigates control-steering vulnerabilities, which are fundamental to unconstrained OoO execution, slows execution by only 10.7% and is 4.8× faster than in-order. If we also preclude Meltdown-like hardware implementation flaws, *NDA*'s strictest policy slows down execution by 125% compared to an insecure OoO processor and is 2.4× faster than in-order execution.

In short, in this chapter I make the following contributions:

- I introduce a speculative-execution-attack taxonomy based on how attacks induce wrong-path execution.
- I design *NDA*, a new technique to control speculative data propagation in out-of-order processors to defeat speculative execution attacks. *NDA* offers multiple variants with differing security/performance tradeoffs.
- I evaluate six *NDA* variants on SPEC 2017 and show they are effective and efficient.

6.2 Background

Data Propagation in OoO Processors. Figure 6.2 illustrates conceptual steps in an instruction's life-cycle in a modern OoO processor. Upon dispatch into the reorder buffer (ROB), an instruction is not ready to execute until all of its source operands—coming from instructions S1 and S2 in Figure 6.2—are ready (step 1). Once all source operands are ready, the instruction issues and enters the execution pipeline (step 2). When execution completes (step 3), the instruction wakes its dependents (D1-D5) by broadcasting a tag corresponding to its destination physical register to waiting instructions (step 4), marking those instructions ready.

The essence of the *NDA* technique is to *delay tag broadcast*, i.e., transition from step 3 to step 4. Rather than waking dependent instructions when their input operands become *ready*, *NDA* wakes them up when their input operands are *safe*. We expand on this basic concept in Section 6.5.

Speculative Execution Attacks. Speculative execution attacks exploit side-effects of wrong-path execution, which are typically left undefined by processor vendors. While the contents of architectural registers and memory are guaranteed to reflect precise state of only committed instructions, wrong-path execution affects micro-architectural structures. For instance, a wrong-path cache access may allocate new lines or modify the cache replacement order; these changes are not reverted when wrong-path instructions are squashed. A variety of other micro-architectural structures are also not reverted during squash, for example, branch direction predictors (e.g., pattern history table), pre-decoded micro-op/trace caches, memory dependence predictors, prefetchers, TLBs, fine-grain power management state (e.g., for FPU/AVX units), and performance counters. State changes in these micro-architectural structures can create side channels, where the state can be inferred, for example, based on timing particular execution sequences. We refer to a side channel that is used to intentionally transmit data as a *covert channel*. Attackers can use *wrong-path* execution to transmit data, via a covert channel, that is later inferred by *correct-path* execution and hence leaks that data into architectural state.

6.3 Problem Analysis

We next classify speculative execution attacks based on three fundamental attack phases that exist in all known attacks. We then describe the existing mitigation techniques, how they block the attacks, and their shortcomings. Lastly, to demonstrate that closing specific side channels is insufficient, we show an attack via a new covert channel—the BTB.

6.3.1 Classifying Attacks

Attack Phases. All speculative execution attacks of which we are aware comprise three key phases—*access, transmit,* and *recover*—shown in Figure 6.3. In the *Access Phase* (①), secret data is loaded into a temporary register. During the *Transmit Phase* (②) the secret data is covertly transmitted using micro-architectural side effects that are not reverted when wrong-path instructions are squashed. Finally, in the *Recover Phase* (③), the transmitted secret is recovered to non-speculative state (e.g., by observing the memory access latency). Whereas the instructions involved in phases ① and ② are speculatively

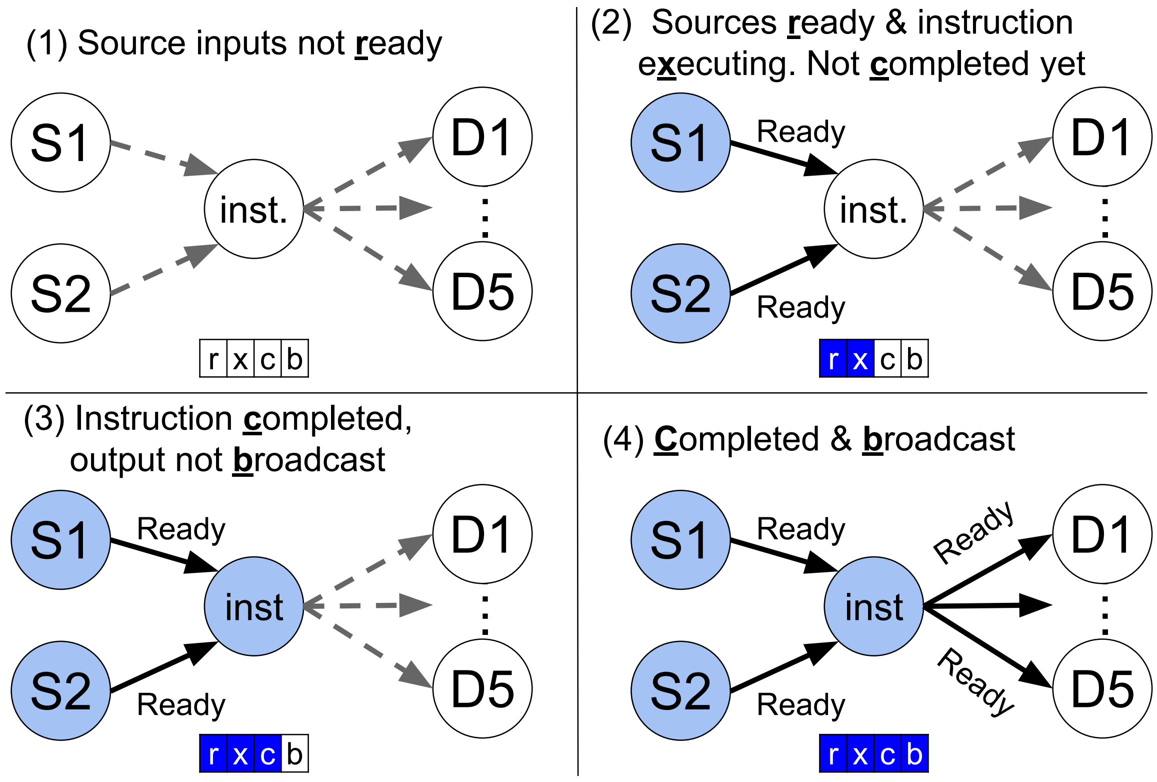

(1) Source inputs not ready
S1
S2
inst.
D1
D5
r x c b
(2) Sources ready & instruction executing. Not completed yet
S1
S2
Ready
Ready
inst.
D1
D5
r x c b
(3) Instruction completed, output not broadcast
S1
S2
Ready
Ready
inst
D1
D5
r x c b
(4) Completed & broadcast
S1
S2
Ready
Ready
inst
Ready
Ready
D1
D5
r x c b

Figure 6.2: Life-cycle of instructions in OoO processors. Even after the instruction has completed execution (3), the dependant instructions (D1-D5) will not be able to access the output until it is broadcast (4).

executed and eventually squashed, the phase ③ results are committed to the architectural state. Wrong-path execution is essential to these attacks, as it evades software and hardware protection mechanisms that prevent the secret data from leaking through architectural state.

Control-Steering and Chosen-Code Attacks. We classify attacks based on their methodology for performing the *Access Phase* (①) and the *Transmit Phase* (②). We divide attacks based on their *Access Phase* into two categories, which correspond to different attacker threat models. We further subdivide these two attack classes according to the covert channel exploited in the *Transmit Phase*. table 6.1 illustrates this taxonomy for currently-known attacks.

In *control-steering* attacks, the attacker subverts a victim program's control flow to speculatively execute instructions that, as a side-effect, leak data into a covert channel. This attack class leaks data to which the victim application has hardware access privileges, but

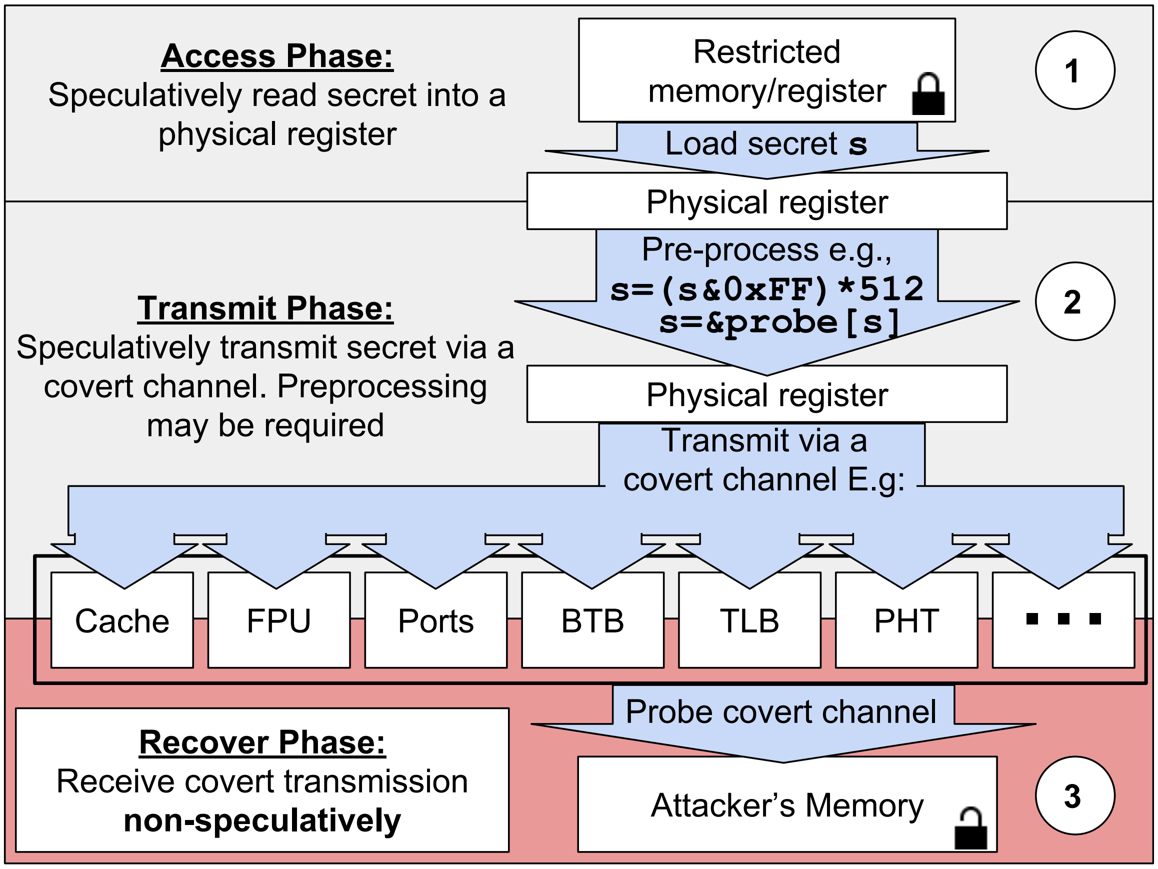

Access Phase:
Speculatively read secret into a physical register
Restricted memory/register
1
Load secret s
Physical register
Transmit Phase:
Speculatively transmit secret via a covert channel. Preprocessing may be required
Pre-process e.g.,
s=(s&0xFF)*512
s=&probe[s]
2
Physical register
Transmit via a covert channel E.g:
Cache
FPU
Ports
BTB
TLB
PHT
...
Probe covert channel
Recover Phase:
Receive covert transmission non-speculatively
Attacker's Memory
3

Figure 6.3: Three phases of speculative execution attacks. Prior defenses focus mostly on the cache covert channel, failing to prevent leaks through other channels such as the FPU [242], the BTB (Section 6.3), and others.

are intended to be secret and might be protected (e.g., by permission or bounds checks) in software. For example, SGXPectre [59] coerces a secure SGX [183] enclave to access and leak its encrypted memory. We illustrate control-steering attacks in Figure 6.1a.

Unlike a classical security vulnerability, wherein the attacker directly hijacks the program counter (e.g., a stack-smashing attack that overwrites a return address), speculative control-steering attacks only misdirect wrong-path execution, for example, by mis-training branch predictors to direct instruction fetch to an attacker-selected target. Hence, they leave no trace in the committed instruction sequence, but still leak data into a covert channel. Several approaches that use control-steering have been demonstrated [154, 176, 156, 153].

In control-steering attacks, the attacker does not typically introduce new instructions

```
for (i=0; i < 256; i++) // init channel
  clflush(probeArray[i*512]);
// Phase ① - access secret data:
// The attacker mis-trains the branch:
if (x < array_size) { // predicted taken
  // wrong-path, x >= array_size
  secret = array[x];
// Phase ② - covertly transmit secret:
  t = probeArray[secret * 512];
}
// ... somewhere else in attacker's code
// Phase ③ - recover secret:
for (guess = 0; guess < 256; guess++) {
  addr = &probeArray[guess*512];
  t1 = rdtscp(); // read timer
  temp = *addr;  // access probing array
  t2 = rdtscp(); // read timer
  if (t2-t1 <= CACHE_HIT_THRESHOLD)
    results[guess] += 1;
}
```

Listing VI.1: Exfiltrating secret data using Spectre v1 *control-steering* and the cache covert channel.

into the victim binary, rather, the attacker composes a series of gadgets from existing code, similar to Return Oriented Programming (ROP [247, 234, 52]).

By contrast, in *chosen-code* attacks—the second category based on the *Access Phase*—we consider an adversary who can generate and execute arbitrary code sequences to mount the attack. Such an adversary already has access to its own registers and memory; these attacks instead seek to circumvent *hardware* protections that preclude the attacker from accessing secret data in correct-path code. For instance, Meltdown [173] accesses kernel memory; Foreshadow [269, 282, 120] accesses SGX and hypervisor memory; and LazyFP [256] accesses AVX registers used by another process. These attacks exploit implementation flaws in the relative timing of hardware protection checks and data flow between wrong-path instructions—the secret data propagates among instructions and can be leaked into a covert channel before protection checks squash the wrong-path execution. We show chosen-code attacks in Figure 6.1b.

Sample Attack Code. Listing VI.1 illustrates these phases for the Spectre v1 [154] bounds check bypass attack [124], which is a control-steering attack. In this attack, the

```
1 // Phase ① - access secret:
2 secret = *kernel_addr; // Faulting load
3 // Phase ② - covertly transmit secret:
4 // Executed in wrong-path
5 // before fault is fired:
6 t = probeArray[secret * 512];
7 // Phase ③ - recover secret:
8 // see Listing VI.1
```

Listing VI.2: Exfiltrating secret data using the Meltdown *chosen-code* attack and a cache side-channel.

victim code includes instructions that access `array` at a given index `x` (Line 7). Before accessing `array`, the victim code performs a bounds check on `x` (Line 5). To circumvent the bounds check, the attacker mis-trains the branch direction-predictor by invoking the victim code repeatedly with a valid `x`.

To mount the attack, the attacker now calls the victim code with an illegal value of `x`. The attacker chooses `x` such that `array[x]` will refer to a location in the victim's memory containing a secret. The direction predictor mis-predicts the branch on Line 5 as taken, executing Lines 7–7 on the wrong path. During wrong-path execution, the code *accesses* (①) the secret on Line 7. It then *transmits* (②) the secret (still in wrong-path) on Line 7. Later, in correct-path execution, the attacker executes Lines 13–20 to *recover* (③) the secret from the cache side-channel. The timing for each access to `probeArray` on Line 16 will vary based on whether or not the corresponding cache line was loaded on Line 7. In the evaluation, we illustrate the difference in access timing (blue squares in Figure 6.4), which reveals the secret data.

Listing VI.2 depicts an example of a chosen-code attack—a simplified Meltdown exploit. Whereas the illegal load on Line 2 will eventually fault, the instruction on Line 6—which executes on the wrong path—will leave evidence in the cache from which the attacker can recover the secret. The *recover* phase is identical to that in Listing VI.1. To avoid trapping into the fault handler, the attacker may use control-steering techniques to ensure the faulting `load` executes under a mis-predicted branch [173]. Nevertheless, we classify the attack as chosen-code since the attacker controls the executed binary.

Phase ①	Phase ② Attack	d-cache	i-cache	FPU	Ports	BTB
Control steering	Spectre v1 [154]	✓				★
	Spectre v1.1 [153]	✓				
	Spectre v2 [154]	✓	✓			
	Ret2spec [156,176]	✓				
	NetSpectre [242]			✓		
	SMoTher Spectre [43]				✓	
	SSB (Spectre v4) [124]	✓				
	<future attacks>					
Chosen code	Meltdown (v3 / v3a) [172]	✓				
	LazyFP[256]	✓				
	Foreshadow (L1TF) [269,282]	✓				
	MDS attacks [55,241,272]	✓				
	<future attacks>					

✓ - demonstrated in prior work; ★ - demonstrated in this thesis

▨ - d-cache-based attacks are defeated by prior work [31,48,53,69]

Table 6.1: Taxonomy of attacks based on secret data access method ① and covert channel ②. *NDA* blocks all existing attacks regardless of the covert channel they use. Most common attacks use the d-cache side channel to exfiltrate secret data. All currently known chosen-code attacks use `loads` and `load`-like operations. Future attacks may use other instructions or other covert transmission channels.

6.3.2 Limitations of Existing Defenses

Current Mitigations. Hardware defenses mitigating control-steering attacks try to prevent the attacker from mis-training branch predictors (IBRS and STIBP [141]) or use a barrier instruction to prevent speculation after a branch or context switch (`lfence`/IBPB [141]). Unfortunately, recent attacks [153, 176, 156] reveal techniques to overcome these mitgations. SSBD [141, 27] disables Speculative Store Bypass (SSB, explained in Section 6.4.1) to prevent attackers from reading data that was overwritten [124, 295]. However, SSBD only blocks Spectre v4. and introduces up to 8% overhead [142].

Software defenses, such as Retpoline [96] and RSB stuffing [134], protect `call` and `ret` instructions from mis-steering. Other compiler approaches [210, 98] create a data dependency between a branch condition and code that follows the branch, disabling speculation. However, these compiler approaches can only defeat Spectre v1 [154] attacks. A recent study suggested a compiler modification that also blocks Spectre v2 attacks [249]. Unfortunately, this approach can only defeat cache-based attacks with 68-247% overhead.

Chosen-code attacks are mitigated by preventing speculative `load`s from accessing restricted memory. For instance, Kernel Address Space Layout Randomization (KASLR [101]) and Kernel Page Table Isolation (KPTI [175, 188]) prevent Meltdown attacks from reading privileged kernel memory. KASLR [101] randomizes the kernel address space similar to how ASLR is used to protect user-space processes. KPTI manages separate page tables for the kernel and user-space processes, preventing user code from issuing even illegal loads to kernel memory. KPTI swaps page tables on every transfer between CPU privilege levels. Mitigating Foreshadow [269, 282, 120] requires modifications to the OS, hypervisor, and SMM code, such as modifying page-table management, altering virtual machine scheduling, and adding L1 cache flushes when switching security domains [282, 120].

Unfortunately, all these defense mechanisms block only specific exploit techniques. Therefore, one must deploy a myriad software and hardware defenses to be resilient against *all* control-steering and chosen-code attacks.

Recent work suggests preventing both control-steering and chosen-code attacks by blocking the cache side channel [150, 288, 220, 237], thus interdicting the *transmit* phase. However, given the abundant supply of covert channels (see Figure 6.3), defeating speculative attacks by closing each channel individually is challenging. Exploits have already been demonstrated for other channels. Netspectre [242] demonstrated that the power state of the FPU is a viable speculative covert channel. SMoTher Spectre [43] showed how to transmit data via port contention [16]. We next show an attack via the BTB.

The BTB Covert Channel. We demonstrate a new covert channel that can be exploited

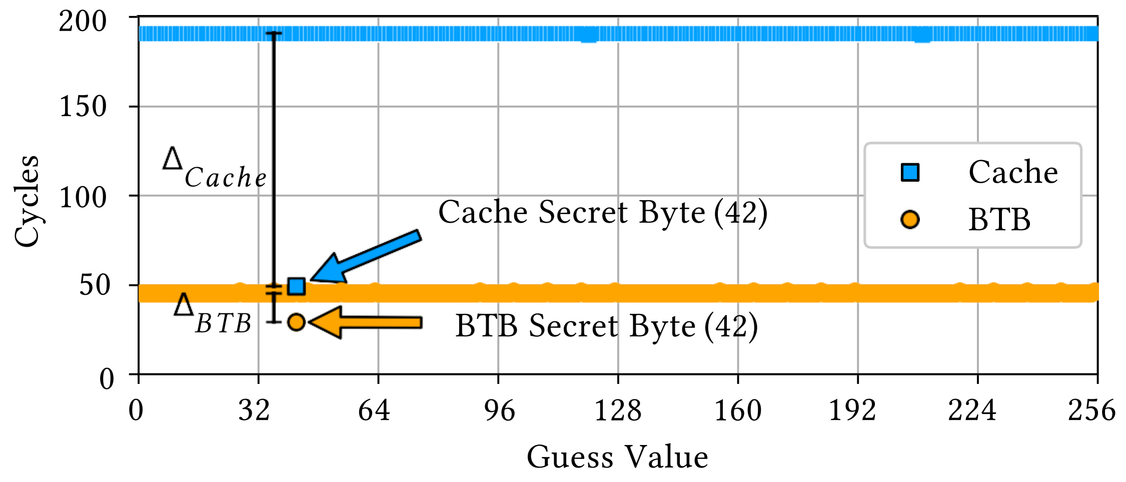


Figure 6.4: Spectre v1, using either the cache (blue squares) or the BTB (orange circles) as a covert channel. For the cache channel, only the correct guess produces a cache hit, creating the cycle difference Δ_{Cache}. For the BTB channel, only the correct guess successfully predicts the jump target, creating the cycle difference Δ_{BTB}.

even when the cache covert channel is not available—the BTB. The BTB stores a mapping between branch instructions' addresses and the associated target addresses. For example, a `call` instruction located at address `A` to a function located at address `B` installs the mapping `A => B` in the BTB. The next time the processor fetches the `call` instruction at address `A`, the processor's front-end will speculatively redirect fetch to address `B`.

If the BTB predicts correctly (Figure 6.5a), the speculatively-fetched instructions are eventually retired. However, if the prediction is wrong, the processor will squash the wrong-path execution, starting at the mispredicted instruction at address `B`, before executing the correct path. This recovery process is illustrated in Figure 6.5b. In the experiments on the *gem5* [44] simulator, we observe that it takes ~16 cycles for the BTB miss to resolve, wrong-path execution to be squashed, and execution to resume at the correct target ($1+2$ in Figure 6.5b). Crucially, updates to the BTB during speculation are not reverted by the squash, making it an effective covert channel. Note that (as with caches) in the absence of security concerns, filling the BTB (and updating its replacement policy) during speculation may be advantageous to avoid future BTB misses.

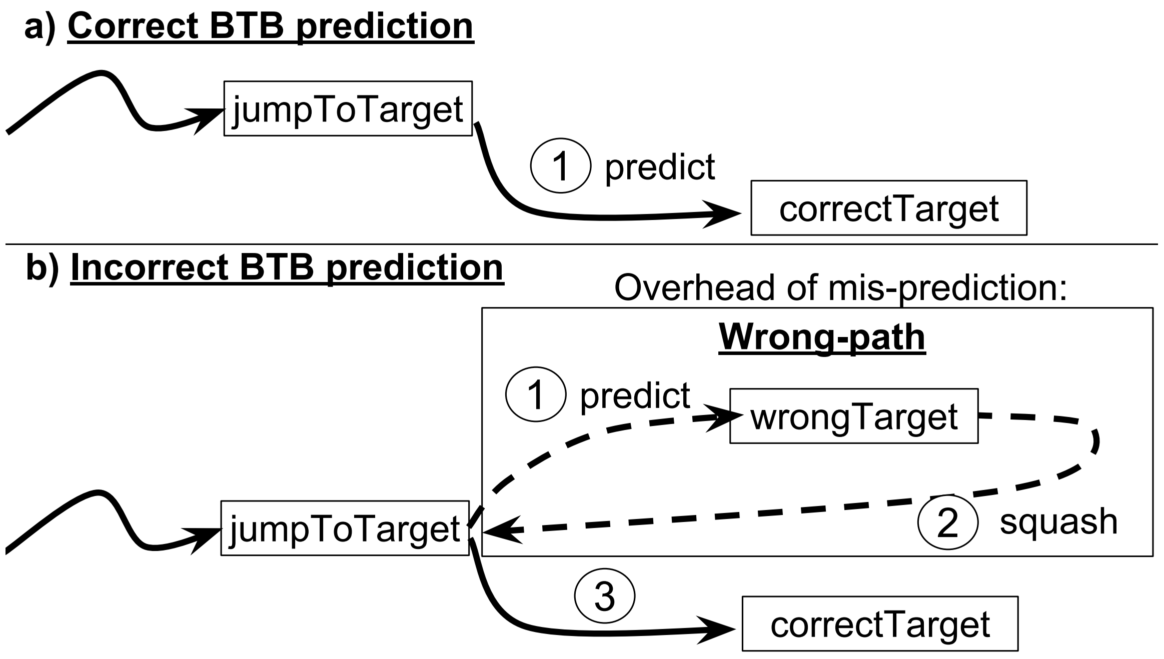


Figure 6.5: The BTB covert channel. The attacker can observe if the BTB prediction was correct by measuring execution time.

To demonstrate the BTB covert channel, we construct a variant of Spectre v1 [154] that leaks a secret byte through a speculative BTB update, as illustrated in Listing VI.3. To leak a single byte, the covert channel comprises 256 distinct functions (`targets` in Line 2). During both the *Transmit Phase* and *Recover Phase*, we invoke `targets` only from a single call site, `jumpToTarget` (Line 6), ensuring that BTB entries mapping to `targets` all originate from the same PC and therefore conflict in the BTB.

When the branch on Line 10 is mispredicted, the attacker can *access* any value from the process' address space, depending on the value of `x`. The attacker then *transmits* the secret by speculatively calling `jumpToTarget` with the secret value in Line 13. If the speculation window is large enough, the processor updates the BTB entry for the `call` instruction in Line 6 based on `secret`.

The *access* phase must be repeated for every guess (Line 19) since the *recover* phase is destructive: The execution of Line 21 alters the contents of the BTB to point to `targets[guess]`. To confirm that the BTB acts as the covert channel in the attack, it is important to validate that execution time differences do *not* arise from i-cache or d-cache hit or miss latency; no change to the cache hierarchy during the attack may depend upon the secret value. To validate the attack, we ensure the `targets` array in Line 2 and all 256 target functions are cached during access, transmission, and recovery.

We report the effectiveness of the BTB covert channel on *gem5* via the orange circles in Figure 6.4. During the *Recover Phase*, in lines 17-24, all the *wrong* guesses will incur the 16-cycle prediction and squashing delay, as depicted in Figure 6.5b. The *correct* guess will execute faster, as depicted in Figure 6.5a.

The BTB covert channel is one of many potential machine-specific transmission channels. We use the BTB channel PoC to demonstrate that *NDA* is agnostic to any specific transmission channel (Section 6.6).

6.4 Threat Models

NDA design variants address four different threat models, which vary in the locations from which secret data are stolen and whether the attacker may mount control-steering or chosen-code attacks. *NDA*'s goal is to eliminate side-channels created in *wrong-path* execution. Correct-path side channels have been studied in prior work [228, 289, 290].

All threat models are agnostic to the covert channel used in the attacks. For control-steering attacks, we consider two threat models, based on where secrets reside. The first model considers attacks against secrets stored in memory or special registers, as is the case for all currently-known control-steering attacks. The second control-steering threat model additionally considers hypothetical attacks that leak secrets residing in general-purpose registers (GPRs). In the third threat model, for chosen-code attacks, we consider only threats against secrets in privileged memory and registers, since chosen-code attacks presuppose attacker-controlled GPRs. Lastly, the fourth threat model comprises the union of these threats, considering both control-steering and chosen-code attacks for secrets in memory, special-registers, and GPRs.

```
// array of 256 unique target functions
void (*targets[256])(void);
// all jumps are from the same location,
// hence the same BTB entry is consulted
void jumpToTarget(int index)
 { targets[index](); }
void victim_function(x) {
  // Phase ① - access secret data:
  // The attacker mis-trains the branch:
  if (x < array_size) {    // predicted taken
    secret = array[x];     // wrong path
  // Phase ② - covertly transmit secret:
    jumpToTarget(secret); // updates BTB
} }
// ... somewhere else in attacker's code
// Phase ③ - recover secret:
for (guess = 0; guess < 256; guess++) {
  // Induce victim to leak secret value
  victim_function(x);
  t1 = rdtscp();           // read timer
  jumpToTarget(guess);     // BTB prediction
  t2 = rdtscp();           // read timer
  if (t2-t1 <= CORRECT_PATH_THRESHOLD)
    results[guess] += 1;
}
```

Listing VI.3: Exfiltrating secret data using the Spectre v1 *control-steering* attack and the BTB side-channel.

6.4.1 Leaking Memory via Control-Steering

The first step of all known control-steering attacks is to steer wrong-path execution into code accessing a secret in memory or manipulate execution timing to cause a load to observe a stale value. We assume the attacker can steer execution at any branch instruction and manipulate the execution timing of all instructions. Branch instructions include all variants of `jmp`, `call`, and `ret`.

We do not consider *phantom branches*, where the BTB is mis-trained to steer control flow from a program counter value that does not correspond to a branch. The dispatch stage stalls micro-ops whose opcode is unknown. Hence, if the BTB predicts a branch where there is none, dispatch will stall at the phantom branch until its opcode is obtained, which will resolve the misprediction and cause any younger fetched instructions to be discarded before they enter the OoO back-end. Wrong-path instructions that are squashed before dispatch are not a threat.

We also do not consider potentially faulting instructions as steering points in control-steering attacks. Whereas a fault can result in wrong-path execution, we consider attacks based on faulting instructions (e.g., Meltdown, Foreshadow, LazyFP, MDS, etc.) as part of the threat model for chosen-code attacks.

Speculative Store Bypass. Also known as SSB, or Spectre variant 4 [124], this attack performs the *Access Phase* (① in Figure 6.3) by having a malicious speculative load bypass a store whose address is still unresolved. The malicious load then speculatively yields stale (secret) data. Although this attack may not necessarily require misdirected control flow in the *Access Phase*, we consider it a special case of control-steering, since the attacker leverages an existing code snippet. If the attacker could choose the code, they could read the stale data without the need to exploit the speculative store-bypass.

6.4.2 Leaking GPRs via Control-Steering

All currently-known control-steering attacks extract secrets residing in memory. Nevertheless, we recognize that future attacks might extract secrets residing in the victim's GPRs. So, the second threat model considers the attacker of Section 6.4.1 that steers the victim's control flow to leak GPR contents.

In this scenario, the steered victim's code already possesses the secret in a GPR. At this point, the access phase of the control-steering attack (① in Figure 6.3) has already (possibly unintentionally) been done by the victim. We therefore focus on hindering the attacker from performing the second phase (② in Figure 6.3)—transmitting the GPR-resident secret. All known attacks require data flow between micro-ops during the transmit phase to preprocess the secret (e.g., calculate an offset relative to a base address) before it can be leaked.

We do not prevent an attack that leaks a secret using only a single speculative micro-op. In principle, it may be possible to covertly transmit GPR-based secrets using a single micro-op. For instance, if a GPR contains a secret value that corresponds to a valid virtual memory address, the attacker can speculatively issue a `load` that will fetch this address into the cache hierarchy, thus performing the transmit phase in a single micro-op. However, such a scenario would require (a) a secret value that forms a valid memory address, and (b) victim code that voluntarily loads the secret into a GPR shortly before the vulnerable steering point. No known attacks (cf. table 6.1) exploit this behavior.

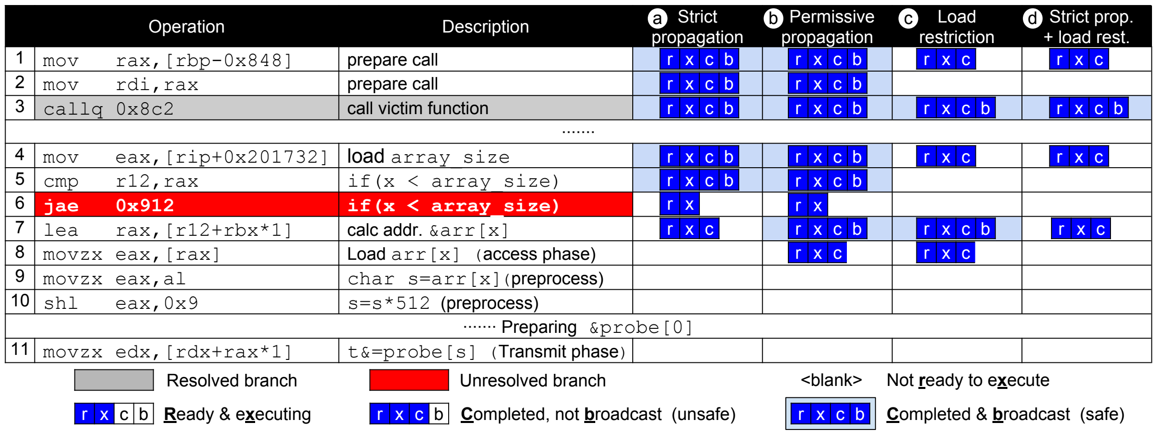
6.4.3 Leaking Memory with Chosen-Code

For chosen-code attacks, we consider attackers that attempt to access secrets residing in memory. Specifically, we consider an attacker who can influence code generation to control both correct-path and wrong-path execution. We treat read operations from special-purpose registers, such as AVX (as abused in LazyFP [256]) and Model Specific Registers (MSRs, in Meltdown variant 3a [124]) like memory accesses in crafting our defense—the special instructions (e.g., `rdmsr`) used to access these registers are treated like loads in our solution. In chosen-code attacks, the attacker already controls their own GPRs and we therefore do not consider the contents of any GPR to be secret.

Instructions are guaranteed to be correct-path when they retire. At retirement, the head of the ROB satisfies *hardware* permission and memory-ordering checks. Ergo, retired instructions cannot leak secrets accessed from the *wrong-path*.

6.4.4 Combining the Threat Models

Finally, we consider *NDA*'s most conservative threat model—a combination of all threats outlined above. We suppose an attacker that conducts both (a) control-steering attacks to extract secrets from the victim's memory and GPRs, *and* (b) chosen-code attacks to access privileged memory and special registers. This combined threat model is similar to the practical approach taken by Windows and Linux, which deploy mitigations for both classes of attacks [175, 188, 141, 190, 267].



	Operation	Description	(a) Strict propagation	(b) Permissive propagation	(c) Load restriction	(d) Strict prop. + load rest.
1	`mov rax,[rbp-0x848]`	prepare call	r x c b	r x c b	r x c	r x c
2	`mov rdi,rax`	prepare call	r x c b	r x c b		
3	`callq 0x8c2`	call victim function	r x c b	r x c b	r x c b	r x c b
		....				
4	`mov eax,[rip+0x201732]`	load `array_size`	r x c b	r x c b	r x c	r x c
5	`cmp r12,rax`	`if(x < array_size)`	r x c b	r x c b		
6	`jae 0x912`	`if(x < array_size)`	r x	r x		
7	`lea rax,[r12+rbx*1]`	calc addr. `&arr[x]`	r x c	r x c b	r x c b	r x c
8	`movzx eax,[rax]`	Load `arr[x]` (access phase)		r x c	r x c	
9	`movzx eax,al`	`char s=arr[x]`(preprocess)				
10	`shl eax,0x9`	`s=s*512` (preprocess)				
		.... Preparing `&probe[0]`				
11	`movzx edx,[rdx+rax*1]`	`t&=probe[s]` (Transmit phase)				

▨ Resolved branch 🟥 Unresolved branch <blank> Not **r**eady to e**x**ecute

r x c b **R**eady & e**x**ecuting r x c b **C**ompleted, not **b**roadcast (unsafe) r x c b **C**ompleted & **b**roadcast (safe)

Figure 6.6: An ROB snapshot during the execution of Spectre v1 (Listing VI.1), with different *NDA* policies. The branch (call) at line 3 has been resolved, therefore the load in line 4 is *safe* under strict and permissive propagation and can broadcast (wake-up dependants). Under the load restriction policy, the instructions in lines 1,4, and 8 can be executed but are not *safe* until retirement. Therefore, line 2 cannot be issued to execute.

6.5 Design

NDA's main design goal is to mitigate both control-steering and chosen-code attacks while reaping the benefits of OoO speculative execution as much as possible. We next discuss different variants of *NDA*, which provide different policies for speculative data propagation depending on the threat model. Different *NDA* data propagation policies offer different security guarantees and have corresponding performance implications. We build *NDA* upon a baseline physical register-based OoO micro-architecture [294].

The key insight behind *NDA*'s design is that speculative instructions (either in the *correct* or the *wrong*-path) can safely execute without leaking secrets as long as their inputs are results of *safe* instructions. We define instructions as *safe* with respect to the threat models such that wrong-path execution can not leak any more information into a side channel than a correct-path instruction. Consequently, we eliminate the gap between *speculative* side channel attacks and *non-speculative* side channels, which security-conscious programmers already must reason about. The different *NDA* policies, listed in rows 1-6 of table 6.2, define which instructions are considered *safe* such that they may wake-up dependent instructions (allow instructions to advance from step 3 to step 4 in Figure 6.2).

To mitigate control-steering attacks, *NDA* restricts data propagation following an unresolved branch or unresolved store address (rows 1-4 in table 6.2), depending on where secrets reside and if store-bypass (SSB) is a threat. We consider any instruction following a predicted branch as *unsafe* until the branch target and direction are resolved. We also consider loads that follow a store with an unresolved address as *unsafe* (see Bypass Restriction in Section 6.5.2). To mitigate chosen-code attacks, *NDA* introduces a *propagate-on-retire* mechanism (row 5), which defeats all 11 documented chosen-code attack variants [56, 272, 55, 241] and similar future exploits that rely on speculative loads. In this policy, the value returned by *any* load instruction (or other instructions that read sensitive registers, such as `rdmsr` on x86) are considered *unsafe* until the load is ready to retire. Finally, the two mechanisms can be combined to defend against both classes of attacks (row 6).

6.5.1 Strict Data Propagation

NDA addresses control-steering attacks by defining unresolved branches and unresolved stores—for which predictions may be incorrect—as the borders between safe and unsafe speculation. When a branch micro-op enters the ROB, it is *unresolved*. Since the fetch unit predicts which instructions to fetch following the branch (via the BTB, RSB, etc.), subsequently dispatched micro-ops may be wrong-path. Similarly, when a store micro-op enters the ROB, it is *unresolved* until its address is calculated. If a store's address has not been calculated, loads that follow the store may erroneously access stale data if their addresses overlap. We consider two variants of data propagation restrictions with regards to control-steering attacks: strict and permissive. Both variants leverage a *Bypass Restriction* mechanism to defeat SSB attacks. We now describe strict propagation and then explain permissive propagation and bypass restriction in Section 6.5.2.

Strict Propagation (rows 3-4 in table 6.2) defends against threat models where secrets may reside in memory, special registers, and GPRs (i.e., the union of the threats described in §6.4.1 and §6.4.2). Under this policy, *NDA* marks *all* micro-ops dispatched after an unresolved branch or store as *unsafe*. Unsafe instructions may wake up and compete to issue as in a baseline OoO (i.e., they may issue when their operands become ready). But, when an unsafe micro-op completes execution (step 3 in Figure 6.2), it writes back to its destination physical register, but does not broadcast its destination tag to dependent instructions, does not mark its destination register ready, and does not forward its output value on the bypass network. Hence, dependent instructions will not issue and cannot observe the unsafe value.

Managing Value Propagation. When the eldest outstanding micro-op resolves, it marks instructions in the ROB *safe* until the next eldest unresolved branch/store. ROB entries are extended with three bits: `unsafe` tracks if the instruction follows a still-unresolved micro-op, `exec` tracks if the instruction has executed, and `bcast` tracks if the instruction has broadcast its tag to wake dependents. Upon instruction completion, if `unsafe`, tag broadcast is deferred. When a micro-op resolves, the `unsafe` bit for subsequent ROB entries until the next unresolved branch/store are cleared. `!unsafe && exec && !bcast` instructions arbitrate for tag broadcast ports, competing with instructions completing in the current cycle (completing instructions have priority to avoid pipeline stalls); `bcast` is set when broadcasting.

When *safe* instructions broadcast their tags to the issue queue, they mark their destination register(s) ready, waking their dependents (step 4 in Figure 6.2). We do not add additional tag broadcast ports to the ROB over baseline OoO; the number of broadcasts is unchanged, broadcasts are time-shifted until preceding micro-ops resolve. For example, assume that the broadcast bandwidth is four and that two instructions completed this cycle. If another three instructions were marked safe, two of these newly-safe instructions can wake dependents; the third waits for the next cycle. In the majority of the evaluation, we assume broadcast and wake-up of newly-*safe* instructions fit within the existing wake-up critical path. In Figure 6.9e, we include a sensitivity study that shows the impact of further delay due to critical path constraints; a one-cycle delay reduces CPI by less than 3.6%.

Figure 6.6 illustrates an ROB snapshot when executing code akin to Listing VI.1, depicting various *NDA* data propagation policies. Column ⓐ shows the ROB snapshot under strict propagation. The branch at Line 6 has not resolved, so all following instructions are marked *unsafe*. Whereas the instruction at Line 7 executes to completion, it is *unsafe* and therefore cannot wake the dependent instruction on Line 8.

Branches resolve when the branch micro-op completes execution. Upon a misprediction, all younger micro-ops in the ROB are squashed and renaming tables are recovered, discarding values in physical registers that never became safe, preventing potentially secret data from leaking.

6.5.2 Permissive Data Propagation

For threat models where *NDA* only protects secrets in memory or special registers, we can safely optimize performance via *permissive propagation* (rows 1-2 in table 6.2), which marks only *load* instructions after an unresolved branch/store as *unsafe*. Arithmetic and control instructions are unconditionally marked *safe* at dispatch.

The key intuition for this policy is that only loads can introduce new secret values into the microarchitecture. Loads that precede the eldest unresolved micro-op will commit their value to architectural GPRs, which are not protected under this threat model. Note that wrong-path execution due to exceptions (As in Meltdown or Foreshadow) are also not addressed under this threat model; we address these as chosen-code attacks (§6.5.3).

For example, consider two dependent instructions i_1 and i_2 fetched after an unresolved branch. If i_1 is an arithmetic instruction (any non-load), it is considered *safe*. Therefore, i_1 can broadcast its output upon completion—allowing i_2 to issue—without waiting for the branch to resolve.

This threat model also protects the contents of special registers (e.g., AVX or MSRs, see LazyFP [256] and Meltdown v3a [124]). The instructions to read these registers (e.g., `rdmsr`) are treated like `loads` and are also marked *unsafe* when dispatched after an *unresolved* branch.

Lines 7-8 in Figure 6.6 illustrates the difference between strict (column ⓐ) and permissive (column ⓑ) propagation. In contrast to strict propagation, the `lea` instruction on Line 7 is marked *safe* since it is not a load operation. Therefore, `lea` wakes its dependent instruction on Line 8 immediately.

Bypass Restriction (BR). To defeat SSB [124] attacks we introduce a new mechanism for safe store bypass, which we use in tandem with both strict and permissive propagation (rows 2,4 in table 6.2). In this scheme, unlike Intel's SSBD [141], loads are allowed to execute even if they bypass stores in the Load Store Queue (LSQ). However, loads are marked *unsafe* until all bypassed stores' addresses are resolved. If a bypassed store resolves its address in a way that generates an order violation, the offending load and younger instructions are squashed by the memory dependency unit.

6.5.3 Load Restriction

NDA protects against chosen-code attacks by blocking data propagation from speculative loads (row 5 in table 6.2), such as in Meltdown [173], Foreshadow [269, 282], LazyFP [256], and MDS attacks [272, 55, 241]. These attacks exploit specific flaws in processor implementations where data propagates from a load that will eventually fault. Each of these flaws has been individually patched [120, 141]. However, given the complexity of modern processor implementations, one might expect similar implementation errors in the future. Moreover, in the chosen-code context, there are a myriad of ways to induce wrong-path execution (faulting loads, Intel TSX transaction aborts, interrupt delivery, breakpoint and syscall instructions, performance counter overflow, load replay due to memory-order misspeculation [294, 88], etc.) As prior work [288] suggests, effective defenses must address the common problems underlying chosen-code attacks.

We therefore propose a blanket *NDA* protection policy, *load restriction*, which both

	Mechanism	Control steering (memory)	Control steering (GPRs)	Chosen code	Overhead vs. OoO
1	Perm. propagation	□			10.7%
2	Perm. propagation+BR	■			22.3%
3	Strict propagation	□	◇		36.1%
4	Strict propagation+BR	■	◇		45%
5	Load restriction	■		■	100%
6	Full protection (4+5)	■	◇	■	125%
7	InvisiSpec-Spectre*	○	○		7.6%
8	InvisiSpec-Future*	○	○	○	32.7%

■ Defeats all covert channels ○ Defeats d-cache based attacks

□ Defeats all covert channels, but does not block SSB

◇ Defeats all covert channels, except single micro-op GPR-attacks

∗ Our evaluation of InvisiSpec[69] on SPEC 2017 is detailed in §6.1

Table 6.2: *NDA* propagation policies (rows 1-6) and the attacks they prevent. Bypass Restriction (BR) adds protection against SSB (Spectre v4). Special registers, such as AVX and MSRs (LazyFP [256] and Spectre v3a [124]), are protected by treating their accesses like loads. None of the 25 documented attacks [56, 43] leak data from GPRs nor without at least two dependent micro-ops.

blocks all 11 documented [56, 272, 55, 241] chosen-code attacks and offers the potential to prevent future variants. Under load restriction, loads are considered unsafe until they are the eldest unretired instruction (i.e., at the head of the ROB). With load restriction, the micro-architecture guarantees that a load will wake its dependents if and only if it will immediately retire. Column ⓒ of Figure 6.6 illustrates an ROB snapshot when load restriction is used. The `loads` in Lines 1, 4 are independent and can execute concurrently, enabling high Memory & Instruction Level Parallelism MLP & ILP. However, each will wake its dependents (at Lines 2, 5) only when it retires.

6.5.4 Preventing All Classes of Attacks

To defeat both control-steering and chosen-code attacks, *NDA*'s final policy composes strict propagation and load restriction (row 6 in table 6.2). This *NDA* policy is the most defensive, so we call it *full protection*. Column ⓓ in Figure 6.6 illustrates an ROB snapshot when the full-protection policy is used. The loads on Lines 1 and 4 are issued and executed to completion, but are not considered *safe*. In contrast to the load-restriction case presented in Column ⓒ, the arithmetic operation on Line 7 is considered *unsafe* in Column ⓓ and therefore cannot wake the instruction on Line 8. However, parallel execution is still possible (e.g., lines 4 and 7 still execute in parallel) unlike in an in-order processor.

6.5.5 Security Analysis

Strict Propagation with Bypass Restriction. This policy protects secrets in memory and hinders exfiltration of secrets in GPRs via control-steering attacks. Spectre v1, v1.1, v2, v4 (SSB) [124], and ret2spec [176, 156] are blocked. Most importantly, NetSpectre [242], SMoTher Spectre [43], and the BTB attack (Section 6.3)—which are not addressed by prior work [288, 220, 260]—are defeated. For secrets residing in memory, the output of the access phase (① in Figure 6.3) cannot be used by the transmit phase ② in the same wrong-path execution window. For an attacker to leak contents from a GPR the transmit phase in a successful attack must comprise only micro-ops that do not depend on one another and that only depend on values from instructions prior to the branch. We note that all existing attacks (cf. table 6.1) require multiple dependent micro-ops to transmit secrets.

Permissive Propagation with Bypass Restriction. This policy protects secrets in memory but does not protect secrets in GPRs (e.g., `rax`). This level of protection is on par with the threat model presented in recent work [220, 260] with the added benefit of blocking *all* covert channels. All 14 documented control-steering attacks [56, 43], including those listed above, are blocked. Any `load` following an unresolved branch or store is marked unsafe. Therefore, the transmission phase ② will not be able to read the output of the

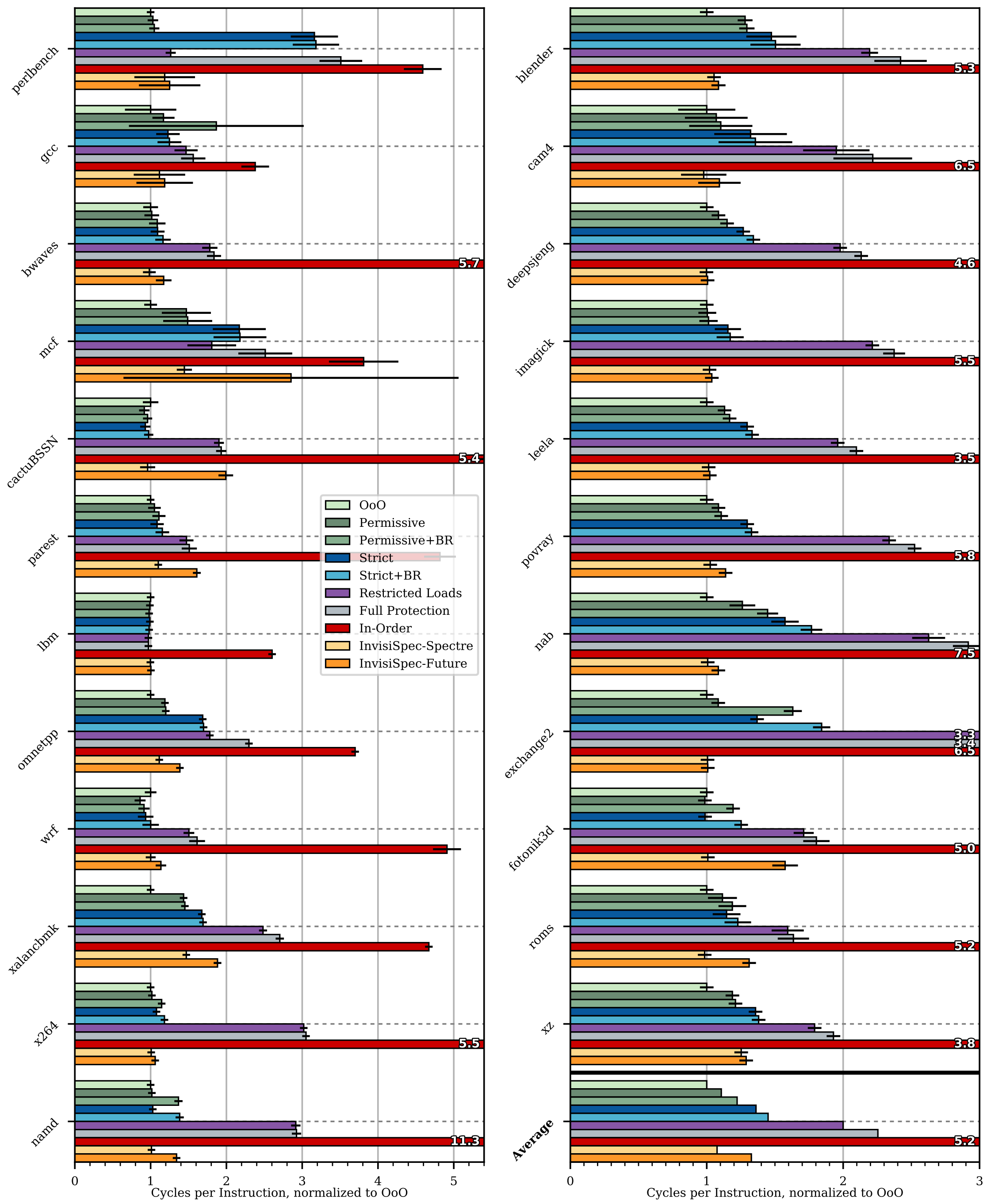


Figure 6.7: *NDA* and InvisiSpec [288] performance on SPEC 2017. Error bars depict the 95% confidence intervals.

`load`. However, unlike in strict propagation, non-load micro-ops are marked safe. If the secret already resides in a GPR, the attacker can pre-process and transmit the secret using a sequence of wrong-path operations.

Parameter	**Value**
Architecture	X86-64 at 2.0 GHz
Core (OoO)	8-issue, no SMT, 32 Load Queue entries, 32 Store Queue entries, 192 ROB entries, 4096 BTB entries, 16 RAS entries
Core (in-order)	TimingSimpleCPU from *gem5*
L1-I/L1-D Cache	32kB, 64B line, 8-way set associative (SA), 4 cycle round-trip (RT) latency, 1 port
L2 Cache	2MB, 64B line, 16-way SA, 40 cycle RT latency
DRAM	50ns response latency

Table 6.3: Gem5 simulation configuration.

Load Restriction. The *load restriction* policy addresses all known chosen-code attacks, including Spectre v3, v3a, v4 [124], LazyFP [256], Foreshadow/NG [269, 282, 120], and MDS attacks [272, 55, 241]. In chosen-code threat models, the attacker already controls the executed code, and can thus trivially access the contents of their own GPRs and memory space. Load restriction protects secrets in privileged memory and special registers. Specifically, any micro-op depending on a `load` (or `load`-like instruction) will be ready only after the load retires. Upon retirement, the values returned by `loads` are no longer speculative and are accordingly safe to read.

Load restriction also has the potential to block future chosen-code attacks that access memory and special registers. Additionally, given that none of the 25 existing speculative execution attacks [56, 43] leak secrets from GPRs, the load restriction policy prevents all known control-steering attacks.

Full Protection. Combining load restriction with the strict propagation policy (row 6 in table 6.2) offers the most defensive design point of *NDA*. The *full-protection* policy defeats all 25 known control-steering and chosen-code attacks exfiltrating data from memory, special registers, *and* hinders the attacker's ability to transmit contents of GPRs.

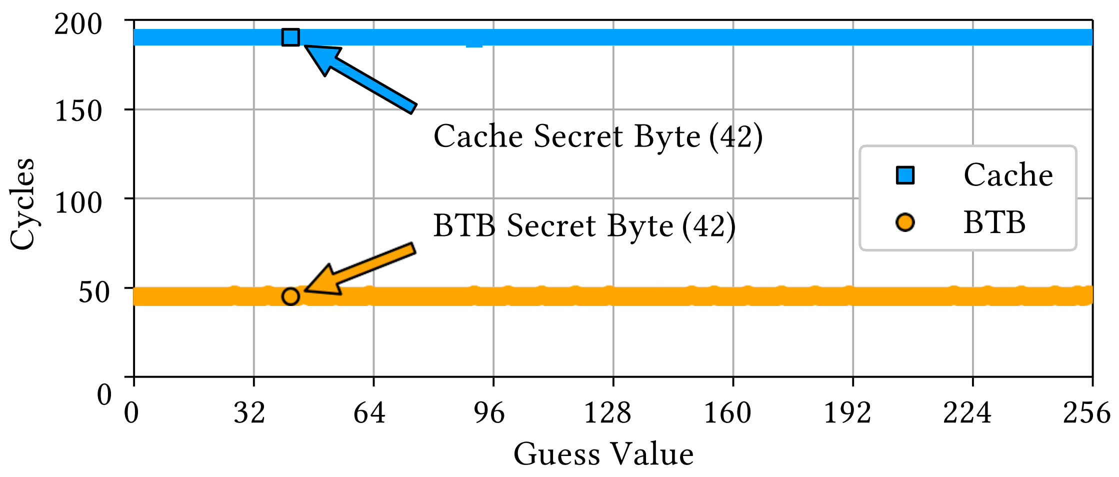


Figure 6.8: Spectre v1 when using *NDA* permissive propagation policy. The cycle differences in Figure 6.4 (Spectre v1 *without NDA*) are eliminated. Thus, *NDA* conceals the secret byte's value, regardless of the covert channel.

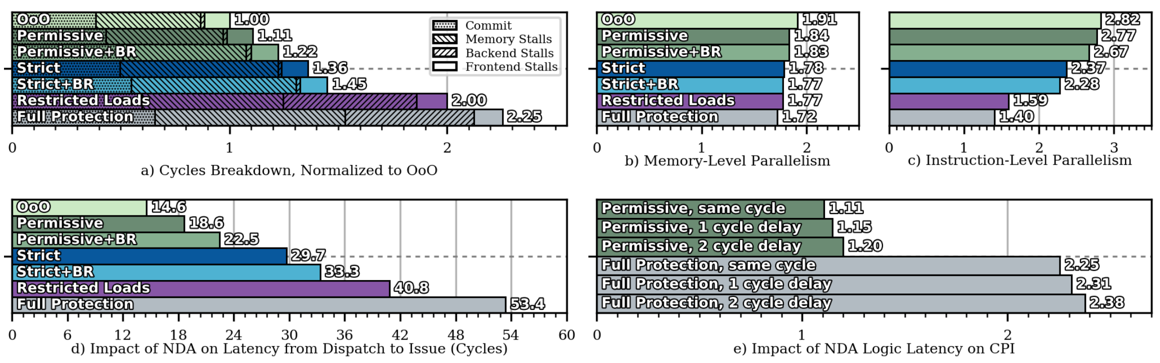


Figure 6.9: Aggregated statistics over SPEC 17 benchmarks. *(a) NDA* extend the cycles spent on commit and backend stalls. *(b),(c)* MLP & ILP is still high across *NDA* policies. *(d)* As expected, *NDA* causes delays in latency-to-issue. However, overall impact on CPI is substantially smaller. *(e)* The impact of *NDA* logic latency on CPI is relatively small.

6.6 Evaluation

We next demonstrate *NDA*'s effectiveness in mitigating speculative execution attacks and evaluate the performance of six different *NDA* policies.

6.6.1 Experimental Setup & Methodology

We evaluate *NDA* on *gem5* [44] running the SPEC CPU 2017 benchmark suite [253]. Table 6.3 shows the CPU configuration, which reflects a Haswell-like microarchitecture and matches that used in recent architectural studies of speculative execution attacks [288]. To obtain results that represent SPEC benchmark performance with statistical confidence guarantees, we extend *gem5* to enable a simulation sampling methodology similar to SMARTS [285]. We run SPEC benchmarks on real hardware (Haswell Xeon E5-2699) and dump snapshots of their execution state at fixed intervals using `gdb`. We have developed a new tool to convert these snapshots to *gem5* checkpoints and resume their execution in simulation [15, 14].

From each checkpoint, we warm simulation state for 5 million instructions and measure performance for 100,000 instructions. We validate that the number of unknown cache references during measurement (references to a cache set for which not all tags are initialized in warmup) is negligible (i.e., the worst-case performance error due to unknown cache references is much smaller than the sampling error). We report 95% confidence intervals of CPI in Figure 6.7.

We compare NDA's performance to both variants of InvisiSpec [288] with the same SMARTS methodology and *gem5* configuration, using the source code provided by the authors [13]. NDA's and InvisiSpec's performance for the baseline configuration on SPEC 17 are similar within the confidence interval. Absolute performance numbers for InvisiSpec, depicted in Figure 6.7, differ from the original paper due to different benchmarks (SPEC 06 vs. SPEC 17) and sampling methodology (a single billion-instruction segment vs. SMARTS sampling). Post-publication, the InvisiSpec authors released a bug fix that affects performance, which we include.

6.6.2 Effectiveness of NDA

We evaluate Spectre v1 [154] (Listing VI.1 and Listing VI.3) on unmodified *gem5* without *NDA* protections. As illustrated in Figure 6.4, both the cache and the BTB covert timing channels clearly leak the secret byte. For the correct guess of the secret byte, the cache covert channel yields a ~140-cycle decrease due to a cache hit. The BTB covert channel similarly yields a ~16-cycle decrease due to the overhead of mis-prediction, as shown in Figure 6.5. However, when running the Spectre v1 cache and BTB attacks with permissive propagation enabled, *NDA* blocks the speculative data leakage *regardless of the covert channel in use*. As depicted in Figure 6.8, the correct secret value is indistinguishable from the other 255 candidates.

6.6.3 NDA Performance

We evaluate *NDA*'s performance with ten different configurations; the six *NDA* policies described in Section 6.5, two baselines, and two InvisiSpec configurations. The baseline configurations are the in-order and unconstrained OoO processors listed in Table 6.3. The in-order processor represents the extreme case of no speculation and is thus trivially immune to speculative execution attacks. We note that, besides *NDA*'s *load-restriction* and *full-protection*, the in-order processor is the only other execution model known to defeat all 25 documented speculative execution attacks, regardless of the covert channel they use. The unconstrained OoO processor offers the best performance, but is insecure.

Cycles Per Instruction (CPI). Figure 6.7 depicts the CPI of all configurations across all benchmarks, normalized to OoO (averages at the bottom right). The overheads of different policies are summarized in table 6.2. Defeating SSB with Bypass Restriction (BR) adds 6.6-9.9 % overhead. In the case of *permissive propagation with BR* (row 2 in table 6.2)—the highest performance policy which prevents all 14 control-steering vulnerabilities—the average performance loss relative to the OoO baseline is 10.7%. This policy thwarts all known control-steering attacks and recovers 96% of the performance gap between the OoO and In-Order baselines.

In the case of *full protection* (row 6 in table 6.2)—the most secure policy—the average performance loss is 125%. This policy prevents all 25 documented variants of both control-steering and chosen-code attacks while also offering potential protection against future attacks. Despite the restrictions it imposes on the dynamic schedule, full protection still closes 68% of the performance gap between in-order and OoO.

Figure 6.9a depicts an average time breakdown for all OoO design variants. The bars are normalized to the baseline OoO design point. *Commit* cycles are cycles in which at least one instruction retires. *Memory stalls* are cycles in which the head of the ROB is an incomplete memory operation. *Back-end stalls* are cycles in which the head of the ROB is a non-memory operation that is not yet ready to retire. *Front-end stalls* are cycles in which the ROB is empty or cycles which are spent squashing wrong-path execution. *NDA* policies restrict data propagation and thereby limit dynamic scheduling. Therefore, on average, fewer instructions are committed in a given cycle, increasing the overall number of *commit* cycles. Since instruction-level parallelism for both memory and non-memory instructions is reduced, more cycles are spent on *memory stalls* and *back-end stalls*. *Front-end stall* cycles generally vary little across designs, on average contributing only 2% of the difference in cycles.

Wake-up Latency. *NDA* introduces a delay between instruction completion and tag broadcast. Whereas broadcast delay does not *directly* affect CPI, the delay propagates to dependent instructions in the ROB by delaying their issue. We measure this effect by measuring the average delay instructions experience from dispatch to wake-up under each design. The average latencies across all benchmarks are shown in Figure 6.9d. *NDA* policies add on average 4-39 cycles. This increased latency also manifests in up to 78% increase in cycles spent on *back-end stalls*, shown in Figure 6.9a. However, the wake-up latency has a modest impact on overall performance (CPI).

Memory and Instruction Parallelism (MLP/ILP). The favorable performance of

NDA compared to the in-order processor can be explained by observing the Memory- and Instruction-Level Parallelism of each profile. The geometric means of MLP & ILP across all benchmarks are depicted in Figure 6.9b-c. We follow Chou et al [65] and report MLP as the average outstanding off-chip misses when at least one is outstanding. Whereas the MLP & ILP in the various *NDA* profiles are at times lower than the OoO baseline by as much as 6% and 44% (respectively), they are better than the in-order baseline processor by 72% and 39%, where MLP & ILP cannot exceed 1.0. These results suggest that *NDA* enables execution parallelism among off-chip misses despite the scheduling restrictions of speculative instructions. Importantly, *NDA* does not typically restrict the issue time of loads, only when they may wake dependents. Ergo, typically only dependent loads are delayed, which do not add to MLP or ILP.

Comparison to InvisiSpec [288]. Since *NDA* and InvisiSpec have different threat models, detailed in table 6.2, a direct comparison is not straight forward. In the evaluation, InvisiSpec-Spectre defeats all cache-based control-steering attacks with 7.6% slowdown. In comparison, *NDA* blocks control-steering attacks, regardless of the covert channel they use, with 10.7%-36.1% slowdown, depending on where secrets reside. For futuristic chosen-code attacks, InvisiSpec-Future introduces 32.7% overhead compared to 125% in *NDA*. However, *NDA* blocks all covert channels, including port contention [43], the FPU [242], and the BTB (Section 6.3).

6.7 Related Work

The first micro-architectural side-channel attacks used the cache side channel to infer AES keys from a neighboring process or VM [217, 42, 45]. Since then, a myriad of side channel techniques have been developed, such as Flush+Reload [292] and other advanced techniques [18, 84, 146, 85, 222, 104, 296, 291]. We refer to these attacks as *classical* cache attacks. These attacks do not leverage speculative wrong-path execution. Other work demonstrates how the cache side channel can be used as a *covert channel* [284, 181, 287].

DRAM [4] and issue ports [16, 43] are also demonstrated as viable covert channels.

The first speculative execution attacks—Meltdown [173] and Spectre [154]—leveraged prior work on cache covert channels to transmit data obtained from wrong-path execution via the data-cache (*d-cache*). Other speculative attacks using various techniques to access secrets or steer execution also leveraged the d-cache covert channel [176, 256, 156, 59, 153, 269, 282, 120, 124, 77]. Since the d-cache covert channel is widely exploited, initial defenses [288, 150, 220, 237] have exclusively focused on protecting the d-cache. However, these defenses do not mitigate non d-cache speculative execution attacks [242, 56, 270, 43, 177]. Specifically, Mambretti et al. [177] demonstrated covert transmission of secrets via the instruction-cache (*i-cache*).

Unfortunately, it is not trivial to apply the same d-cache defense-techniques to provide i-cache protection. For example, Sakalis et al. [237] delay speculative loads on an L1 cache-miss to prevent speculative d-cache modifications. However, the authors mention it is difficult to apply the same policy to i-cache misses with low overhead: While d-cache delays do not preclude other in-flight instructions from executing OoO, i-cache delays stall the front-end and starve the entire pipeline.

InvisiSpec [288] allows speculative loads to execute using a dedicated buffer, only committing updates to the d-cache once speculation resolves. While the authors hypothesize that a similar method could be applied to the i-cache, they do not implement or evaluate the performance overhead of such i-cache protection. In comparison to cache-only defenses, *NDA* is agnostic to the covert channel used in the *Transmit Phase* and blocks all known attacks.

Conditional-Speculation [220] protects secrets placed in memory, but not in GPRs. In comparison, *NDA*'s strict-propagation prevents the attacker from performing the pre-processing required for the *Transmit Phase*. *NDA* thus defeats NetSpectre and SMother-Spectre attacks, while providing better protection for secrets in registers.

Prior work [260, 83] suggest mitigations to defeat the Spectre v1 variant. Taram et

```
1 stop_speculative_exec();
2 register long secret = *secret_addr;
3 // ... operate on secret
4 secret = 0; // scrub secret
5 resume_speculative_exec();
```

Listing VI.4: Closing the registers-to-memory security gap.

al. [260] suggest Context Sensitive Fencing, a hardware modification to automatically insert `lfence` micro-ops where needed, to block the d-cache channel. SpectreGuard [83] suggested delaying broadcast of completed micro-ops to defeat Spectre v1 across multiple covert channels. However, as stated by the authors, their main goal is to block Spectre v1 attacks. NDA defeats all known variants regardless of the covert channel they use.

Recent work (such as DAWG [152], CEASER [228], and others [289, 290]) hinder the attacker's ability to deterministically cause a cache line collision with another process or VM, thwarting most cache-based side and covert channels. However, these techniques do not mitigate attacks that use non-cache covert channels.

We addressed related work on deployed defense mechanisms for speculative execution attacks in Section 6.3.2.

6.8 Discussion

NDA is capable of defeating both control-steering and chosen-code attacks while performing considerably better than in-order processors. However, even though *NDA* blocks all known attacks, it may still be possible to use a control-steering attack to read general-purpose registers if there exists a feasible single micro-op that can leak the register's contents.

To protect registers, one can introduce an instruction or a processor mode that temporarily disables speculation and out-of-order execution during the window of vulnerability when a secret value is loaded from memory and resides in a register until it is overwritten. We illustrate such a defense in Listing VI.4. We note this defense would only be effective if used in addition to *NDA*. Without *NDA*, a control-steering attack could simply steer the execution to bypass Line 1 and speculatively execute Lines 2-3 to leak the register's contents.

6.9 Conclusion

Speculative execution attacks are challenging to mitigate. Blocking individual covert channels or specific exploitation techniques is insufficient. To design effective mitigations, I introduced a new classification of speculative execution attacks based on how each attack induces wrong-path execution. My new technique for controlling speculative data propagation, *NDA*, defeats all known speculative execution attacks and drastically reduces the attack surface for future variants. On SPEC 2017, I show that the four *NDA* design points offer effective and performant mitigations.

CHAPTER VII

Conclusion

In this thesis I analyzed the impact of TEE technologies on the security of services deployed in the cloud, identified key challenges and suggested various approaches for enabling usable and performant trusted execution.

In chapter II, I characterized the performance bottlenecks imposed by Intel SGX on common applications. While the overhead numbers are specific to SGX the bottlenecks identified are not. Encrypted memory and secure context switching are fundamental to TEEs including AMD SEV and ARM TrustZone. The optimization strategy I implemented and tested on SGX is also conceptually viable for the other two TEEs as well.

In the chapters III-IV I tackled the challenge of managing clients' permissions across a distributed TEE-based service and how cloud providers can support hardware-based integrity verification independently of the hardware vendor. Managing trustworthy distribute sate is paramount to trusting services running on third party machines. Decoupling cloud providers from the hardware vendor is crucial to making TEE technologies practical on a large scale deployment.

As I demonstrated in the chapter V, even perfectly secure code running in TEEs is vulnerable to speculative execution attacks. Unsurprisingly, even a single vulnerability—such as Foreshadow—can derail all security guarantees of trusted execution including confidentiality, long-term storage, and integrity. Further research is required to evaluated the resilience of SEV and TrustZone to such micro-architectural attacks.

Looking forward, in chapter VI I proposed a fundamental way to redesign out-of-order processors to defeat speculative execution micro-architectural attacks. Further research is required to refine the *NDA* approach in order to reduce its performance overhead. Hardware vendors should consider implementing *NDA*-like mechanism in their processors as a proactive approach against speculative execution attacks.

Together, the five concepts described in this thesis would hopefully strengthen the foundations of trustworthy computing, enabling better and cheaper services without compromising security and privacy of users data.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] *AMD: Secure Encrypted Virtualization API Version 0.14 Technical Preview*. http://support.amd.com/TechDocs/55766_SEV-KM%20API_Specification.pdf.

[2] *CERT: Vulnerability Note VU#491375 Intel Active Management Technology (AMT) does not properly enforce access control*. http://www.kb.cert.org/vuls/id/491375.

[3] *CVE-2017-5689 Detail*. https://nvd.nist.gov/vuln/detail/CVE-2017-5689.

[4] drama: Exploiting dram addressing for cross-cpu attacks.

[5] *Getting Started with Intel Active Management Technology (AMT)*. https://software.intel.com/en-us/articles/getting-started-with-intel-active-management-technology-amt.

[6] *HSM Buyers' Guide*. https://wiki.opendnssec.org/display/DOCREF/HSM+Buyers'+Guide#HSMBuyers%27Guide-3.Medium-tohigh-costcryptographicacc\elerators(PCIcards/separateunits).

[7] *http_load - multiprocessing http test client*. http://acme.com/software/http_load/.

[8] *HTTPS Certificate Revocation is broken, and its time for some new tools*. https://arstechnica.com/information-technology/2017/07/https-certificate-revocation-is-broken-and-its-time-for-some-new-

[9] *The Legion of the Bouncy Castle*. https://www.bouncycastle.org/.

[10] *Microsoft: Trusted Cloud*. https://www.microsoft.com/en-us/research/project/trusted-cloud/.

[11] *Revocation checking and Chrome's CRL*. https://www.imperialviolet.org/2012/02/05/crlsets.html.

[12] *SGX Secure Enclaves in Practice: Security and Crypto Review. Black Hat*. https://www.blackhat.com/docs/us-16/materials/us-16-Aumasson-SGX-Secure-Enclaves-In-Practice-Security-And-\Crypto-Review.pdf.

[13] *InvisiSpec-1.0 source code*, 2019. https://github.com/mjyan0720/InvisiSpec-1.0.

[14] *Lapidary: Crafting more beautiful gem5 simulations*, 2019. https://medium.com/@iangneal/lapidary-crafting-more-beautiful-gem5-simulations-4bc6f6aad717.

[15] *Lapidary: creating beautiful gem5 simulations*, 2019. https://github.com/efeslab/lapidary.

[16] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida Garcia, and Nicola Tuveri. Port contention for fun and profit. Cryptology ePrint Archive, Report 2018/1060, 2018. https://eprint.iacr.org/2018/1060.

[17] Fritz Alder, Arseny Kurnikov, Andrew Paverd, and N Asokan. Migrating sgx enclaves with persistent state. *arXiv preprint arXiv:1803.11021*, 2018.

[18] Thomas Allan, Billy Bob Brumley, Katrina E. Falkner, Joop van de Pol, and Yuval Yarom. Amplifying side channels through performance degradation. In *ACSAC*, pages 422–435. ACM, 2016.

[19] Tiago Alves and Don Felton. TrustZone: Integrated Hardware and Software Security-Enabling Trusted Computing in Embedded Systems. 2004.

[20] Amazon. *Amazon Law Enforcement Guidelines*. https://d0.awsstatic.com/certifications/Amazon_LawEnforcement_Guidelines.pdf.

[21] Amazon. *FICO Selects AWS as Its Cloud Provider*. http://phx.corporate-ir.net/phoenix.zhtml?c=176060&p=irol-newsArticle&ID=2293694.

[22] Amazon. *GRAIL Selects AWS as its Cloud Provider*. http://phx.corporate-ir.net/phoenix.zhtml?c=176060&p=irol-newsArticle&ID=2290165.

[23] Amazon. *NUC7i3BNH, reported by Intel to support SGX*. https://www.amazon.com/NUC7i3BNH-Dual-Core-i3-7100U-Bluetooth-Professional/dp/B071W57BKQ/ref=sr_1_2?s=electronics&ie=UTF8&qid=1508551702&sr=1-2&keywords=NUC7i3BNH.

[24] Amazon. *OpenVPN Access Server*. https://aws.amazon.com/marketplace/pp/B00MI40CAE/ref=mkt_wir_openvpn_byol.

[25] Amazons. https://aws.amazon.com/security/security-bulletins/AWS-2018-019/.

[26] AMD. *AMD64 Architecture Programmers Manual Volume 2: System Programming, ch. 7.10*. http://support.amd.com/TechDocs/24593.pdf.

[27] AMD. Speculative Store Bypass Disable. https://developer.amd.com/wp-content/resources/124441_AMD64_SpeculativeStoreBypassDisable_Whitepaper_final.pdf.

[28] Nikos Anastopoulos and Nectarios Koziris. Facilitating efficient synchronization of asymmetric threads on hyper-threaded processors. In *Proc. of IEEE IPDPS*, 2008.

[29] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. Innovative technology for CPU based attestation and sealing. In *Proc. of HASP*, 2013.

[30] Anjuna. *THE FORESHADOW ATTACK ON INTEL SGX*. https://www.anjuna.io/blog/2018/8/14/foreshadow.

[31] apigee. *Cleveland Clinic: Turbocharging Patient Data with API Management*. https://apigee.com/about/tags/healthcare.

[32] Apple. *App Store Review Guidelines*. https://developer.apple.com/app-store/review/guidelines/.

[33] Apprenda. *Internal Cloud Computing*. https://apprenda.com/library/cloud/internal-cloud-computing/.

[34] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Daniel O'Keeffe, Mark L Stillwell, et al. SCONE: Secure Linux Containers with Intel SGX. In *Proc. of OSDI*, 2016.

[35] Krste Asanović and David A Patterson. Instruction Sets Should be Free: The Case for RISC-V. Technical report, University of California at Berkeley, http://www. eecs. berkeley. edu/Pubs/TechRpts/2014/EECS-2014-146. pdf, 2014.

[36] Nicholas Luis Astete, Aaron Benjamin Brethorst, Joseph Michael Goldberg, Matthew Hanlon, Anthony A Hutchinson, Gopalakrishnan Janakiraman, Alexander Kotelnikov, Petr Novodvorskiy, David William Richardson, Roxanne Camille Skelly, et al. Multitenant hosted virtual machine infrastructure, June 25 2013. US Patent 8,473,594.

[37] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *Proc. of ACM SIGMETRICS Performance Evaluation Review*, 2012.

[38] Atmel. *Trusted Platform Module*. http://www.atmel.com/products/security-ics/embedded/.

[39] Pierre-Louis Aublin, Florian Kelbert, Dan OKeeffe, Divya Muthukumaran, Christian Priebe, Joshua Lind, Robert Krahn, Christof Fetzer, David Eyers, and Peter Pietzuch. Libseal: Revealing service integrity violations using trusted execution. *environments*, 2:5, 2018.

[40] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding Applications from an Untrusted Cloud with Haven. In *Proc. of ACM TOCS*, 2015.

[41] Jethro G Beekman, John L Manferdelli, and David Wagner. Attestation transparency: Building secure internet services for legacy clients. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, pages 687–698. ACM, 2016.

[42] Daniel J Bernstein. Cache-timing attacks on aes. 2005.

[43] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. SMoTherSpectre: exploiting speculative execution through port contention. *arXiv preprint arXiv:1903.01843*, 2019.

[44] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. The gem5 simulator. *ACM SIGARCH Computer Architecture News*, 39(2):1–7, 2011.

[45] Joseph Bonneau and Ilya Mironov. Cache-collision timing attacks against AES. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 201–215. Springer, 2006.

[46] Robert Bradford, Evangelos Kotsovinos, Anja Feldmann, and Harald Schiöberg. Live wide-area migration of virtual machines including local persistent state. In *Proceedings of the 3rd international conference on Virtual execution environments*, pages 169–179. ACM, 2007.

[47] Marcus Brandenburger, Christian Cachin, Matthias Lorenz, and Rüdiger Kapitza. Rollback and forking detection for trusted execution environments using lightweight collective memory. *arXiv preprint arXiv:1701.00981*, 2017.

[48] Ferdinand Brasser, Srdjan Capkun, Alexandra Dmitrienko, Tommaso Frassetto, Kari Kostiainen, Urs Müller, and Ahmad-Reza Sadeghi. DR.SGX: hardening SGX enclaves against cache attacks with data location randomization. *CoRR*, abs/1709.09917, 2017.

[49] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software grand exposure: SGX cache attacks are practical. In *11th USENIX Workshop on Offensive Technologies (WOOT 17)*, Vancouver, BC, 2017. USENIX Association.

[50] Stefan Brenner, Colin Wulf, David Goltzsche, Nico Weichbrodt, Matthias Lorenz, Christof Fetzer, Peter R Pietzuch, and Rüdiger Kapitza. Securekeeper: Confidential zookeeper using intel sgx. In *Middleware*, page 14, 2016.

[51] Ernie Brickell and Jiangtao Li. Enhanced privacy id from bilinear pairing for hardware authentication and attestation. *International Journal of Information Privacy, Security and Integrity 2*, 1(1):3–33, 2011.

[52] Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. When good instructions go bad: Generalizing return-oriented programming to RISC. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 27–38. ACM, 2008.

[53] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1041–1056, Vancouver, BC, 2017. USENIX Association.

[54] Rodrigo N Calheiros, Rajiv Ranjan, and Rajkumar Buyya. Virtual machine provisioning based on analytical performance and qos in cloud computing environments. In *Parallel processing (ICPP), 2011 international conference on*, pages 295–304. IEEE, 2011.

[55] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. Fallout: Leaking data on meltdown-resistant cpus. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2019.

[56] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A Systematic Evaluation of Transient Execution Attacks and Defenses. *arXiv preprint arXiv:1811.05441*, 2018.

[57] CBT. *Microsoft Azure users hit by 300% rise in cyber-attacks*. https://www.cbronline.com/cybersecurity/breaches/microsoft-azure-users-hit-300-rise-cyber-attacks/.

[58] Stephen Checkoway and Hovav Shacham. Iago attacks: Why the System Call API is a Bad Untrusted RPC Interface. In *Proc. of ACM SIGARCH Computer Architecture News*, 2013.

[59] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H Lai. Sgxpectre attacks: Leaking enclave secrets via speculative execution. *arXiv preprint arXiv:1802.09085*, 2018.

[60] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H. Lai. SgxPectre attacks: Leaking enclave secrets via speculative execution. *CoRR*, abs/1802.09085, 2018.

[61] Guoxing Chen, Wenhao Wang, Tianyu Chen, Sanchuan Chen, Yinqian Zhang, XiaoFeng Wang, Ten-Hwang Lai, and Dongdai Lin. Racing in hyperspace: Closing hyper-threading side channels on sgx with contrived data races. IEEE.

[62] Lily Chen. NIST special publication 800-108: Recommendation for key derivation using pseudorandom functions. *Information Technology Laboratory, NIST, Gaithersburg, MD*, pages 20899–8900, 2009.

[63] Xiaoxin Chen, Tal Garfinkel, E Christopher Lewis, Pratap Subrahmanyam, Carl Waldspurger, Dan Boneh, Jeffrey Dwoskin, and Dan Ports. Overshadow: A virtualization-based approach to retrofitting protection in commodity operating systems. In *Proc. of ACM SIGARCH Computer Architecture News*, 2008.

[64] Siddhartha Chhabra, Brian Rogers, Yan Solihin, and Milos Prvulovic. SecureME: a Hardware-Software Approach to Full System Security. In *Proc. of ACM ICS*, 2011.

[65] Yuan Chou, Brian Fahs, and Santosh Abraham. Microarchitecture optimizations for exploiting memory-level parallelism. In *Computer Architecture, 2004. Proceedings. 31st Annual International Symposium on*, pages 76–87. IEEE, 2004.

[66] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In *Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation-Volume 2*, pages 273–286. USENIX Association, 2005.

[67] CNBC. *Amazon, Microsoft, and Google respond to Intel chip vulnerability*. https://www.cnbc.com/2018/01/03/microsoft-google-respond-to-intel-chip-vulnerability.html.

[68] CNN. *Pentagon exposed some of its data on Amazon server*. http://money.cnn.com/2017/11/17/technology/centcom-data-exposed/index.html.

[69] Victor Costan and Srinivas Devadas. Intel SGX explained. Technical report, Cryptology ePrint Archive, Report 2016/086, 2016. https://eprint. iacr. org/2016/086.

[70] Victor Costan, Ilia Lebedev, and Srinivas Devadas. Sanctum: Minimal Hardware Extensions for Strong Software Isolation. In *Proc. of USENIX Security*, 2016.

[71] CRN. *Researchers Uncover 'Massive Security Flaws' In Amazon Cloud*. https://www.crn.com/news/cloud/231901911/researchers-uncover-massive-security-flaws-in-amazon-cloud.htm.

[72] Christoffer Dall, Shih-Wei Li, Jin Tack Lim, Jason Nieh, and Georgios Koloventzos. ARM Virtualization: Performance and Architectural Implications. In *Proc. of ISCA*, 2016.

[73] Debian. *Debian Bug report logs - #886367 intel-microcode: spectre microcode updates*. https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=886367.

[74] Zakir Durumeric, James Kasten, David Adrian, J Alex Halderman, Michael Bailey, Frank Li, Nicolas Weaver, Johanna Amann, Jethro Beekman, Mathias Payer, et al. The Matter of HeartBleed. In *Proc. of ACM IMC*.

[75] Shawn Embleton, Sherri Sparks, and Cliff C Zou. SMM Rootkits: A New Breed of OS Independent Malware. In *Security and Communication Networks*. Wiley Online Library, 2013.

[76] Epedient. *Private Cloud Computing*. https://www.expedient.com/services/infrastructure-as-a-service/cloud/private-cloud-computing/.

[77] Dmitry Evtyushkin, Ryan Riley, Nael CSE Abu-Ghazaleh, Dmitry Ponomarev, et al. BranchScope: A New Side-Channel Attack on Directional Branch Predictor. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 693–707. ACM, 2018.

[78] FFSB. *The Flexible Filesystem Benchmark*. https://github.com/FFSB-Prime/ffsb.

[79] Nadeem Firasta, Mark Buxton, Paula Jinbo, Kaveh Nasri, and Shihjong Kuo. Intel AVX: New frontiers in performance improvements and energy efficiency. 2008.

[80] Elia Florio. When malware meets rootkits. *Virus Bulletin*, 12, 2005.

[81] Forbes. *Heartland Payment Systems Suffers Data Breach (as many as 100 million debit and credit cards in 2008)*. https://www.forbes.com/sites/davelewis/2015/05/31/heartland-payment-systems-suffers-data-breach/#f78b778744ad.

[82] Fortanix. *Intel SGX*. https://fortanix.com/intel-sgx/.

[83] Jacob Fustos, Farzad Farshchi, and Heechul Yun. Spectreguard: An efficient data-centric defense mechanism against spectre attacks. In *DAC*, pages 61–1, 2019.

[84] Cesar Pereida García and Billy Bob Brumley. Constant-time callees with variable-time callers. In *USENIX Security Symposium*, pages 83–98. USENIX Association, 2017.

[85] Cesar Pereida García, Billy Bob Brumley, and Yuval Yarom. "make sure DSA signing exponentiations really are constant-time". In *ACM Conference on Computer and Communications Security*, pages 1639–1650. ACM, 2016.

[86] Gemalto. *SafeNet Network HSM*. https://safenet.gemalto.com/resources/data-protection/luna-sa-network-attached-hsm-product\-brief/.

[87] Daniel Genkin, Lev Pachmanov, Itamar Pipman, Eran Tromer, and Yuval Yarom. Ecdsa key extraction from mobile devices via nonintrusive physical side channels. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1626–1638. ACM, 2016.

[88] Kourosh Gharachorloo, Anoop Gupta, and John L Hennessy. Two techniques to enhance the performance of memory consistency models. 1991.

[89] Google. *Giving banks a friendly, modern makeover with digital signage: Heritage Bank*. https://enterprise.google.com/chrome/case-studies/heritage-bank/.

[90] Google. *Google Play Developer Distribution Agreement*. https://play.google.com/intl/ALL_us/about/developer-distribution-agreement.html.

[91] Google. *Google Play Protect*. https://www.android.com/play-protect/.

[92] Google. *Legal process for user data requests FAQs*. https://support.google.com/transparencyreport/answer/7381738/.

[93] Google. *NextPlane: Improving patient safety with machine learning*. https://cloud.google.com/customers/nextplane/.

[94] Google. *Omise provides easy-to-deploy payment gateway services for merchants on Google Cloud Platform*. https://cloud.google.com/customers/omise/.

[95] Google. *Protecting against the new L1TF speculative vulnerabilities*. https://cloud.google.com/blog/products/gcp/protecting-against-the-new-l1tf-speculative-vulnerabilities.

[96] Google. *Retpoline: a software construct for preventing branch-target-injection*. https://support.google.com/faqs/answer/7625886.

[97] Google. *Serving Civil Subpoenas or Other Civil Requests on Google*. https://support.google.com/faqs/answer/6151275?hl=en.

[98] Google. *Speculative Load Hardening*. https://docs.google.com/document/d/1wwcfv3UV9ZnZVcGiGuoITT_61e_Ko3TmoCS3uXLcJR0/edit#heading=h.phdehs44eom6.

[99] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. Cache attacks on intel SGX. In *EUROSEC*, pages 2:1–2:6. ACM, 2017.

[100] Daniel Gruss, Julian Lettner, Felix Schuster, Olya Ohrimenko, Istvan Haller, and Manuel Costa. Strong and efficient cache side-channel protection using hardware transactional memory. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 217–233, Vancouver, BC, 2017. USENIX Association.

[101] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. Kaslr is dead: long live kaslr. In *International Symposium on Engineering Secure Software and Systems*, pages 161–176. Springer, 2017.

[102] Daniel Gruss, Moritz Lipp, Michael Schwarz, Daniel Genkin, Jonas Juffinger, Sioli O'Connell, Wolfgang Schoechl, and Yuval Yarom. Another flip in the wall of Rowhammer defenses. In *IEEESP*, 2018.

[103] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+ flush: a fast and stealthy cache attack. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 279–299. Springer, 2016.

[104] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache template attacks: Automating attacks on inclusive last-level caches. In *USENIX Security Symposium*, pages 897–912. USENIX Association, 2015.

[105] Jinyu Gu, Zhichao Hua, Yubin Xia, Haibo Chen, Binyu Zang, Haibing Guan, and Jinming Li. Secure live migration of sgx enclaves on untrusted cloud. In *Dependable Systems and Networks (DSN), 2017 47th Annual IEEE/IFIP International Conference on*, pages 225–236. IEEE, 2017.

[106] Liang Gu, Xuhua Ding, Robert Huijie Deng, Bing Xie, and Hong Mei. Remote attestation on program execution. In *Proceedings of the 3rd ACM Workshop on Scalable Trusted Computing*, STC '08, pages 11–20, New York, NY, USA, 2008. ACM.

[107] Shay Gueron. A memory encryption engine suitable for general purpose processors. *IACR Cryptology ePrint Archive*, 2016:204, 2016.

[108] NIST Electronic Authentication Guideline. Special publication 800-63 version 1.0. 2, 2006.

[109] David Gullasch, Endre Bangerter, and Stephan Krenn. Cache games – bringing access-based cache attacks on AES to practice. In *IEEESP*, pages 490–505, May 2011.

[110] J Alex Halderman, Seth Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A Calandrino, Ariel Feldman, Jacob Appelbaum, and Edward Felten. Lest we remember: Cold-boot attacks on encryption keys. In *Communications of the ACM*, 2009.

[111] John L Henning. SPEC CPU2006 benchmark descriptions. In *Proc. of ACM SIGARCH Computer Architecture News*, 2006.

[112] Andrew Douglas Hilton, BC Lee, and TS Lehman. PoisonIvy: Safe Speculation for Secure Memory. In *Proc. of ACM MICRO*, 2016.

[113] Matthew Hoekstra, Reshma Lal, Pradeep Pappachan, Vinay Phegade, and Juan Del Cuvillo. Using innovative instructions to create trustworthy software solutions. In *Proc. of HASP*, 2013.

[114] Russell Housley, William Polk, Warwick Ford, and David Solo. Internet x. 509 public key infrastructure certificate and certificate revocation list (crl) profile. Technical report, 2002.

[115] Tyler Hunt, Zhiting Zhu, Yuanzhong Xu, Simon Peter, and Emmett Witchel. Ryoan: A Distributed Sandbox for Untrusted Computation on Secret Data. In *Proc. of OSDI*, 2016.

[116] IBM. *Cloud Data Guard Technology Preview*. https://www.ibmdataguard.com/.

[117] IBM. *IBM 4767 Performance Measurements CA Release 5.2.23*. http://www-03.ibm.com/security/cryptocards/pciecc2/pdf/CCA_release_5_2_23_perf.pdf.

[118] IETF. *The AES-CMAC Algorithm*. https://tools.ietf.org/html/rfc4493.

[119] IETF. *The Transport Layer Security (TLS) Protocol Version 1.2*. https://www.ietf.org/rfc/rfc5246.txt.

[120] Intel. *Deep Dive: Intel Analysis of L1 Terminal Fault*. https://software.intel.com/security-software-guidance/insights/deep-dive-intel-analysis-l1-terminal-fault.

[121] Intel. *Description and mitigation overview for L1 Terminal Fault*. https://software.intel.com/security-software-guidance/software-guidance/l1-terminal-fault.

[122] Intel. *EPID Provisioning and Attestation Services*. https://software.intel.com/sites/default/files/managed/ac/40/2016%20WW10%20sgx%20provisioning%20and%20attesatation%20final.pdf.

[123] Intel. *How to Run Intel Software Guard Extensions*. https://software.intel.com/en-us/blogs/2016/05/30/usage-of-simulation-mode-in-sgx-enhanced-application.

[124] Intel. Intel Analysis of Speculative Execution Side Channels. https://software.intel.com/security-software-guidance/api-app/sites/default/files/336983-Intel-Analysis-of-Speculative-Execution-Side-Channels-White
pdf.

[125] Intel. *Intel Developer Zone: GROUP OUT OF DATE from IAS*. https://software.intel.com/pt-br/node/742927.

[126] Intel. *Intel Integrated Performance Primitives*. https://software.intel.com/en-us/intel-ipp.

[127] Intel. *Intel SGX and Side-Channels*. https://software.intel.com/en-us/articles/intel-sgx-and-side-channels.

[128] Intel. *Intel SGX Software Development Kit (SDK)*. https://software.intel.com/en-us/sgx-sdk.

[129] Intel. *Intel Software Guard Extensions*. https://software.intel.com/sites/default/files/332680-001.pdf.

[130] Intel. *Intel Software Guard Extensions - Enclave Writer's Guide*. https://software.intel.com/sites/default/files/managed/ae/48/Software-Guard-Extensions-Enclave-Writers-Guide.pdf.

[131] Intel. *Intel Software Guard Extensions SDK for Linux OS*. https://01.org/sites/default/files/documentation/intel_sgx_sdk_developer_reference_for_linux_os_pdf.pdf.

[132] Intel. *Intel Software Guard Extensions SDK for Windows OS*. https://software.intel.com/sites/default/files/managed/b4/cf/Intel-SGX-SDK-Developer-Reference-for-Windows-OS.pdf.

[133] Intel. *Intel(R) Software Guard Extensions for Linux* OS*. https://github.com/01org/linux-sgx.

[134] Intel. *Retpoline: A Branch Target Injection Mitigation*. https://software.intel.com/sites/default/files/managed/1d/46/Retpoline-A-Branch-Target-Injection-Mitigation.pdf.

[135] Intel. *SGX Update*. https://security-center.intel.com/advisory.aspx?intelid=INTEL-SA-00076&languageid=en-fr.

[136] Intel. *Software Developer Manual*. http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf.

[137] Intel. *Software Developer Manual, chapters 37-43*. http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf.

[138] Intel. *Software Guard Extensions: EPID Provisioning and Attestation Services*. https://software.intel.com/sites/default/files/managed/57/0e/ww10-2016-sgx-provisioning-and-attestation-final.pdf.

[139] Intel. *Software Guard Extensions: EPID Provisioning and Attestation Services API*. https://software.intel.com/sites/default/files/managed/7e/3b/ias-api-spec.pdf.

[140] Intel. Software guard extensions programming reference, revision 2.

[141] Intel. Speculative Execution Side Channel Mitigations. https://software.intel.com/security-software-guidance/api-app/sites/default/files/336996-Speculative-Execution-Side-Channel-Mitigations.pdf.

[142] Intel. *Details and Mitigation Information for Variant 4*, 2018. https://newsroom.intel.com/editorials/addressing-new-research-for-side-channel-analysis/#gs.4778nz.

[143] Intel. *Intel Software Guard Extensions (SGX) SW Development Guidance for Potential Edger8r Generated Code Side Channel Exploits*, March 2018. https://software.intel.com/sites/default/files/managed/e1/ec/180309_SGX_SDK_Developer_Guidance_Edger8r.pdf.

[144] Iperf. *A tool for active measurements of the maximum achievable bandwidth on IP networks*. https://iperf.fr/.

[145] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. S$a: a shared cache attack that works across cores and defies vm sandboxing–and its application to aes. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 591–604. IEEE, 2015.

[146] Gorka Irazoqui Apecechea, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. Wait a minute! A fast, cross-VM attack on AES. In *RAID*, volume 8688 of *Lecture Notes in Computer Science*, pages 299–319. Springer, 2014.

[147] Yeongjin Jang, Jaehyuk Lee, Sangho Lee, and Taesoo Kim. Sgx-bomb: Locking down the processor via Rowhammer attack. In *SysTEX'17*, pages 5:1–5:6. ACM, 2017.

[148] Ryan Jansen and Paul R Brenner. Energy efficient virtual machine allocation in the cloud. In *Green Computing Conference and Workshops (IGCC), 2011 International*, pages 1–8. IEEE, 2011.

[149] Simon Johnson, Vinnie Scarlata, Carlos Rozas, Ernie Brickell, and Frank Mckeen. Intel® software guard extensions: Epid provisioning and attestation services. *White Paper*, 1:1–10, 2016.

[150] Khaled N Khasawneh, Esmaeil Mohammadian Koruyeh, Chengyu Song, Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. SafeSpec: Banishing the Spectre of a Meltdown with Leakage-Free Speculation. *arXiv preprint arXiv:1806.05179*, 2018.

[151] 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 *ACM SIGARCH Computer Architecture News*, volume 42, pages 361–372. IEEE Press, 2014.

[152] Vladimir Kiriansky, Ilia Lebedev, Saman Amarasinghe, Srinivas Devadas, and Joel Emer. DAWG: A Defense Against Cache Timing Attacks in Speculative Execution Processors.

[153] Vladimir Kiriansky and Carl Waldspurger. Speculative buffer overflows: Attacks and defenses. *arXiv preprint arXiv:1807.03757*, 2018.

[154] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. *arXiv preprint arXiv:1801.01203*, 2018.

[155] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. Spectre attacks: Exploiting speculative execution. In *40th IEEE Symposium on Security and Privacy*, 2019.

[156] Esmaeil Mohammadian Koruyeh, Khaled N Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. Spectre returns! speculation attacks using the return stack buffer. In *12th USENIX Workshop on Offensive Technologies, WOOT*, pages 13–14, 2018.

[157] Robert Krahn, Bohdan Trach, Anjo Vahldiek-Oberwagner, Thomas Knauth, Pramod Bhatotia, and Christof Fetzer. Pesos: Policy enhanced secure object store. In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys)*, 2018.

[158] D Richard Kuhn, Vincent C Hu, W Timothy Polk, and Shu-Jen Chang. Introduction to public key technology and the federal pki infrastructure. Technical report, National Inst of Standards and Technology Gaithersburg MD, 2001.

[159] Sanjay Kumar, Himanshu Raj, Karsten Schwan, and Ivan Ganev. Re-architecting VMMs for Multicore Systems: The Sidecore Approach. In *Proc. of WIOSCA*. Citeseer, 2007.

[160] Dmitrii Kuvaiskii, Oleksii Oleksenko, Sergei Arnautov, Bohdan Trach, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. Sgxbounds: Memory safety for shielded execution. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 205–221. ACM, 2017.

[161] Jaehyuk Lee, Jinsoo Jang, Yeongjin Jang, Nohyun Kwak, Yeseul Choi, Changho Choi, Taesoo Kim, Marcus Peinado, and Brent B Kang. Hacking in darkness: Return-oriented programming against secure enclaves. In *USENIX Security*, pages 523–539, 2017.

[162] Jaehyuk Lee, Jinsoo Jang, Yeongjin Jang, Nohyun Kwak, Yeseul Choi, Changho Choi, Taesoo Kim, Marcus Peinado, and Brent ByungHoon Kang. Hacking in darkness: Return-oriented programming against secure enclaves. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 523–539, Vancouver, BC, 2017. USENIX Association.

[163] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring fine-grained control flow inside sgx enclaves with branch shadowing. *arXiv preprint arXiv:1611.06952*, 2016.

[164] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 557–574, Vancouver, BC, 2017. USENIX Association.

[165] Jiaxin Li, Dongsheng Li, Yuming Ye, and Xicheng Lu. Efficient multi-tenant virtual machine allocation in cloud data centers. *Tsinghua Science and Technology*, 20(1):81–89, 2015.

[166] LibreSSL. *Cryptography and SSL/TLS Toolkit*. https://www.libressl.org/.

[167] lighttpd. *An open-source web server optimized for speed-critical environments*. https://www.lighttpd.net/.

[168] Kevin Lim, David Meisner, Ali Saidi, Parthasarathy Ranganathan, and Thomas Wenisch. Thin servers with smart pipes: Designing SoC accelerators for memcached. In *Proc. of ACM SIGARCH Computer Architecture News*, 2013.

[169] Joshua Lind, Christian Priebe, Divya Muthukumaran, Dan OKeeffe, Pierre-Louis Aublin, Florian Kelbert, Tobias Reiher, David Goltzsche, David Eyers, Rüdiger Kapitza, et al. Glamdring: Automatic application partitioning for intel sgx. In *2017 USENIX Annual Technical Conference (USENIX ATC 17), Santa Clara, CA*, 2017.

[170] Linux PAM. *shadow v4.2.1*. http://linux-pam.org/library/Linux-PAM-1.3.0.tar.bz2.

[171] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. Armageddon: Cache attacks on mobile devices. In *USENIX Security Symposium*, pages 549–564, 2016.

[172] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown. *arXiv preprint arXiv:1801.01207*, 2018.

[173] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown. *arXiv preprint arXiv:1801.01207*, 2018.

[174] Jiuxing Liu and Bulent Abali. Virtualization polling engine (VPE): Using Dedicated CPU Cores to Accelerate I/O Virtualization. In *Proc. of ACM ICS*, 2009.

[175] LWN. *A page-table isolation update*. https://lwn.net/Articles/752621/.

[176] Giorgi Maisuradze and Christian Rossow. ret2spec: Speculative execution using return stack buffers. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 2109–2122. ACM, 2018.

[177] Andrea Mambretti, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, and Anil Kurmus. Two methods for exploiting speculative control flow hijacks. In *13th USENIX Workshop on Offensive Technologies (WOOT 19)*, 2019.

[178] Mohammad Mannan and PC van Oorschot. 3rd usenix workshop on hot topics in security (hotsec08). 2008.

[179] Simon Marechal. Advances in password cracking. *Journal in computer virology*, 4(1):73–81, 2008.

[180] Sinisa Matetic, Mansoor Ahmed, Kari Kostiainen, Aritra Dhar, David Sommer, Arthur Gervais, Ari Juels, and Srdjan Capkun. ROTE: Rollback protection for trusted execution. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1289–1306, Vancouver, BC, 2017. USENIX Association.

[181] Clémentine Maurice, Christoph Neumann, Olivier Heen, and Aurélien Francillon. C5: cross-cores cache covert channel. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 46–64. Springer, 2015.

[182] Ross Mcilroy, Jaroslav Sevcik, Tobias Tebbi, Ben L Titzer, and Toon Verwaest. Spectre is here to stay: An analysis of side-channels and speculative execution. *arXiv preprint arXiv:1902.05178*, 2019.

[183] 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. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*. ACM, 2013.

[184] Dirk Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal*, 2014(239):2, 2014.

[185] Ralph C. Merkle. A digital signature based on a conventional encryption function. In *CRYPTO*, pages 369–378, 1987.

[186] Microsoft. *Deploy an Openvpn Access Server*. https://azure.microsoft.com/en-us/resources/templates/openvpn-access-server-ubuntu/.

[187] Microsoft. *Hyper-V HyperClear Mitigation for L1 Terminal Fault*. https://blogs.technet.microsoft.com/virtualization/2018/08/14/hyper-v-hyperclear/.

[188] Microsoft. *Mitigating speculative execution side channel hardware vulnerabilities*. https://blogs.technet.microsoft.com/srd/2018/03/15/mitigating-speculative-execution-side-channel-hardware-vulnerabili

[189] Microsoft. *Prescription for data visualization: lab uses Power BI to aggregate behavioral health and other test data*. https://customers.microsoft.com/en-us/story/precisiondiagnostics-healthprovider-powerbi.

[190] Microsoft. *Protect your Windows devices against Spectre and Meltdown*. https://support.microsoft.com/en-us/help/4073757/protect-your-windows-devices-against-spectre-meltdown.

[191] Microsoft. *Responding to government and law enforcement requests to access customer data*. https://www.microsoft.com/en-us/trustcenter/privacy/govt-requests-for-data.

[192] Microsoft. *What is a private cloud?* https://azure.microsoft.com/en-us/overview/what-is-a-private-cloud/.

[193] Ahmad Moghimi, Thomas Eisenbarth, and Berk Sunar. Memjam: A false dependency attack against constant-time crypto implementations in SGX. In *CT-RSA*, volume 10808 of *Lecture Notes in Computer Science*, pages 21–44. Springer, 2018.

[194] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. Cachezoom: How SGX amplifies the power of cache attacks. In *CHES*, volume 10529 of *Lecture Notes in Computer Science*, pages 69–90. Springer, 2017.

[195] NAKIVO. *Crash-Consistent vs. Application-Consistent Backup*. https://www.nakivo.com/blog/crash-consistent-vs-application-consistent-backup/.

[196] Gil Neiger, Amy Santoni, Felix Leung, Dion Rodgers, and Rich Uhlig. Intel virtualization technology: Hardware support for efficient processor virtualization. In *Intel Technology Journal*, 2006.

[197] Netcraft. *April 2018 Web Server Survey*. https://news.netcraft.com/archives/2018/.

[198] New York Times. *Equifax Says Cyberattack May Have Affected 143 Million in the U.S.* https://www.nytimes.com/2017/09/07/business/equifax-cyberattack.html.

[199] New York Times. *Uber Hid 2016 Breach, Paying Hackers to Delete Stolen Data)*. https://www.nytimes.com/2017/11/21/technology/uber-hack.html.

[200] nginx. *Nginx web server*. http://nginx.org/.

[201] NIST. *Digital Signature Standard (DSS)*. https://nvlpubs.nist.gov/nistpubs/fips/nist.fips.186-4.pdf.

[202] NIST. *The Elliptic Curve Cryptography Cofactor Diffie-Hellman (ECC CDH) Primitive*. https://csrc.nist.gov/CSRC/media/Projects/Cryptographic-Algorithm-Validation-Program/documents/components/ecccdhvs.pdf.

[203] NIST. *FIPS 140-2: SECURITY REQUIREMENTS FOR CRYPTOGRAPHIC MODULES*. http://csrc.nist.gov/publications/fips/fips140-2/fips1402.pdf.

[204] NIST. *Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC*. http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38d.pdf.

[205] NIST. *Recommendation for Pair-Wise Key Establishment Schemes Using Discrete Logarithm Cryptography*. http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-56Ar2.pdf.

[206] Oasis Labs. *A New Era for Scalability and Privacy-Preserving Built with ever Smart Contracts Platform*. https://medium.com/@icogens.platform/oasis-labs-a-new-era-for-scalability-and-privacy-preserving-built

[207] Philippe Oechslin. Making a faster cryptanalytic time-memory trade-off. In *Annual International Cryptology Conference*, pages 617–630. Springer, 2003.

[208] Dan O'Keeffe, Divya Muthukumaran, Pierre-Louis Aublin, Florian Kelbert, Christian Priebe, Josh Lind, Huanzhou Zhu, and Peter Pietzuch. SGXSpectre - Spectre attack against SGX. https://github.com/lsds/spectre-attack-sgx, 2018.

[209] Oleksi Oleksenko, Bohdan Trach, Robert Krahn, Andre Martin, Mark Silberstein, and Christof Fetzer. VARYS: Protecting sgx enclaves from practical side-channel attacks. In *USENIX Annual Technical Conference*, 2018.

[210] Oleksii Oleksenko, Bohdan Trach, Tobias Reiher, Mark Silberstein, and Christof Fetzer. You shall not bypass: Employing data dependencies to prevent bounds check bypass. *arXiv preprint arXiv:1805.08506*, 2018.

[211] Opendnssec. *SoftHSM*. https://www.opendnssec.org/softhsm/.

[212] OpenSSL. *Cryptography and SSL/TLS Toolkit*. https://www.openssl.org/.

[213] OpenVPN. *An open source SSL VPN solution*. https://openvpn.net/.

[214] Openwall. *John the Ripper password cracker*. http://www.openwall.com/john/.

[215] Yossef Oren, Vasileios P Kemerlis, Simha Sethumadhavan, and Angelos D Keromytis. The spy in the sandbox: Practical cache attacks in javascript and their implications. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 1406–1418. ACM, 2015.

[216] Meni Orenbach, Pavel Lifshits, Marina Minkin, and Mark Silberstein. Eleos: Exitless os services for sgx enclaves. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 238–253. ACM, 2017.

[217] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of aes. In *Cryptographers Track at the RSA Conference*, pages 1–20. Springer, 2006.

[218] Jaemin Park, Sungjin Park, Jisoo Oh, and Jong-Jin Won. Toward live migration of sgx-enabled virtual machines. In *Services (SERVICES), 2016 IEEE World Congress on*, pages 111–112. IEEE, 2016.

[219] Payment Card Industry. *Hardware Security Module (HSM): Security Requirements version 1.0*. https://www.pcisecuritystandards.org/documents/PCI%20HSM%20Security%20Requirements%20v1.0%20final.pdf.

[220] Lutan Zhao Peinan Li and CAS) Rui Hou (Institute of Information Engineering, CAS); Lixin Zhang (HXT Semiconductor Co.LTD); Dan Meng (Institute of Information Engineering. Conditional Speculation: An Effective Approach to Safeguard Out-of-Order Execution Against Spectre Attacks. In *Proceedings of the 25th IEEE International Symposium on High-Performance Computer Architecture*. IEEE, 2019.

[221] Colin Percival. Cache missing for fun and profit, 2005.

[222] Peter Pessl, Leon Groot Bruinderink, and Yuval Yarom. To BLISS-B or not to be: Attacking strongswan's implementation of post-quantum signatures. In *CCS*, pages 1843–1855. ACM, 2017.

[223] Raluca Ada Popa, Catherine Redfield, Nickolai Zeldovich, and Hari Balakrishnan. CryptDB: Protecting confidentiality with encrypted query processing. In *Proc. of SOSP*, 2011.

[224] Donald E Porter, Silas Boyd-Wickizer, Jon Howell, Reuben Olinsky, and Galen C Hunt. Rethinking the Library OS from the Top Down. In *Proc. of ACM SIGPLAN*, 2011.

[225] Christian Priebe, Kapil Vaswani, and Manuel Costa. Enclavedb - a secure database using sgx. IEEE, May 2018.

[226] Project Zero. *Trust Issues: Exploiting TrustZone TEEs*. https://googleprojectzero.blogspot.com/2017/07/trust-issues-exploiting-trustzone-tees.html.

[227] Nguyen Quang-Hung, Pham Dac Nien, Nguyen Hoai Nam, Nguyen Huynh Tuong, and Nam Thoai. A genetic algorithm for power-aware virtual machine allocation in private cloud. In *Information and Communication Technology-EurAsia Conference*, pages 183–191. Springer, 2013.

[228] Moinuddin K Qureshi. CEASER: Mitigating Conflict-Based Cache Attacks via Encrypted-Address and Remapping. In *Proceedings of 51th International Symposium on Microarchitecture*, 2019.

[229] Rackspace. *Single-Tenant Security for Your Cloud*. `https://www.rackspace.com/en-us/cloud/private`.

[230] Kaveh Razavi, Ben Gras, Erik Bosman, Bart Preneel, Cristiano Giuffrida, and Herbert Bos. Flip feng shui: Hammering a needle in the software stack. In *Proceedings of the 25th USENIX Security Symposium*, 2016.

[231] Redis Labs. *memtier_benchmark: A High-Throughput Benchmarking Tool for Redis & Memcached*. `https://https://redislabs.com/blog/memtier_benchmark-a-high-throughput-benchmarking-tool-for-redis-memcached` `.WBz0PNzHXeA`.

[232] Redis Labs. *The World's Fastest Database*.

[233] The Register. *Sensitive client emails, usernames, passwords exposed in Deloitte hack*. `https://www.theregister.co.uk/2017/09/25/deloitte_email_breach/`.

[234] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-oriented programming: Systems, languages, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 15(1):2, 2012.

[235] Joanna Rutkowska. Subverting vistatm kernel for fun and profit. *Black Hat Briefings*, 2006.

[236] Paul Saab. Scaling Memcached at Facebook. 2008.

[237] Christos Sakalis, Stefanos Kaxiras, Alberto Ros, Alexandra Jimborean, and Magnus Själander. Efficient invisible speculative execution through selective delay and value prediction. In *Proceedings of the 46th International Symposium on Computer Architecture*, pages 723–735. ACM, 2019.

[238] Jamal Hadi Salim, Robert Olsson, and Alexey Kuznetsov. Beyond softnet. In *Proc. of USENIX ALSC*, 2001.

[239] Vipin Samar. Unified login with pluggable authentication modules (pam). In *Proceedings of the 3rd ACM conference on Computer and communications security*, pages 1–10. ACM, 1996.

[240] Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. VC3: Trustworthy data analytics in the cloud using SGX. In *Proc. of IEEE S&P*, 2015.

[241] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. ZombieLoad: Cross-privilege-boundary data sampling. *arXiv:1905.05726*, 2019.

[242] Michael Schwarz, Martin Schwarzl, Moritz Lipp, and Daniel Gruss. Netspectre: Read arbitrary memory over network. *arXiv preprint arXiv:1807.10535*, 2018.

[243] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Malware guard extension: Using SGX to conceal cache attacks. In *DIMVA*, pages 3–24, 2017.

[244] SCM. *Another misconfigured Amazon S3 server leaks data of 50,000 Australian employees.* `https://www.scmagazine.com/contractor-misconfigures-aws-exposes-data-of-50000-australian-empl` `article/704873/`.

[245] Security Week. *Microsoft Azure Flaws Exposed RHEL Instances.* `https://www.securityweek.com/microsoft-azure-flaws-exposed-rhel-instances`.

[246] Jaebaek Seo, Byounyoung Lee, Seongmin Kim, Ming-Wei Shih, Insik Shin, Dongsu Han, and Taesoo Kim. SGX-Shield: Enabling address space layout randomization for SGX programs. In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA*, 2017.

[247] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 552–561. ACM, 2007.

[248] Zhiming Shen, Sethuraman Subbiah, Xiaohui Gu, and John Wilkes. Cloudscale: elastic resource scaling for multi-tenant cloud systems. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, page 5. ACM, 2011.

[249] Zhuojia Shen, Jie Zhou, Divya Ojha, and John Criswell. Restricting control flow during speculative execution with venkman. *arXiv preprint arXiv:1903.10651*, 2019.

[250] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. T-SGX: Eradicating controlled-channel attacks against enclave programs. In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA*, 2017.

[251] Shweta Shinde, Zheng Leong Chua, Viswesh Narayanan, and Prateek Saxena. Preventing page faults from telling your secrets. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, pages 317–328. ACM, 2016.

[252] Livio Soares and Michael Stumm. FlexSC: Flexible System Call Scheduling with Exception-Less System Calls. In *Proc. of OSDI*, 2010.

[253] SPEC. Standard Performance Evaluation Corporation SPEC CPU 2017. https://www.spec.org/cpu2017/.

[254] SQLite. *LSM Benchmarks*. https://sqlite.org/src4/doc/trunk/www/lsmperf.wiki.

[255] SQLite. *Small. Fast. Reliable. Choose any three.* https://sqlite.org/index.html.

[256] Julian Stecklina and Thomas Prescher. LazyFP: Leaking FPU Register State using Microarchitectural Side-Channels. *arXiv preprint arXiv:1806.07480*, 2018.

[257] Stratoscale. *How to efficiently create a Snapshot for Crash Consistency*. https://www.stratoscale.com/blog/storage/how-to-efficiently-create-a-snapshot-for-crash-consistency/.

[258] Sumologic. *Biggest AWS Security Breaches of 2017*. https://www.sumologic.com/blog/security/aws-security-breaches-2017/.

[259] Sungard. *The Difference Between Public and Private Cloud*. https://www.sungardas.com/en/about/resources/articles/difference-public-private-cloud/?PD_OmnitureChannel=ps&gclid=Cj0KCQjw-uzVBRDkARIsALkZAdnDeuNqMqjZbl8wjd4akYifkfGZHxPqV4tz
wcB.

[260] Mohammadkazem Taram, Ashish Venkat, and Dean Tullsen. Context-sensitive fencing: Securing speculative execution via microcode customization. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019.

[261] Thales. *Thales nShield Connect Series TECHNICAL SPECIFICATIONS*. http://images.go.thales-esecurity.com/Web/ThalesEsecurity/%7B418634b6-438b-4803-9218-f688572d4f11%7D_nShield_Connect_ds.pdf.

[262] Threatpost. *Permissions Flaw Found on Azure AD Connect*. https://threatpost.com/permissions-flaw-found-azure-ad-connect/129170/.

[263] Hongliang Tian, Qiong Zhang, Shoumeng Yan, Alex Rudnitsky, Liron Shacham, Ron Yariv, and Noam Milshten. Switchless calls made practical in intel sgx. In *Proceedings of the 3rd Workshop on System Software for Trusted Execution*, pages 22–27. ACM, 2018.

[264] Chia-Che Tsai, Donald E Porter, and Mona Vij. Graphene-sgx: A practical library os for unmodified applications on sgx. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, page 8, 2017.

[265] Yukiyasu Tsunoo, Etsuko Tsujihara, Kazuhiko Minematsu, and Hiroshi Hiyauchi. Cryptanalysis of block ciphers implemented on computers with cache. In *International Symposium on Information Theory and Its Applications*, Xi'an, CN, October 2002.

[266] Ubuntu. *L1 Terminal Fault (L1TF)*. https://wiki.ubuntu.com/SecurityTeam/KnowledgeBase/L1TF.

[267] Ubuntu. *Spectre And Meltdown*. https://wiki.ubuntu.com/SecurityTeam/KnowledgeBase/SpectreAndMeltdown.

[268] Ultra Electronics. *Keyper HSM Datasheet*. https://www.veritech.net/datasheets/KeyperHSM.pdf.

[269] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *Proceedings of the 27th USENIX Security Symposium. USENIX Association*, 2018.

[270] Jo Van Bulck, Frank Piessens, and Raoul Strackx. Nemesis: Studying Microarchitectural Timing Leaks in Rudimentary CPU Interrupt Logic. 2018.

[271] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution. In *Proceedings of the 26th USENIX Security Symposium*. USENIX Association, 2017.

[272] Stephan van Schaik, Alyssa Milburn, Sebastian sterlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. RIDL: Rogue in-flight data load. In *S&P*, May 2019.

[273] Vish Viswanathan. Disclosure of H/W prefetcher control on some Intel processors. https://software.intel.com/en-us/articles/disclosure-of-hw-prefetcher-control-on-some-intel-processors, September 2014.

[274] VMWare. *Eight Key Ingredients for Building an Internal Cloud*. https://www.vmware.com/content/dam/digitalmarketing/vmware/en/pdf/whitepaper/cloud/eight-key-ingredients-building-internal-cloud-white-paper.pdf.

[275] William Voorsluys, James Broberg, Srikumar Venugopal, and Rajkumar Buyya. Cost of virtual machine live migration in clouds: A performance evaluation. In *IEEE International Conference on Cloud Computing*, pages 254–265. Springer, 2009.

[276] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A. Gunter. Leaky cauldron on the dark land: Understanding memory side-channel hazards in SGX. In *CCS*, pages 2421–2434. ACM, 2017.

[277] Zhenghong Wang and Ruby B Lee. Covert and side channels due to processor architecture. In *Computer Security Applications Conference, 2006. ACSAC'06. 22nd Annual*, pages 473–482. IEEE, 2006.

[278] Andrew Waterman, Yunsup Lee, David A Patterson, and Krste Asanovic. The RISC-V Instruction Set Manual, Volume I: Base user-level ISA. *EECS Department, UC Berkeley, Tech. Rep. UCB/EECS-2011-62*, 2011.

[279] Nico Weichbrodt, Anil Kurmus, Peter Pietzuch, and Rüdiger Kapitza. Asyncshock: Exploiting synchronisation bugs in intel sgx enclaves. In *European Symposium on Research in Computer Security*, pages 440–457. Springer, 2016.

[280] Samuel Weiser, Raphael Spreitzer, and Lukas Bodner. Single trace attack against RSA key generation in Intel SGX SSL. In *AsiaCCS*, 2018.

[281] Ofir Weisse, Valeria Bertacco, and Todd Austin. Regaining lost cycles with hotcalls: A fast interface for sgx secure enclaves. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pages 81–93. ACM, 2017.

[282] Ofir Weisse, Jo Van Bulck, Marina Minkin, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Raoul Strackx, Thomas F. Wenisch, and Yuval Yarom. Foreshadow-NG: Breaking the virtual memory abstraction with transient out-of-order execution. *Technical report*, 2018.

[283] Rafal Wojtczuk and Joanna Rutkowska. Attacking SMM Memory via Intel CPU Cache Poisoning. In *Invisible Things Lab*, 2009.

[284] Zhenyu Wu, Zhang Xu, and Haining Wang. Whispers in the hyper-space: High-speed covert channel attacks in the cloud. In *USENIX Security symposium*, pages 159–173, 2012.

[285] Roland E Wunderlich, Thomas F Wenisch, Babak Falsafi, and James C Hoe. SMARTS: Accelerating microarchitecture simulation via rigorous statistical sampling. In *ACM SIGARCH Computer Architecture News*, volume 31, pages 84–97. ACM, 2003.

[286] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *Proc. of IEEE S&P*, 2015.

[287] Yunjing Xu, Michael Bailey, Farnam Jahanian, Kaustubh Joshi, Matti Hiltunen, and Richard Schlichting. An exploration of l2 cache covert channels in virtualized environments. In *Proceedings of the 3rd ACM workshop on Cloud computing security workshop*, pages 29–40. ACM, 2011.

[288] Mengjia Yan, Jiho Choi, Dimitrios Skarlatos, Adam Morrison, Christopher W Fletcher, and Josep Torrellas. InvisiSpec: Making Speculative Execution Invisible in the Cache Hierarchy. In *Proceedings of the 51th International Symposium on Microarchitecture (MICRO'18)*, 2018.

[289] Mengjia Yan, Bhargava Gopireddy, Thomas Shull, and Josep Torrellas. Secure hierarchy-aware cache replacement policy (SHARP): Defending against cache-based side channel attacks. In *Computer Architecture (ISCA), 2017 ACM/IEEE 44th Annual International Symposium on*, pages 347–360. IEEE, 2017.

[290] Mengjia Yan, Yasser Shalabi, and Josep Torrellas. ReplayConfusion: detecting cache-based covert channel attacks using record and replay. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, page 39. IEEE Press, 2016.

[291] Mengjia Yan, Read Sprabery, Bhargava Gopireddy, Christopher Fletcher, Roy Campbell, and Josep Torrellas. Attack Directories, Not Caches: Side Channel Attacks in a Non-Inclusive World. In *Attack Directories, Not Caches: Side Channel Attacks in a Non-Inclusive World*, page 0. IEEE.

[292] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: a high resolution, low noise, L3 cache side-channel attack. In *USENIX Security*, pages 719–732, 2014.

[293] Yuval Yarom, Daniel Genkin, and Nadia Heninger. Cachebleed: a timing attack on openssl constant-time rsa. *Journal of Cryptographic Engineering*, 7(2):99–112, 2017.

[294] Kenneth C Yeager. The MIPS R10000 superscalar microprocessor. *IEEE micro*, 16(2):28–41, 1996.

[295] Project Zero. speculative execution, variant 4: speculative store bypass. https://bugs.chromium.org/p/project-zero/issues/detail?id=1528.

[296] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-tenant side-channel attacks in paas clouds. In *ACM Conference on Computer and Communications Security*, pages 990–1003. ACM, 2014.

[297] Wenting Zheng, Ankur Dave, Jethro Beekman, Raluca Ada Popa, Joseph Gonzalez, and Ion Stoica. Opaque: An Oblivious and Encrypted Distributed Analytics Platform. In *Proc. of NSDI*, 2017.