KU LEUVEN

ARENBERG DOCTORAL SCHOOL
Faculty of Engineering Science

Microarchitectural Side-Channel Attacks for Privileged Software Adversaries

Jo Van Bulck

Supervisor:
Prof. dr. ir. F. Piessens

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor of Engineering Science (PhD): Computer Science

September 2020

Microarchitectural Side-Channel Attacks for Privileged Software Adversaries

Jo VAN BULCK

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor of Engineering Science (PhD): Computer Science

September 2020

Preface

DEAR reader, laid out before you is the result of five years of hard work. The journey has been quite a roller coaster at times, and I am much indebted to the people who supported me along the way.

First and foremost, I would like to thank my supervisor, Frank Piessens, for being, quite frankly, the finest mentor one could wish for. Whenever in doubt, I could always count on your intelligence and commitment to guide me through this PhD. Your ability to spark enthusiasm and optimism has been truly inspiring, both as a scientist and as a person.

Second, I want to thank the members of my jury, Daniel Gruss, Thorsten Holz, Wouter Joosen, Bart Preneel, and Robert Puers, for their time and insightful comments. I feel deeply privileged to be assessed by such excellent jury members.

Next, the research described in this thesis was not performed in isolation, and over the past years, I have been fortunate to meet and collaborate with many exceptionally kind and talented people, both at home and abroad. Although it is not possible to list everyone here by name, I want to thank all of you for the discussions we had and the research excitement we shared. A special mention goes to Daniel Gruss and his team at TU Graz, including Michael Schwarz and Moritz Lipp, for the fruitful collaborations and a wonderful and hospitable research stay. Likewise, thanks to David Oswald and Flavio Garcia for the pleasant cooperation and a short yet energetic research visit at the University of Birmingham. Thanks to Daniel Moghimi for the insightful discussions and engaging collaborations. Additionally, thanks to the people at Intel who provided feedback on the conclusion chapter.

I am further grateful to everyone at DistriNet, and in particular Wouter Joosen, Annick Vandijck, Katrien Janssens, and Ghita Saevels, for creating an enjoyable working environment with the support, trust, and freedom needed to pursue ambitious research results. Thanks to Job Noorman and Jan Tobias Mühlberg for introducing me to research during my master thesis and beyond. More generally,

thanks to all the people involved with the Sancus project. Maintaining an open-source embedded enclave processor has been a superb learning experience and, above all, a lot of fun!

This PhD thesis would not have been possible without the continuous support of my family and friends. I am particularly grateful to my parents for giving me a solid foundation to face the world by prioritizing studying and exploring one's interests. Special thanks goes to my siblings for always being supportive along the way. Thanks also to all of my friends for the pleasurable conversations and beers we shared, and in particular to Kevin for his constant interest in my research and for the years we spent in Parkstraat.

The final words go to my dearest person. Vera, getting to know you has been the best thing about this PhD. Thank you for your love and your kindness, for your patience and your support, and for continuing to make me see the world in new and unexpected lights. I am excited to embark on new adventures together—поехали!

Jo Van Bulck

This work was supported by a PhD Fellowship from the Research Foundation – Flanders (FWO) and by the Research Fund KU Leuven.

Abstract

Recent developments on hardware-based trusted execution environments, such as the Software Guard Extensions (SGX) included in recent Intel x86 processors, hold the promise of securely outsourcing sensitive computations to untrusted remote platforms. The compelling aspect of these architectures is that they aim to protect small software components, called enclaves, even against a very powerful type of root adversaries that have full control over the operating system on the target device. This thesis shows, however, that the protection offered by today's trusted execution environments is not sufficiently understood and should be nuanced in terms of microarchitectural attack surface.

In the first part of this dissertation, we develop several innovative side-channel attack techniques that allow a privileged software adversary to reliably derive metadata from an enclaved execution. These results show that traditionally privileged x86 processor interfaces, such as page tables and interrupts, can be abused in new and unexpected ways to construct highly accurate side-channel oracles that reveal code and data access patterns performed by a victim enclave. In several practical attack scenarios, we furthermore demonstrate that these metadata access patterns can lead to full disclosure of application-level secrets.

In the second part, we move from metadata exposure to direct data extraction in a critical new line of transient-execution attacks. These results show that current out-of-order processors fail to safeguard enclave secrets against subtle microarchitectural leakage coming from instructions that were tentatively executed before a CPU exception is raised. Building upon these insights, we demonstrate several innovative attacks that led to a full collapse of the Intel SGX ecosystem and required extensive hardware and software updates.

We conclude this dissertation with a systematization of the last five years of SGX attacks, and we outline several promising defense avenues for next-generation hardened trusted execution architectures.

Beknopte samenvatting

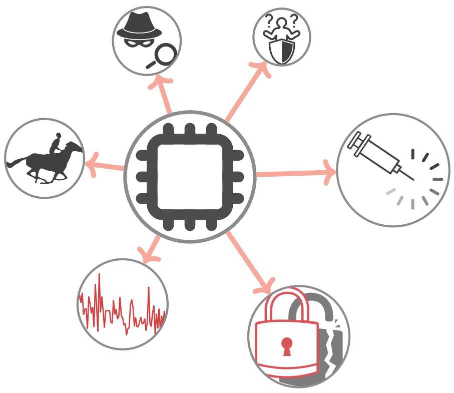
Recente ontwikkelingen op het gebied van hardware-gebaseerde vertrouwde uitvoeringsomgevingen, zoals de SGX-extensies die in recente Intel x86-processoren zijn opgenomen, houden de belofte in om op een veilige manier gevoelige berekeningen uit te besteden aan niet-vertrouwde externe platformen. Het overtuigende aspect van deze architecturen is dat ze bedoeld zijn om kleine softwarecomponenten, genaamd enclaves, te beschermen, zelfs tegen een zeer krachtig type tegenstanders die volledige controle hebben over het besturingssysteem op het doelapparaat. Deze thesis toont echter aan dat de bescherming geboden door hedendaagse vertrouwde uitvoeringsomgevingen niet voldoende begrepen wordt en meer bepaald moet worden genuanceerd in termen van microarchitecturale aanvallen.

In het eerste deel van dit proefschrift ontwikkelen we verschillende innovatieve side-channel aanvalstechnieken die een geprivilegieerde software-tegenstander in staat stellen om op betrouwbare wijze metadata af te leiden tijdens de uitvoering van een enclave. Deze resultaten laten zien dat traditioneel geprivilegieerde x86-processor interfaces, zoals paginatabellen en interrupts, op nieuwe en onverwachte manieren kunnen worden misbruikt om zeer nauwkeurige side-channel orakels te construeren die de code- en gegevenstoegangspatronen van een getroffen enclave aan het licht brengen. In verschillende praktische aanvalsscenario's tonen we bovendien aan dat deze toegangspatronen kunnen leiden tot de onthulling van gevoelige gegevens op het niveau van de applicatie.

In het tweede deel gaan we over naar directe gegevensextractie in een kritische nieuwe lijn van transitieve uitvoeringsaanvallen. Deze resultaten tonen aan dat huidige out-of-order processoren er niet in slagen om enclave data te beschermen tegen subtiele microarchitecturale lekken afkomstig van instructies die tentatief werden uitgevoerd alvorens een exceptie op te roepen. Voortbouwend op deze inzichten demonstreren we verschillende innovatieve aanvallen die hebben geleid tot een volledige ineenstorting van het Intel SGX-ecosysteem en die vergaande hardware- en software-updates vereisen.

We sluiten deze dissertatie af met een systematisering van de laatste vijf jaar van SGX-aanvallen, en we schetsen verschillende veelbelovende verdedigingstechnieken voor de volgende generatie van versterkte beveiligingsarchitecturen.

Contents



List of figures

List of tables

List of acronyms

ABI	Application Binary Interface
AEP	Asynchronous Exit Pointer
AEX	Asynchronous Enclave Exit
API	Application Programming Interface
APIC	Advanced Programmable Interrupt Controller
ASLR	Address Space Layout Randomization
BSL	Bootstrap Loader
BTB	Branch Target Buffer
DMA	Direct Memory Access
EDL	Enclave Description Language
EPC	Enclave Page Cache
EPCM	Enclave Page Cache Map
EPID	Enhanced Privacy Identifier
EPT	Extended Page Table
IDT	Interrupt Descriptor Table
IPI	Inter Processor Interrupt
IRQ	Interrupt Request
ISA	Instruction Set Architecture
ISR	Interrupt Service Routine

L1TF	L1 Terminal Fault
LE	Launch Enclave
LVI	Load Value Injection
MAC	Message Authentication Code
MDS	Microarchitectural Data Sampling
MEE	Memory Encryption Engine
MMIO	Memory Mapped I/O
MMU	Memory Management Unit
MSR	Model Specific Register
OE	Open Enclave
OS	Operating System
PMC	Performance Monitoring Counter
PRM	Processor Reserved Memory
PTE	Page Table Entry
QE	Quoting Enclave
ROB	Reorder Buffer
RPC	Remote Procedure Call
RSB	Return Stack Buffer
SDK	Software Development Kit
SEV	Secure Encrypted Virtualization
SGX	Software Guard Extensions
SMAP	Supervisor Mode Access Prevention
SMEP	Supervisor Mode Execution Prevention
SMT	Simultaneous Multithreading
SSA	State Save Area
TCB	Trusted Computing Base

TEE	Trusted Execution Environment
TLB	Translation Lookaside Buffer
TRTS	Trusted Runtime System
TSC	Time Stamp Counter
TSX	Transactional Synchronization Extensions
VMM	Virtual Machine Monitor

Chapter 1

Introduction

"It's a terrifying moment. And then when I start, I'm always amazed—so that wasn't so bad."

— *Frank Gehry (Sketches of Frank Gehry, 2006)*

In today's increasingly connected world, we interface with software systems on a daily basis. These interactions reach from entrusting valuable personal data to our phones or computers, to the critical infrastructure that keeps our society afloat—think about communications networks, data centers, cloud systems, up to the power grid and hospital equipment. Not only do we entrust sheer amounts of personal data to these systems, their complexity has also grown to proportions that exceed our levels of understanding. Consider a sizable modern web application, for instance, which is commonly interpreted by a highly optimized JavaScript engine, embedded inside a multi-layered web browser application, supported by a monolithic operating system kernel, potentially managed by a virtual machine monitor, all of this on top of a modern out-of-order CPU which itself is decomposed into several layers of microcode and silicon. Alarmingly, each of these components is designed by error-prone humans, and a single vulnerability in one of the critical lower layers may jeopardize the security of the system as a whole. The Linux kernel, for instance, recently exceeded 27.8 million lines of code, more than a doubling over the last decade, and modern cars are estimated to even have over 100 million lines of code [172].

An important research question in this respect is how to isolate software components from mutually distrusting parties executing on the same platform.

To this end, processor vendors have developed several effective memory isolation hardware mechanisms over the past decades, most notably virtual memory and privilege rings, which allow trusted system software to govern untrusted applications running at higher levels in the system stack. More recently, in light of the steady stream of vulnerabilities in mainstream operating systems, *enclaves* have been proposed as a promising new trusted execution paradigm that anchors security primitives directly in the processor, without having to trust any of the intermediate software layers. This thesis, however, develops several novel attack techniques to compromise enclave security on commodity Intel x86 platforms. Our findings show that, while enclaves hold the compelling potential of securely offloading sensitive computations to untrusted remote platforms, their security limitations in terms of microarchitectural attack surface are not sufficiently understood. This thesis contributes to building a systematic understanding of enclave limitations, which will ultimately be instrumental to the success of emerging trusted execution technology.

1.1 The need for trust

To enforce security guarantees, a computer system should be able to protect the internal state of a running software entity against other potentially malicious entities. This requirement is commonly referred to as *software isolation*. For decades, software isolation has been one of the key principles underpinning secure system design, and widespread solutions have been established in commodity computing platforms.

Memory isolation is traditionally rooted in the processor, by extending the hardware-software contract with a notion of privilege rings and address translation that allow to separate a privileged Operating System (OS) and different applications in their respective security domains [240, 114]. Virtual memory techniques introduce a level of indirection between the virtual addresses used by programs and the actual locations in physical memory. A dedicated Memory Management Unit (MMU) hardware component is responsible to automate the virtual-to-physical address mapping by consulting an in-memory page-table tree data structure configured by the trusted operating system. Conventional operating systems allocate separate page tables per process and update an MMU register that points to the root of the current page-table tree on context switch. Beyond the convenience of simulating a memory space much larger than the system's available physical memory, virtual address spaces provide the basis for software security. Specifically, since each process has its own virtual-to-physical address mapping, one process cannot access the physical memory assigned to another process, unless such a mapping is explicitly created

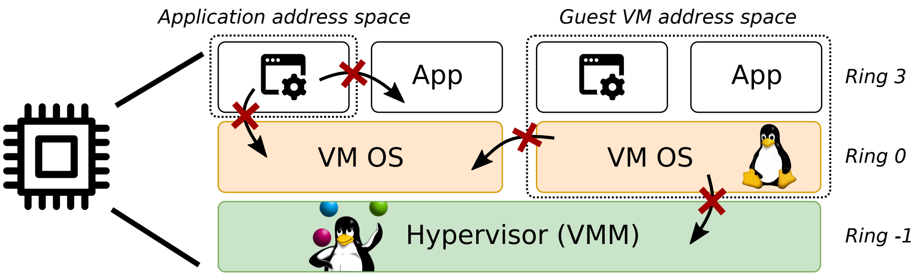


Figure 1.1: Overview of protection rings and software isolation in modern Intel x86 processors. Application software executes in user space (ring 3) in a separate virtual address space setup by the trusted operating system, which is itself isolated as a privileged program in kernel space (ring 0). Additionally, in a virtualization environment, the operating system and its applications may be further confined in a guest-physical address space managed by an even more privileged hypervisor (ring -1).

to realize shared memory. Memory protection guarantees via private address spaces of course only hold if user programs are prevented from compromising the OS, or from changing the page tables themselves. Modern processors, therefore, additionally enforce protection rings that allow the OS kernel to run more privileged, as illustrated in Fig. 1.1. Regular user programs can request services from the privileged OS through system calls, which switch the processor into kernel mode and start executing the appropriate handler code. More recent processors, with hardware extensions for virtualization, furthermore offer an extra privilege level to support a hypervisor or Virtual Machine Monitor (VMM) software layer. Similar to traditional software isolation for user applications, VMMs confine an untrusted operating system via an additional level of hardware-backed address translation, this time configured through extended page tables that map guest-physical addresses onto the actual machine's host-physical address space.

Already since the early days of computing, the traditionally layered approach to software isolation via virtual memory and privilege rings has been criticized from a security perspective [60, 240, 159]. In this respect, a particularly relevant metric when reasoning about the security of a system is the Trusted Computing Base (TCB), which denotes the set of hardware and software components that need to be trusted in order to ensure the correct execution of a software program. Since the privileged operating system kernel traditionally belongs to the TCB of any user program, a single kernel vulnerability can often lead to a complete collapse of all security guarantees in the entire system. Over the past decades, considerable research efforts have focused on making operating systems more trustworthy, exemplified through, for instance, the extensive and ongoing line

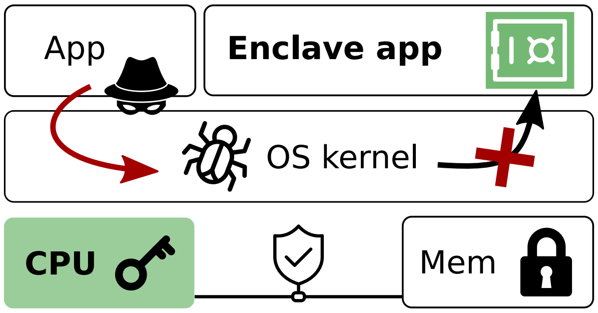


Figure 1.2: Trusted execution environments, like Intel SGX, offer hardware-level isolation and attestation for unprivileged software components, called *enclaves*, using a cryptographic key that never leaves the CPU package. The processor furthermore prevents unauthorized accesses to enclave-private memory by *any* outside software, including traditionally privileged operating system or hypervisor layers.

of work on microkernels [159, 142, 57]. However, as mentioned in the beginning of this chapter, the tendency is for real-world software systems to grow only more complex, and critical vulnerabilities continue to be found in today's widely deployed operating systems [41].

In response to these challenges, recent efforts from both academia [174, 173, 238, 191, 100, 147, 48] and industry [10, 136, 176, 14, 11] have developed Trusted Execution Environments (TEEs) that support the secure and isolated execution of critical application compartments, called *enclaves*, with a minimal TCB. From a high level, as illustrated in Fig. 1.2, TEEs provide enclaves with strong confidentiality and integrity guarantees regarding their internal state, without having to trust *any* other software executing on the platform, including traditionally privileged operating system or hypervisor code.[1] To this end, TEEs enforce rigid access control on enclave secrets while they reside inside the processor and transparently encrypt and integrity protect all enclave memory while it resides in untrusted DRAM outside the processor. Besides strong memory isolation, TEEs typically offer an *attestation* primitive that allows local or remote stakeholders to cryptographically verify at runtime that a specific enclave has been loaded on a genuine TEE processor, without having been tampered with. By embedding a root-of-trust directly in the processor, TEEs drastically reduce the trusted computing base, to the point where application developers solely rely on the correctness of the CPU and the implementation of

[1]We note that some TEE designs, like ARM TrustZone [205] or RISC-V Keystone [153], do rely on a small trusted software layer to manage enclave transitions. However, this thesis has an explicit focus on hardware-only TEEs, like Intel SGX [176] or Sancus [189].

their own enclaves. Enclaved execution hence holds the promise of enforcing strong security and privacy requirements for local and remote computations.

As a relevant instance of a real-world TEE, recent Intel x86 processors from 2015 onwards ship with Software Guard Extensions (SGX) [176, 14] which bring strong, hardware-enforced security guarantees to commodity computing devices, while remaining backwards compatible with legacy virtual memory and privilege rings. Particularly, SGX enclaves live in the virtual address space of a conventional user-space host application. While the untrusted operating system remains in charge of resource management and memory mappings, SGX enclaves are directly protected and measured by the processor itself. For this, the address translation logic is extended with an additional level of SGX sanitization checks that block any unauthorized access to enclave memory regardless of the current CPU privilege level. SGX furthermore comes with several new x86 instructions to create and destroy enclaves, to switch the processor in and out of enclave mode, and to aid secure cryptographic key derivations.

1.2 Software-based microarchitectural attacks

The protection model envisioned by TEEs, such as Intel SGX, outlines clear-cut architectural isolation between an enclave and its untrusted surroundings: no outside code, at any privilege level, is ever allowed to access enclave-private memory. While such rigid enclave encapsulation is well-understood in a mathematical sense, including even successful formal modeling and verification efforts [62, 194, 203], this section explains that architectural protection boundaries in real-world processors are not absolute and should be nuanced in terms of side channels.

Modern processors are exceptionally complex pieces of engineering comprising several abstraction layers. The highest abstraction layer, the Instruction Set Architecture (ISA), describes the processor's hardware-software contract, including the supported instructions, the available registers and addressing modes, and the execution semantics. Modern ISAs furthermore specify several important security mechanisms to configure access restrictions for the software running on top, e.g., virtual memory, privilege rings, and enclaves. The processor's microarchitecture then describes how exactly the ISA is implemented in terms of, for instance, the CPU pipeline, the execution units, the memory hierarchy, and the interconnection between these elements. In contrast to the ISA specification, microarchitectural implementation details are generally ill-documented as they are considered important intellectual property. By shielding off microarchitectural implementation aspects, the ISA-level abstraction has furthermore proven to be a crucial enabler for processor extensibility and performance gains over the last decades. Consider the popular x86 ISA, for instance, where several competing vendors, including Intel and AMD, have developed a succession of ever-faster microarchitectures while maintaining strict backwards compatibility with the original x86 software specification.

Abstraction levels are only relative in the eyes of attackers, however, and a long line of research has shown that unconstrained microarchitectural optimizations can be abused to bypass security restrictions imposed at the ISA level. Particularly, by exploiting timing or other observable side effects of execution, confined adversaries can infer secret-dependent traces left in the shared microarchitectural state during victim execution. This class of software-based microarchitectural side-channel attacks has been recognized for over two decades and has received growing attention from the research community [71]. More recently, in early 2018, the disruptive real-world impact of microarchitectural side channels became acutely evident when they were used as building blocks for the high-profile Meltdown [162], Spectre [146], and Foreshadow [249] line of transient-execution attacks.

In the following, we first overview how execution metadata can be reconstructed through microarchitectural side-channel analysis and thereafter zoom in on the threat of direct data leakage through transient execution. Note that this section only serves to introduce the main concepts, and we defer a systematic overview of microarchitectural attacks and the various ways in which they have been abused to dismantle enclave isolation to Chapter 8.

1.2.1 Timing side-channel attacks

Software-based microarchitectural side channels commonly rely on contention in a shared microarchitectural element, e.g., branch predictors or caches. Generally, adversaries proceed by first initializing the shared microarchitectural element in a known state and subsequently measuring state changes during or after victim execution. The observed microarchitectural state updates allow to gain insight into the victim's behavior, even when attackers are strictly isolated at the architectural level and are supposedly only allowed to interact with the victim in a black-box style, via well-defined input and output channels.

Hence, in order to mount a successful side-channel attack, adversaries need some way to gain insight into the underlying microarchitectural behavior, which, however, is designed to be transparent to the programmer. Depending on the attacker's capabilities, some insights can be derived via, for instance, transactional memory aborts [54] or performance monitoring counters [79, 29, 156], but by far the most common way to observe microarchitectural state changes is through timing [71]. That is, whenever microarchitectural optimizations rely on global stateful elements, such as translation-lookaside buffers, caches, or branch predictors, any updates to these elements during victim execution will cause measurable timing differences in the attacker domain. Perhaps the most widely studied subcategory of microarchitectural timing side channels are cache timing attacks [202, 280, 88, 87]. These attacks exploit that the time to load a memory location depends on whether or not that location was recently accessed and brought into the CPU cache. The basic idea behind caching is to speed up repeated accesses to the same or neighboring memory locations by maintaining local copies of the recently accessed data near the processor, through an intricate hierarchy of hidden cache levels. Caching has become an indispensable performance optimization for modern processors, which can process data several orders of magnitude faster than it can be fetched from DRAM. However, the crucial role of cache memories to maintain performance in today's processors also makes them prime candidates for side-channel analysis for at least two reasons. First, to avoid costly flushes or partitioning, caches are commonly shared across protection domains and privilege levels. Second, the high latency of DRAM transactions makes cache misses distinctly observable, with timing penalties of up to hundreds of CPU cycles.

As an exemplary cache timing attack, Flush+Reload [280] has become a standard method in recent years. This attack technique is conceptually very simple, yet has proven to be extremely powerful and versatile to mount high-resolution and low-noise cache timing attacks in practice. Flush+Reload has, for instance, recently been leveraged as one of the building blocks for transient-execution attacks, including Meltdown [162], Spectre [146], Foreshadow [249], and LVI [251]. Assuming that the attacker and the victim share some read-only memory locations, e.g., as is commonly the case for shared library code, Fig. 1.3 illustrates how the Flush+Reload attack proceeds in three phases to determine whether a victim program accessed a shared memory location A:

1. **Flush.** The attacker first initializes the shared microarchitectural CPU cache in a known state by explicitly flushing the memory location A from the cache. This phase ensures that any subsequent access to address A will suffer a cache miss and will hence have to be served from memory.

2. **Victim execution.** Now the attacker waits for the victim to execute. In our example code, the victim only accesses A if a certain application-level secret is true. In this case, the victim's access to A causes the processor to initiate a memory transaction and to cache the result, speeding up future references to A. If the secret is false, on the other hand, the victim loads some other memory location B into the cache.

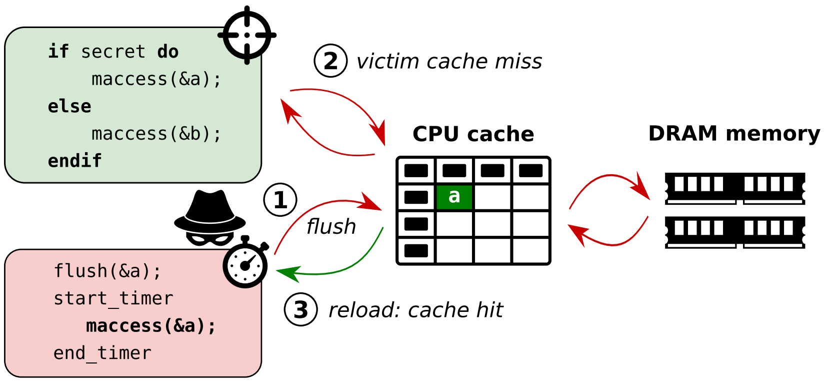


Figure 1.3: Overview of a Flush+Reload side-channel attack to learn whether a victim program accessed a shared memory region in three phases: (1) the attacker flushes the shared memory region from the CPU cache; (2) the victim program conditionally accesses the shared memory region, depending on some secret; (3) the attacker learns whether the victim brought the shared memory location into the CPU cache by measuring the amount of time it takes to reload that location.

3. **Reload.** After completing the victim execution, the attacker carefully measures the amount of time it takes to reload the shared memory location A. If the access is fast, the attacker learns that the application-level secret is true, as the victim's execution must have brought A into the shared CPU cache in step 2 above. In case of a slow access, on the other hand, A was not accessed during the victim's execution and the attacker concludes that the secret was false.

The above attack procedure serves as a minimal, yet clear example of how unconstrained optimizations at the microarchitectural level can break security objectives imposed at higher levels. Particularly, the only requirements for the Flush+Reload attack is that the processor features a cache and that attacker and the victim share some read-only memory mapping. At the architectural level, both processes can be fully isolated in their own separate virtual address spaces or privilege rings. In our example, the read-only restriction for the shared memory region implies that the attacker and victim have no direct communication channel and are conceptually isolated from each other. However, despite these strict architectural restrictions, the attacker and the victim still implicitly share the underlying CPU cache at the microarchitectural level. By resorting to subtle timing differences, attackers can learn execution metadata in terms of the trace of memory addresses accessed by the victim. A special use case of side channels, where the attacker controls both the sender and receiver processes, is referred to as a "covert channel" and allows to setup hidden communication channels to bypass architectural information-flow restrictions.

Note that we focused on Flush+Reload in the above example for its simplicity, but more advanced cache timing attacks based on the Prime+Probe technique [202, 225], for instance, leverage similar procedures to establish memory access patterns across arbitrary protection domains, without requiring any form of shared memory. These attacks only assume a shared CPU cache and have even been demonstrated in fully isolated virtualization environments [171].

1.2.2 Transient-execution attacks

Microarchitectural timing side-channel attacks, as outlined in the previous section, have traditionally been strictly limited to leaking execution metadata, e.g., the addresses of code or data accesses in a victim program. The preferred way to defend against these attacks is to adopt a "constant-time" programming model [44, 71], which ensures that the victim's code and data memory access patterns never depend on application secrets. For a long time, the side-channel research landscape has been characterized by an ongoing cat-and-mouse game, where attackers developed ever more accurate techniques to leak execution metadata and reconstruct cryptographic keys, prompting developers to refine constant-time code hardening patches in the affected libraries. This fundamentally changed in early 2018, however, when several researchers [67, 101, 146, 162, 249] independently discovered that side channels can also be abused in an entirely different way to reconstruct secret-dependent traces left in the CPU's microarchitectural state following branch mispredictions or exceptions.

Intuitively, this new class of transient-execution attacks[2] broadens the threat of microarchitectural side channels from relatively harmless metadata leakage to direct data extraction of arbitrary victim secrets. We first overview CPU pipeline concepts and outline the abstract phases of a transient-execution attack, and we thereafter elaborate on the distinction between Spectre-type [146] and Meltdown-type [162] attack categories.

[2]The term "transient execution" was originally coined in the Meltdown [162] and Spectre [146] papers as an umbrella term to avoid confusion with the established concept of "speculative execution", which suggests a notion of optimistic prediction that does, however, not clearly apply to attacks based on exception deferral. With Foreshadow, presented in Chapter 6, and the subsequent transient systematization [36], we were among the early adopters of the term and helped its spreading throughout the research community. More recently, in the aftermath of our research on LVI, presented in Chapter 7, Intel [125] decided to officially adopt transient-execution terminology in order to more accurately distinguish between Spectre-type and Meltdown-type threats.

Basic idea. Modern CPU pipelines are massively parallelized allowing hardware logic in prior pipeline stages to perform operations for subsequent instructions ahead of time or even out-of-order. Intuitively, however, pipelines may stall when operations have a dependency on a previous instruction which has not been executed and retired yet. Hence, to keep the pipeline full at all times, it is essential to predict upcoming control flow decisions and data dependencies. Modern processors, therefore, rely on intricate microarchitectural optimizations to predict and sometimes even re-order the instruction stream. Crucially, as these predictions may turn out to be wrong, computation results should only be committed to the CPU's architectural state according to the intended in-order instruction stream specified by the programmer. Pipeline flushes may therefore be necessary to recover from wrong predictions, or in case of an exception or external interrupt request. The pipeline flush ensures functional correctness by discarding any architectural effects of pending instructions that follow the exception or misprediction event. Hence, these instructions are only executed "transiently": first they are, and then they vanish.

Transient instructions reflect unauthorized computations out of the program's intended code or data paths. For functional correctness, it is crucial that transient computation results are never committed to the architectural state. However, transient instructions may still leave secret-dependent traces in the CPU's microarchitectural state, which can later be recovered through side-channel analysis. This observation has led to a proliferation of transient-execution attacks [162, 146, 249, 36, 216, 223, 35, 251], which from a high-level always follow the same abstract flow, as shown in Fig. 1.4. The attacker first brings the microarchitecture into the desired state by flushing or populating internal branch predictors and data caches. Next is the execution of a so-called "trigger instruction". This can be any instruction that causes subsequent operations to be eventually squashed due to, for instance, an exception or a mispredicted branch or data dependency. However, before completion of the trigger instruction, the processor proceeds with the execution of a transient instruction sequence. The attacker abuses the unintended transient instructions to act as the sending end of a microarchitectural covert channel, e.g., by loading a secret-dependent memory location into the CPU cache. Ultimately, at retirement of the trigger instruction, the processor discovers the exception or misprediction event and flushes the pipeline to discard any architectural effects of the transient execution. However, in the final phase of the attack, unauthorized transient computation results are recovered at the receiving end of the covert channel, e.g., by timing memory accesses to deduce the secret-dependent loads from the transient instructions, as in the FLUSH+RELOAD [280] procedure introduced in the previous section.

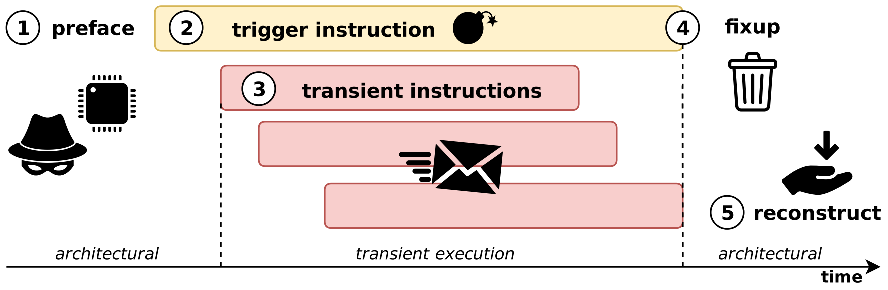


Figure 1.4: High-level overview of a transient-execution attack in 5 phases: (1) prepare microarchitecture, (2) execute a trigger instruction, (3) transient instructions encode unauthorized data through a microarchitectural covert channel, (4) CPU retires trigger instruction and flushes transient instructions, (5) reconstruct secret from microarchitectural state.

High-level classification: Spectre vs. Meltdown. All transient-execution attacks have in common that they abuse transient instructions, whose results are never architecturally committed, to encode unauthorized data in the microarchitectural CPU state. With different instantiations of the abstract phases in Fig. 1.4, a wide spectrum of transient-execution attack variants emerges [36]. While some instantiations may be rather straightforward, e.g., plugging in another type of covert channel to transmit secrets between phases 3 and 5, a more fundamental distinction relates to the nature of the trigger instruction in phase 2. In fact, on the basis of this distinction, the transient-execution attack landscape can be separated into two groups named after the initial attack in each category. Spectre-type attacks exploit transient execution following a control or data flow misprediction, whereas Meltdown-type attacks exploit illegal transient data flow following a faulting or assisted load instruction.

Intuitively, Spectre-type attacks [146, 148, 167, 23, 102] trick a victim program into transiently diverting from its intended execution path. By poisoning the processor's branch predictor machinery, for instance, Spectre adversaries craftily steer the victim's transient execution to selected "gadget" code snippets, which may inadvertently expose secrets through the shared microarchitectural state. Much like in a confused-deputy scenario, Spectre gadgets execute entirely within the victim domain and can hence transiently encode memory locations the victim is authorized to access but the attacker not.

The perpendicular category of Meltdown-type attacks [162, 249, 234, 223, 216, 35], on the other hand, target architecturally inaccessible data by exploiting illegal data flow from faulting or assisted instructions. Particularly, on vulnerable processors, the results of unauthorized loads are still forwarded to subsequent transient operations, which may encode the data before an exception or microcode assist is eventually raised. Note that, while Meltdown-type attacks so far exploit out-of-order execution, even elementary in-order pipelines may allow for similar effects [256]. Unlike Spectre-type attacks, a Meltdown attacker in one security domain can directly exfiltrate architecturally inaccessible data belonging to another domain, e.g., kernel or enclave memory. Meltdown-type attacks have, therefore, been denounced to effectively "melt down" architectural isolation barriers. However, our research on LVI [251] showed that Meltdown-type incorrect transient forwarding effects can also be inversely exploited as an injection primitive to arbitrarily hijack transient execution in a victim domain. Intuitively, LVI combines Spectre-style confused-deputy code gadgets in the victim application with Meltdown-type illegal data flow from faulting or assisted memory load instructions to bypass existing defenses and inject attacker-controlled data into a victim's transient execution.

Importantly, Spectre and Meltdown exploit fundamentally different CPU properties and hence require largely orthogonal defenses. Where the former relies on indispensable performance optimizations via dedicated control or data flow prediction machinery, the latter exploits that faulting load instructions on vulnerable processors may forward incorrect data to transient instructions ahead in the pipeline. In other words, Spectre-type leakage remains largely an unintended consequence of important speculative performance optimizations that are here to stay, whereas Meltdown reflects a failure of certain processors to respect hardware-level protection boundaries for transient instructions. Overall, mitigating Spectre should be considered an industry-wide, long-term challenge that requires careful hardware-software co-design. Meltdown-type effects, on the other hand, are generally more alarming but fortunately remain limited to a subset of today's processors and will expectedly be eradicated on the medium term through silicon-level changes that properly block any data flow from faulting or assisted load instructions.

1.3 Dissertation scope and contributions

In light of the strengthened adversary model pursued by trusted execution environments, such as Intel SGX, the key research question we ask in this dissertation is which new kinds of attack vectors can be exploited by TEE adversaries. In contrast to a concurrent line of research [181, 225, 156, 154] which focuses on developing improved exploitation techniques for conventional side-channel or memory-safety vulnerabilities in a TEE setting, this dissertation has an explicit aim at uncovering fundamentally *new* attack surfaces that were never before considered relevant in a conventional, unprivileged attacker model. We

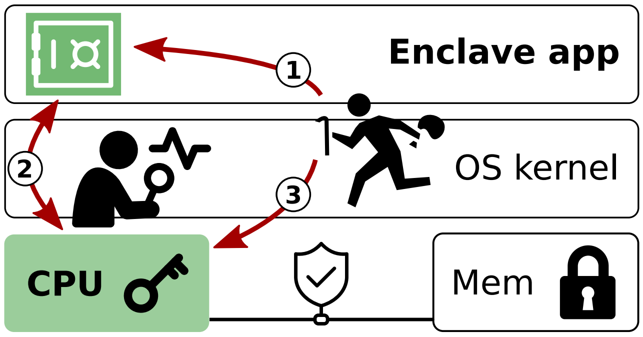


Figure 1.5: Three types of enclave interactions for privileged adversaries: (1) pass attacker-controlled arguments through the enclave interface; (2) derive execution metadata during or after enclave invocation through side channels; (3) extract enclave secrets from the CPU's microarchitectural state through transient execution.

focus our attention on the Intel SGX [14, 176] TEE, given its real-world relevance and widespread deployment in commodity Intel x86 processors. The inclusion of SGX in recent Intel processors has been broadly acclaimed for bringing strong hardware-enforced trusted computing guarantees to mass consumer devices, and for protecting end user data in an untrusted cloud environment. However, ever since its public release in 2015, Intel SGX has also sparked an ongoing line of attack research that aims to build a better understanding of the security limitations of this technology. A reoccurring element in most of these attacks, including the ones presented in this dissertation, is that they commonly take advantage of the attacker's control over the untrusted operating system. This dissertation develops novel attack techniques that abuse traditionally privileged x86 processor features, like paging and interrupts, to mount new and unexpected types of side-channel or transient-execution attacks against SGX enclaves.

Figure 1.5 summarizes three attack avenues for privileged adversaries explored in this dissertation and further described below. At an abstract level, attackers can pass rogue arguments through the enclave interface, observe execution metadata through side channels, or transiently encode secrets in the CPU's microarchitectural state. Several of these interactions can be combined, e.g., LVI attacks (Chapter 7) pass attacker-controlled data through the enclave interface so as to hijack transient execution inside the enclave and encode secrets, which can later be recovered through side-channel analysis. We further note that a relatively small subset of SGX attacks [225, 7, 24] have also been demonstrated for significantly less privileged adversaries, *i.e.*, with just user-level privileges to invoke the enclave. While this is also the case for some of the attack variants [254, 249] presented in Chapters 2 and 6, the overall scope of this dissertation has a clear focus on uncovering fundamentally new attack surfaces that arise from SGX's privileged adversary position.

1.3.1 Breaking memory safety through enclave interfaces

As a first contribution, Chapter 2 nuances the protection offered by TEEs by zooming in on the enclave software itself. That is, even if the processor is trusted and properly prevents any outside software from reading or writing enclave memory, all software executing inside the enclave protection domain still has unrestricted access to secrets and hence forms part of the TCB. Any exploitable vulnerabilities in the enclave application would compromise the security guarantees pursued by trusted execution. Apart from well-understood software vulnerabilities like buffer overflows, which can be prevented using safe programming languages [72, 68], the unique protection model offered by TEEs also requires dealing with untrusted pointers or CPU state inside the host application. TEE development environments therefore commonly include a small trusted *shielding runtime* which intervenes on enclave context switches so as to transparently maintain a secure interface between the enclave application and its hostile environment. However, in a systematic study of 8 major open-source TEE runtimes, we find that enclave shielding responsibilities are generally not well-understood and we uncover a large and re-occurring vulnerability landscape. For several of these runtimes, including production-quality SGX SDKs maintained by Intel and Microsoft, we develop proof-of-concept[3] exploits demonstrating control flow hijacking, read or write primitives, and innovative side-channel oracles caused by inadequate interface sanitization.

Our work draws lessons for TEE design and provides a foundation to methodologically reason about enclave shielding responsibilities and pitfalls. We show that Intel SGX's design decision to model enclave interactions after conventional user-to-kernel context switches also inherits many of the security implications that arise from dealing with low-level CPU register state and passing pointers in a shared address space. Even memory-safe languages cannot fully protect against insufficient interface sanitization in the enclave trusted runtime. Our findings emphasize the need to research more principled interface hardening approaches and raise concerns about trusting sizable enclave software layers with several thousands of lines of code [246, 17].

Publication data: J. Van Bulck, D. Oswald, E. Marin, A. Aldoseri, F. D. Garcia, and F. Piessens. "A Tale of Two Worlds: Assessing the Vulnerability of Enclave Shielding Runtimes". In: *26th ACM Conference on Computer and*

[3]Open-sourced at `https://github.com/jovanbulck/0xbadc0de`.

Communications Security (CCS). Nov. 2019, pp. 1741–1758

1.3.2 Leaking enclave access patterns through side channels

As a second contribution, Chapters 3 to 5 introduce a series of privileged side-channel attack techniques to reconstruct memory accesses and instructions executed in a victim enclave. These attacks demonstrate that even when TEEs properly safeguard enclave applications against direct *data* extraction, kernel-level adversaries may still monitor various side effects of the enclaved execution to reliably derive *metadata*, e.g., instruction types and partial addresses of enclave-private code or data accesses. We show in several case-study attacks that such side-channel observations can effectively lead to full recovery of application-level secrets when the victim enclave contains secret-dependent code or data accesses. Fully mitigating against this class of attacks requires enclave developers to write constant-time code [119], something that has proven to be notoriously hard over the past decade for cryptographic libraries [71]. Moreover, several studies [277, 29, 256, 92] have shown that in the case of enclave applications, sensitive data may be more ill-defined, and hence harder to keep track of in constant-time code paradigms, compared to the typical cryptographic keys of side-channel analysis.

Our work debunks several misconceptions and provides valuable insights for designing effective TEE side-channel mitigations. In Chapter 3, we contribute stealthy page-table attack techniques[4] which for the first time recognize the security implications of updating "accessed" or "dirty" bits and traversing untrusted page tables in privileged operating system memory. This work highlights the deficiency of initial controlled-channel defenses [230, 229] that focus on merely detecting page-fault events and has since informed several improved mitigations [237, 198, 139, 175, 200]. With the SGX-Step[5] enclave execution control framework, presented in Chapter 4, we shift our attention to another crucial privileged x86 interface, namely interrupts. This work contributes an innovative timer interrupt technique that for the first time allows victim enclaves to be precisely single-stepped at a *maximal* temporal resolution, *i.e.*, exactly one instruction at a time. SGX-Step exports traditionally privileged operating system features, such as page tables and interrupts, via a practical user-space attack library and refines the enclaved execution threat landscape by defeating any defenses [156, 69, 119] that are based on partial atomic behavior of the instruction stream. The open-source SGX-Step framework has since been employed in our own research [91, 249, 256, 223, 35, 254, 188, 182, 251], as

[4] Open-sourced at `https://github.com/jovanbulck/sgx-pte`.

[5] Open-sourced and actively maintained at `https://github.com/jovanbulck/sgx-step`.

well as by several independent researchers [270, 6, 105, 130, 269, 8, 208, 211, 5, 94], to enable a long line of new or improved high-resolution enclave attacks. SGX-Step has furthermore guided several recent defensive works [103, 198, 9, 34, 110]. which now explicitly take single-stepping interrupt capabilities into account. Finally, Chapter 5 presents the Nemesis[6] attack, which builds on SGX-Step and for the first time shows that interrupts can not only be used to amplify the temporal resolution of other attacks, but also in themselves enable subtle microarchitectural timing leakage. Particularly, Nemesis abuses interrupt latency as an innovative side-channel attack vector to sample *individual* enclave instruction timings, allowing to reconstruct secret-dependent code paths on both commodity Intel x86 SGX platforms and embedded Sancus processors.

Publication data:

- J. Van Bulck, N. Weichbrodt, R. Kapitza, F. Piessens, and R. Strackx. "Telling your Secrets Without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution". In: *26th USENIX Security Symposium.* Aug. 2017, pp. 1041–1056
- J. Van Bulck, F. Piessens, and R. Strackx. "SGX-Step: A Practical Attack Framework for Precise Enclave Execution Control". In: *2nd Workshop on System Software for Trusted Execution (SysTEX).* ACM, Oct. 2017, 4:1–4:6
- J. Van Bulck, F. Piessens, and R. Strackx. "Nemesis: Studying Microarchitectural Timing Leaks in Rudimentary CPU Interrupt Logic". In: *25th ACM Conference on Computer and Communications Security (CCS).* Oct. 2018, pp. 178–195

1.3.3 Stealing enclave secrets with transient execution

As a final set of contributions, Chapters 6 and 7 develop novel transient-execution attack primitives that led to a complete collapse of the Intel SGX ecosystem. Both of these works revisit untrusted page-table manipulations for privileged adversaries to showcase arbitrary enclave secret extraction in the transient domain. This research nuances the trust we place in the processor itself and ultimately shows the fallacy of building a high-assurance trusted execution environment, such as Intel SGX, on top of a complex out-of-order CPU microarchitecture. That is, even if the TEE's architectural design is sound and the enclave software is free from any memory-safety or side-channel vulnerabilities, subtle oversights in the underlying processor microarchitecture can nullify all of the pursued security objectives.

[6]Open-sourced at `https://github.com/jovanbulck/nemesis`.

With the high-impact Foreshadow attack, developed concurrently to other transient-execution attacks like Spectre and Meltdown, we were among the first to discover the security implications of transient execution in modern Intel processors. Foreshadow, presented in Chapter 6, contributes an innovative page-table manipulation technique that allows to reliably extract plaintext enclave secrets from the CPU cache.[7] At its core, Foreshadow exploits an incorrect transient forwarding effect similar to Meltdown. On top of this leakage primitive, we develop a novel exploitation methodology which for the first time allows to decisively dismantle security guarantees in the Intel SGX ecosystem. We demonstrate Foreshadow's disruptive impact by extracting full cryptographic keys from Intel's vetted architectural enclaves, allowing to forge arbitrary local and remote attestation reports that are indistinguishable from those signed by a genuine Intel processor. In response to our findings, Intel initiated SGX TCB recovery [122] and released microcode updates that flush the L1 cache on every enclave transition [107]. Furthermore, following our disclosure, the underlying technique used by Foreshadow was generalized by Intel engineers to also break operating system process and even virtual machine isolation, leading to patches in all major operating system and hypervisor implementations [107, 271]. In the wider research landscape, Foreshadow led to important new insights by recognizing that Meltdown was not a one-off bug to read kernel memory in Intel processors, but instead comprises an extensive and expanding class of Meltdown-type transient-execution attacks [36, 223, 216, 35, 251].

With Load Value Injection (LVI), presented in Chapter 7, we contribute innovative techniques to reversely exploit Meltdown-type microarchitectural data leakage.[8] LVI shows that the microcode and silicon mitigations widely deployed in response to transient data extraction attacks like Foreshadow are necessary, but not sufficient. We develop novel, gadget-oriented exploitation methodologies to bypass existing SGX microcode mitigations that flush microarchitectural data buffers on enclave transitions, and we abuse privileged page-table manipulations to cause faulting or assisted loads in a victim enclaved execution. At its core, LVI abuses that such faulting or assisted loads may transiently forward dummy values or attacker-controlled data from various microarchitectural buffers to dependent instructions, before eventually being re-issued by the processor. Our research shows that potentially every illegal microarchitectural data flow can be inverted as an injection source to purposefully disrupt the victim's transient behavior. In a wider perspective, this work unifies the transient-execution research landscape by for the first time applying gadget-driven techniques from the Spectre world to reversely exploit prior Meltdown-type data leakages. LVI furthermore marks the end of transparently patching Meltdown-type processor

[7]Further information and guidance via `https://foreshadowattack.eu/`.

[8]Further information and guidance via `https://lviattack.eu/`.

vulnerabilities in CPU microcode. In response to our findings, Intel initiated SGX TCB recovery and developed expensive compiler software mitigations that insert `lfence` barrier instructions to serialize the processor pipeline after potentially *every* memory load [110].

Publication data:

- J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx. "Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution". In: *27th USENIX Security Symposium.* Aug. 2018, pp. 991–1008
- J. Van Bulck, D. Moghimi, M. Schwarz, M. Lipp, M. Minkin, D. Genkin, Y. Yuval, B. Sunar, D. Gruss, and F. Piessens. "LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection". In: *41st IEEE Symposium on Security and Privacy (S&P).* May 2020, pp. 54–72

1.3.4 Ethical considerations and reproducibility

For all of the vulnerabilities discovered during this PhD trajectory, we adhered to best practices of responsible disclosure by notifying the affected vendor well before publicly releasing our findings. When applicable, we have engaged in lengthy embargo periods which have allowed for the development of processor microcode upgrades, compiler mitigations, and software fixes.

To ensure the reproducibility of our findings and to encourage further research into effective defense mechanisms, we have released non-weaponized proof-of-concept attack code for all of the research we conducted. Some of these open-source projects, like SGX-Step, presented in Chapter 4, have gained widespread recognition in the TEE community and have informed improved defenses that take privileged adversary capabilities into account [103, 198, 9, 34, 110]. As a general guideline, for side-channel attacks falling outside of the scope of Intel SGX's official threat model [133], we released the full code for all of the experiments. For more potent transient-execution attacks, on the other hand, we took care to only release deliberately weakened proof-of-concept attacks that still allow to reproduce our findings for legitimate research purposes while avoiding targeted arbitrary secret extraction on unpatched systems.

1.4 Other contributions

The works included in this thesis are a subset of the research conducted over the course of this PhD. Particularly, this dissertation presents selected first-author

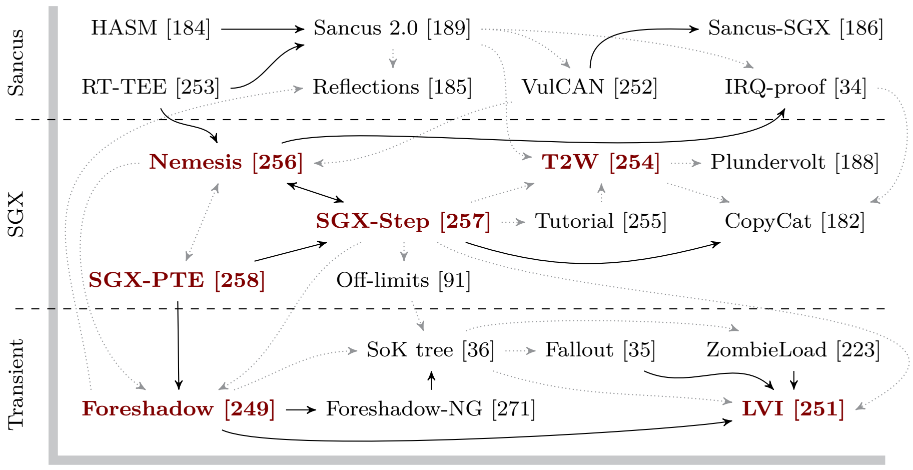


Figure 1.6: Overview of the relation between all publications. Solid arrows illustrate direct flows of ideas or techniques from one paper to another, whereas dotted arrows represent more loose influences. The highlighted papers are included in this thesis.

publications which collectively explore the security implications of privileged side-channel attacks. Other research projects that I participated in are listed below, together with a brief description of the obtained results.

Figure 1.6 illustrates the wider research context by exposing the flow of ideas that interconnects all the papers. The vertical axis shows how all of the research conducted during this PhD can be categorized into three main clusters: *(i)* a mainly defensive line of work on embedded TEE security, built around the Sancus [189] platform; *(ii)* an exploratory line of attack research developing new side-channel techniques to extract execution metadata from Intel SGX enclaves; and *(iii)* the emerging new research area on transient-execution attacks that exposes the perils of today's speculative and out-of-order processor pipelines. It should be noted that this classification is by no means absolute, as ideas freely flow between these clusters and some attacks [256, 254] have even been demonstrated on both Sancus and Intel SGX platforms.

Towards Availability and Real-Time Guarantees for Protected Module Architectures. This paper presents a design and partial implementation results for an embedded security architecture that preserves real-time availability guarantees using a combination of processor extensions for interruptible enclaves and a novel multithreading scheme implemented by an unprivileged scheduler, which itself also resides in an enclave. One of the key objectives of this design was to guarantee a strict upper bound on the worst-case interrupt response time, which consequently let to the discovery of the Nemesis [256] interrupt latency timing side channel (cf. Chapter 5) for Sancus and later also Intel SGX platforms. The interruptible Sancus-enabled MSP430 processor,[9] implemented in this work, furthermore served as a basis for several master thesis student projects.

This research was conducted in collaboration with Jan Tobias Mühlberg and Job Noorman, under the supervision of Frank Piessens.

> J. Van Bulck, J. Noorman, J. T. Mühlberg, and F. Piessens. "Towards Availability and Real-Time Guarantees for Protected Module Architectures". In: *Companion Proceedings of the 15th International Conference on Modularity (MASS).* Mar. 2016, pp. 146–151

Implementation of a High Assurance Smart Meter using Protected Module Architectures. This paper presents a case study of a distributed enclave application[10] that implements secure smart meter functionality on top of the Sancus platform [191, 189] and reports on design trade-offs and challenges for preserving real-time requirements.

The principal author of this work is Jan Tobias Mühlberg who collaborated with Sara Cleemput, Mustafa A. Mustafa, and me, under the supervision of Bart Preneel and Frank Piessens.

> J. T. Mühlberg, S. Cleemput, A. M. Mustafa, J. Van Bulck, B. Preneel, and F. Piessens. "Implementation of a High Assurance Smart Meter using Protected Module Architectures". In: *10th WISTP International Conference on Information Security Theory and Practice (WISTP).* Aug. 2016, pp. 53–69

Sancus 2.0: A Low-Cost Security Architecture for IoT Devices. This journal article describes extensions to the original Sancus [191] TEE architecture and elaborates on insights derived from several years of building extensions and case-study applications on top of Sancus.[11]

The principal author of this work is Job Noorman who collaborated with me, Jan Tobias Mühlberg, Pieter Maene, Johannes Götzfried, and Tillo Müller, under the supervision of Frank Piessens, Bart Preneel, Ingrid Verbauwhede, and Felix Freiling.

[9] Interrupt extensions were upstreamed to `https://github.com/sancus-tee/sancus-core`.
[10] Open-sourced at `https://distrinet.cs.kuleuven.be/software/sancus/wistp16/`.
[11] Open-sourced and actively maintained at `https://github.com/sancus-tee`.

> J. Noorman, J. Van Bulck, J. T. Mühlberg, F. Piessens, P. Maene, B. Preneel, I. Verbauwhede, J. Götzfried, T. Müller, and F. Freiling. "Sancus 2.0: A Low-Cost Security Architecture for IoT Devices". In: *ACM Transactions on Privacy and Security* 20.3 (2017), pp. 1–33

VulCAN: Efficient Component Authentication and Software Isolation for Automotive Control Networks. This paper presents a generic design for efficient vehicle message authentication, plus software component attestation and isolation using lightweight trusted computing technology.[12] We implement two previously proposed CAN authentication protocols on top of Sancus and evaluate security benefits of enclave software isolation and performance gains of hardware-level cryptography. Our security evaluation exposed a cryptographic flaw in the previously published vatiCAN [195] authentication protocol, which led to a revised specification by the original authors [196]. As part of this research, we furthermore reverse-engineered real-world automotive components and constructed an extended demo setup which was presented at several public events and served as the basis for master thesis student projects. VulCAN was nominated for a distinguished paper award at ACSAC 2017.

This research was conducted in collaboration with and under the supervision of Jan Tobias Mühlberg and Frank Piessens.

> J. Van Bulck, J. T. Mühlberg, and F. Piessens. "VulCAN: Efficient Component Authentication and Software Isolation for Automotive Control Networks". In: *33rd Annual Computer Security Applications Conference (ACSAC)*. Dec. 2017, pp. 225–237

Off-limits: Abusing Legacy x86 Memory Segmentation to Spy on Enclaved Execution. This work presents a novel controlled-channel attack vector for 32-bit SGX enclaves.[13] Particularly, we show that legacy IA32 segmentation features can be abused in combination with SGX-Step (cf. Chapter 4) to deterministically reconstruct memory accesses at an improved, byte-level granularity in the first MiB of the enclave address space, and at a conventional 4 KiB page-level granularity (cf. Chapter 3) for the remainder of the address space. Our analysis revealed that Intel patched this behavior in recent microcode updates.

The principal author of this work is Jago Gyselinck, who conducted this research as part of his master thesis, which was supervised by me, Raoul Strackx, and Frank Piessens.

[12] Open-sourced at `https://distrinet.cs.kuleuven.be/software/vulcan/`.

[13] Open-sourced at `https://distrinet.cs.kuleuven.be/software/off-limits/`.

> J. Gyselinck, J. Van Bulck, F. Piessens, and R. Strackx. "Off-limits: Abusing Legacy x86 Memory Segmentation to Spy on Enclaved Execution". In: *International Symposium on Engineering Secure Software and Systems (ESSoS).* June 2018, pp. 44–60

Foreshadow-NG: Breaking the Virtual Memory Abstraction with Transient Out-of-Order Execution. This technical report was written in the aftermath of the Foreshadow [249] transient-execution processor vulnerability presented in Chapter 6. While our original attack only targeted Intel SGX enclaves, Intel's subsequent investigation [107] into the microarchitectural root cause behind Foreshadow revealed that the same underlying vulnerability can also be abused to break conventional process or even virtual machine isolation. The aim of this technical report was to comprehensively analyze the microarchitectural root cause driving this new class of "Foreshadow-NG" attacks and to derive insights that can form the basis for subsequent scientific analyses. An important such insight was to for the first time differentiate Meltdown-type attacks in terms of the page-table bits used to trigger a page fault exception, which subsequently led to our more systematic transient-execution attack classification [36].

This article was written in collaboration with all authors of the original Foreshadow paper.

> O. Weisse, J. Van Bulck, M. Minkin, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, R. Strackx, T. F. Wenisch, and Y. Yarom. "Foreshadow-NG: Breaking the Virtual Memory Abstraction with Transient Out-of-Order Execution". In: *Technical Report* (Aug. 2018)

Reflections on Post-Meltdown Trusted Computing: A Case for Open Security Processors. In this opinion article and call for action, written in the aftermath of Spectre [146], Meltdown [162], and Foreshadow [249], we argue that the recent wave of microarchitectural vulnerabilities in commodity hardware requires us to question our understanding of system security. We briefly survey ongoing community efforts for developing a new generation of open-source security architectures, like Sancus [189] and CHERI [275], for which we can collectively attempt to build a clear understanding of execution semantics.

This opinion article emerged from several extended discussions with Jan Tobias Mühlberg.

> J. T. Mühlberg and J. Van Bulck. "Reflections on Post-Meltdown Trusted Computing: A Case for Open Security Processors". In: *;login: the USENIX magazine* 43.3 (2018), pp. 6–9

Tutorial: Building Distributed Enclave Applications with Sancus and SGX. In this abstract we outline a half-day interactive tutorial[14] in which we taught how to practically deploy and attest distributed I/O-driven enclave applications where Intel SGX enclaves are securely interfaced with embedded Sancus devices.

This tutorial was jointly organized with Jan Tobias Mühlberg.

> J. T. Mühlberg and J. Van Bulck. "Tutorial: Building Distributed Enclave Applications with Sancus and SGX". in: *48th International Conference on Dependable Systems and Networks (DSN)*. June 2018

Tutorial: Uncovering and Mitigating Side-Channel Leakage in Intel SGX Enclaves. This abstract for a half-day invited tutorial[15] overviews known Intel SGX side-channel leakages in order to arrive at a better understanding of best practices and caveats for writing secure enclave applications.

This tutorial was prepared under the supervision of Frank Piessens.

> J. Van Bulck and F. Piessens. "Tutorial: Uncovering and Mitigating Side-Channel Leakage in Intel SGX Enclaves". In: *8th International Conference on Security, Privacy, and Applied Cryptography Engineering (SPACE)*. Dec. 2018, pp. 20–24

A Systematic Evaluation of Transient Execution Attacks and Defenses. This paper responds to the ongoing wild growth of transient-execution attack variants by presenting an extensible classification tree and uniform naming scheme to reason about this new category of attacks and defenses. As part of the classification effort in this paper, we discovered several new Meltdown variants and mistraining strategies for Spectre.[16] Several researchers [223, 35, 18, 251] have since extended our original classification tree, and we maintain an up-to-date interactive tree at `https://transient.fail`. A subtree covering all Meltdown-type attacks that have been demonstrated to date is furthermore included in Appendix A.

The principal author of this work is Claudio Canella, who collaborated with me, Michael Schwarz, Moritz Lipp, Benjamin von Berg, and Philipp Ortner, under the supervision of Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss.

[14] All materials available at `https://github.com/sancus-tee/tutorial-dsn18`.
[15] All materials available at `https://github.com/jovanbulck/sgx-tutorial-space18`.
[16] Open-sourced at `https://github.com/IAIK/transientfail`.

> C. Canella, J. Van Bulck, M. Schwarz, M. Lipp, B. von Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss. “A Systematic Evaluation of Transient Execution Attacks and Defenses”. In: *28th USENIX Security Symposium.* Aug. 2019, pp. 249–266

Breaking Virtual Memory Protection and the SGX Ecosystem with Foreshadow. Our original research on Foreshadow [249], presented in Chapter 6, was selected to appear in a special IEEE Micro issue on “Top picks from the 2018 computer architecture conferences”. This journal theme article describes the original attack as well as its broader implications for a less technical audience.

This article was written in collaboration with all authors of the original Foreshadow paper.

> J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx. “Breaking Virtual Memory Protection and the SGX Ecosystem with Foreshadow”. In: *IEEE Micro Top Picks from the 2018 Computer Architecture Conferences* 39.3 (2019), pp. 66–74

Fallout: Leaking Data on Meltdown-Resistant CPUs. This work presents a new Meltdown-type transient-execution attack that can leak recently stored kernel data from the processor’s internal store buffer. The paper also includes several side-channel attack variants that abuse TLB and store buffer interactions to reconstruct recent usage of virtual pages. Intel addressed the data leakage aspect of Fallout in recent operating system and microcode updates that overwrite the store buffer on context switches [108].

This research was conducted as a collaboration between TU Graz (Claudio Canella, Lukas Giner, Daniel Gruss, Moritz Lipp, Michael Schwarz), University of Michigan (Daniel Genkin, Marina Minkin), Worcester Polytechnic Institute (Daniel Moghimi, Berk Sunar), KU Leuven (Frank Piessens and I), and University of Adelaide and Data61 (Yuval Yarom).

> C. Canella, D. Genkin, L. Giner, D. Gruss, M. Lipp, M. Minkin, D. Moghimi, F. Piessens, M. Schwarz, B. Sunar, J. Van Bulck, and Y. Yarom. “Fallout: Leaking Data on Meltdown-Resistant CPUs”. In: *26th ACM Conference on Computer and Communications Security (CCS).* Nov. 2019, pp. 769–784

ZombieLoad: Cross-Privilege-Boundary Data Sampling. This work uncovers a new Meltdown-type transient-execution attack that can leak recently loaded data brought into the line-fill buffer by the current or a sibling logical CPU.[17] We develop novel data sampling noise elimination techniques and demonstrate ZombieLoad's effectiveness in a multitude of practical attack scenarios across CPU privilege rings, OS processes, virtual machines, and SGX enclaves. Intel addressed this data leakage in recent operating system and microcode updates that overwrite the line-fill buffer on context switches [108].

The principal author of this work is Michael Schwarz, who collaborated with Moritz Lipp, Daniel Moghimi, me, Julian Stecklina, and Thomas Prescher, under the supervision of Daniel Gruss.

> M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss. "ZombieLoad: Cross-Privilege-Boundary Data Sampling". In: *26th ACM Conference on Computer and Communications Security (CCS).* Nov. 2019, pp. 753–768

Plundervolt: Software-Based Fault Injection Attacks Against Intel SGX. This work for the first time breaks Intel SGX's integrity guarantees by abusing an undocumented voltage scaling software interface in recent Intel Core processors to reliably cause predictable faults in enclave computations.[18] We show how the induced faults can be leveraged to recover full cryptographic keys from constant-time algorithms, as well as to induce memory safety misbehavior in bug-free enclave code. In response to our findings, Intel released a microcode update that disables the undocumented voltage scaling interface at boot time.

Our work on Plundervolt has furthermore been selected to appear in IEEE S&P magazine's special issue on hardware-assisted security. This journal theme article describes the original attack as well as its broader implications for a less technical audience.

The principal author of this work is Kit Murdock, who collaborated with me, in further collaboration with and under the supervision of David Oswald, Flavio Garcia, Daniel Gruss, and Frank Piessens.

> - K. Murdock, D. Oswald, F. D. Garcia, J. Van Bulck, D. Gruss, and F. Piessens. "Plundervolt: Software-Based Fault Injection Attacks Against Intel SGX". in: *41st IEEE Symposium on Security and Privacy (S&P).* May 2020, pp. 1466–1482
> - K. Murdock, D. Oswald, F. D. Garcia, J. Van Bulck, D. Gruss, and F. Piessens. "Plundervolt: How a Little Bit of Undervolting Can Create

[17]Further information and guidance via `https://zombieloadattack.com/`.
[18]Further information and guidance via `https://plundervolt.com/`.

> a Lot of Trouble". In: *IEEE Security & Privacy Magazine Special Issue on Hardware-Assisted Security* (2020)

Provably Secure Isolation for Interruptible Enclaved Execution on Small Microprocessors. This work presents the design and a formal model of an interruptible enclave processor which is subsequently proved to be free from interrupt-based timing leaks, including the Nemesis [256] attack presented in Chapter 5. As part of this work we discover several subtle Nemesis attack variants, including side-channel leakage from interrupt counting and resume-to-end timings. The proposed design was furthermore practically implemented and evaluated on top of a modified version of Sancus [189].[19]

The principal author of this work is Matteo Busi, who collaborated with Job Noorman and me, under the supervision of Letterio Galletta, Pierpaolo Degano, Jan Tobias Mühlberg, and Frank Piessens.

> M. Busi, J. Noorman, J. Van Bulck, L. Galletta, P. Degano, J. T. Mühlberg, and F. Piessens. "Provably Secure Isolation for Interruptible Enclaved Execution on Small Microprocessors". In: *33rd IEEE Computer Security Foundations Symposium (CSF)*. June 2020, pp. 262–276

CopyCat: Controlled Instruction-Level Attacks on Enclaves. This work presents an improved controlled-channel attack technique that combines coarse-grained 4 KiB page-table access patterns (cf. Chapter 3) with fine-grained interrupt counts harvested by SGX-Step (cf. Chapter 4) to break prior assumptions and deterministically track intra-page enclave control flow decisions at a maximal instruction-level granularity.

The principal author of this work is Daniel Moghimi, who collaborated with me, in further collaboration with and under the supervision of Nadia Heninger, Frank Piessens, and Berk Sunar.

> D. Moghimi, J. Van Bulck, N. Heninger, F. Piessens, and B. Sunar. "CopyCat: Controlled Instruction-Level Attacks on Enclaves". In: *29th USENIX Security Symposium*. Aug. 2020, pp. 469–486

1.5 Outline

The remainder of this dissertation is structured as follows. Chapters 2 to 7 first present the main contributions to the field, *i.e.*, the selected peer-reviewed

[19]Open-sourced at `https://github.com/sancus-tee/sancus-core/tree/nemesis`.

publications that comprise this thesis. These chapters are based on the previously published conference-proceeding versions of the corresponding papers, which have been minimally modified to preserve a unified style and layout throughout this thesis. Every chapter is furthermore introduced by a short preamble reflecting on the wider research context and impact of the work. Finally, Chapter 8 concludes by reviewing contributions, lessons learned, and opportunities for future research.

The chapters in this thesis have been ordered to tell a coherent story by traversing into ever-deeper realms of the system stack and the processor pipeline. Chapter 2 first explores the attack surface that stems from improper sanitization at the software level and illustrates practical applications for some of the lower-level techniques introduced later. Next, in Chapter 3, we zoom in on the hardware-software boundary by exploiting the processor's paging and interrupt interfaces to mount new types of stealthy side-channel attacks. Chapter 4 takes this to the next level by perfecting interrupt-driven enclave single-stepping attacks via the practical SGX-Step framework. In Chapter 5, we then markedly traverse the hardware-software boundary via the Nemesis attack, which exposes instruction-granular microarchitectural CPU state through an innovative interrupt latency side channel. Chapter 6 subsequently presents our work on Foreshadow, which for the first time recognizes the dangers of transient execution and delayed exception handling in the processor's microarchitectural pipeline organization. In Chapter 7, LVI attacks bypass existing microcode defenses and ultimately pinpoint the root threat of incorrect transient forwarding in the processor pipeline by cleverly inverting prior transient data sampling attacks, including Foreshadow, into a dangerous new type of microarchitectural data injection. Finally, Chapter 8 concludes and places our contributions in perspective by comprehensively systematizing the SGX attack landscape, which unfolded over the past five years, and formulating insights and recommendations for next-generation, hardened TEE designs.

Chapter 2

Assessing the vulnerability of enclave shielding runtimes

This chapter was previously published as:

> J. Van Bulck, D. Oswald, E. Marin, A. Aldoseri, F. D. Garcia, and F. Piessens. “A Tale of Two Worlds: Assessing the Vulnerability of Enclave Shielding Runtimes”. In: *26th ACM Conference on Computer and Communications Security (CCS)*. Nov. 2019, pp. 1741–1758

Preamble

This chapter analyzes the vulnerability space that arises when interfacing a trusted enclave application with untrusted, potentially malicious code. Considerable research and industry effort has gone into developing TEE runtime libraries with the purpose of transparently *shielding* enclave application code from an adversarial environment. However, our analysis reveals that shielding requirements are generally not well-understood in today’s TEE runtimes. We expose a large and reoccurring vulnerability landscape, categorized into 10 distinct shielding responsibility classes across the Application Binary Interface (ABI) and the Application Programming Interface (API). Our analysis reveals over 35 interface sanitization vulnerabilities across 8 major open-source shielding frameworks for Intel SGX, RISC-V, and Sancus TEEs. We practically exploit these vulnerabilities in several proof-of-concept attack scenarios, and we show that they can lead to memory-safety misbehavior and side-channel leakage in the compiled enclave.

This main idea for this research dates back several years. Some of the initial findings even originate from my master thesis [248], which uncovered several subtle ABI sanitization vulnerabilities in the low-level enclave entry code for Sancus. When working on the stealthy page-table attacks (cf. Chapter 3) in 2016, we found that some of these vulnerabilities also apply to Graphene-SGX, and we reported a related issue that got patched. Having an intuition there might be more of these issues, we deferred a thorough code review to later. In 2018, we reported a severe API sanitization side-channel vulnerability in the official Intel SGX Software Development Kit (SDK), tracked via CVE-2018-3626. Informed by these initial results, a research project was initiated in collaboration with the University of Birmingham to explore interface attack surface across major open-source TEE runtimes from both industry and academia, leading to the more systematic understanding of enclave shielding responsibilities described in this chapter. In the wider research landscape, our work on scrutinizing the enclave interface also helped paving the way for later attacks, like LVI (cf. Chapter 7) and Plundervolt [188], which similarly take advantage of pointer passing in the shared address space.

Following responsible disclosure, the vulnerabilities described in this chapter led to security patches in all of the open-source projects we studied, including the Intel SGX SDK, Microsoft Open Enclave, Google Asylo, and the Rust compiler. In the case of Intel and Microsoft, this can be tracked via 5 designated CVE records. After the embargo on our paper publication had been lifted, when reviewing patches rolled out in response to our earlier disclosure of insufficient status flag sanitization in the Open Enclave SDK, we furthermore noticed that Microsoft engineers additionally sanitized the x87 FPU control word and SSE `mxcsr` control registers. This prompted us to more systematically study the previously overlooked ABI-level attack surface from floating-point registers. This follow-up research led to another vulnerability in the Intel SGX SDK, tracked via CVE-2020-0561, allowing to influence the rounding and precision of enclaved floating-point operations. Our analysis furthermore uncovered a limited remaining attack surface in Open Enclave, tracked via CVE-2020-15107, allowing to silently corrupt x87 computations.

All of the proof-of-concept attacks described in this chapter have been released as open-source, which we hope will provide the community with a relevant sample of vulnerable enclave programs to evaluate future enclave hardening schemes and static analysis tools. We further disseminated our findings at the 2020 Free and Open-source Software Developers' European Meeting (FOSDEM) and the 2nd SGX Community Workshop.

2.1 Introduction

Minimization of the Trusted Computing Base (TCB) has always been one of the key principles underlying the field of computer security. With an ongoing stream of vulnerabilities in mainstream operating system and privileged hypervisor software layers, Trusted Execution Environments (TEEs) [166] have been developed as a promising new security paradigm to establish strong hardware-backed security guarantees. TEEs such as Intel SGX [47], ARM TrustZone [205], RISC-V Keystone [153], or Sancus [189] realize isolation and attestation of secure application compartments, called *enclaves*. Essentially, TEEs enforce a dual-world view, where even compromised or malicious system software in the normal world cannot gain access to the memory space of enclaves running in an isolated secure world on the same processor. This property allows for drastic TCB reduction: only the code running in the secure world needs to be trusted for enclaved computation results. Nevertheless, TEEs merely offer a relatively coarse-grained memory isolation primitive at the hardware level, leaving it up to the enclave developer to maintain useful security properties at the software level. This can become particularly complex when dealing with interactions between the untrusted host OS and the secure enclave, e.g., sending or receiving data to or from the enclave. For this reason, recent research and industry efforts have developed several TEE runtime libraries that transparently shield enclave applications by maintaining a secure interface between the normal and secure worlds. Prominent examples of such runtimes include Intel's SGX SDK [116], Microsoft's Open Enclave SDK [177], Graphene-SGX [246], SGX-LKL [207], Google's Asylo [77], and Fortanix's Rust-EDP [68].

There are some differences in the way each trusted runtime handles input and output to and from the enclave. At the system level, all TEEs offer some form of `ecall`/`ocall` mechanism to switch from the normal to the secure word (and vice versa). Building on this hardware-level isolation primitive, TEE runtimes aim to ease enclave development by offering a higher level of abstraction to the enclave programmer. Particularly, commonly used production-quality SDKs [116, 177] offer a secure function call abstraction, where untrusted code is allowed to only call explicitly annotated `ecall` entry points within the enclave. Furthermore, at this level of abstraction the enclave application code can call back to the untrusted world by means of specially crafted `ocall` functions. It is the TEE runtime's responsibility to safeguard the secure function call abstraction by sanitizing low-level ABI state and marshalling input and output buffers when switching to and from enclave mode. However, the SDK-based approach still leaves it up to the developer to manually partition secure application logic and design the enclave interface. As an alternative to such specifically written enclave code, one line of research [20, 246, 17, 231] has developed dedicated enclave library OSs that seamlessly enforce the `ecall`/`ocall` abstraction at the system call level. Ultimately, this approach holds the promise to securely running unmodified executables inside an enclave and fully transparently applying TEE security guarantees.

Over the last years, security analysis of enclaved execution has received considerable attention from a microarchitectural side-channel [256, 156, 161, 257, 180] and more recently also transient-execution perspective [249, 39, 148]. However, in the era where our community is focusing on patching enclave software against very advanced Spectre-type attacks, comparably little effort has gone into exploring how resilient commonly used trusted runtimes are against plain *architectural* memory-safety style attacks. Previous research [154, 24] has mainly focused on developing techniques to efficiently exploit traditional memory safety vulnerabilities in an enclave setting, but has not addressed the question how prevalent such vulnerabilities are across TEE runtimes. More importantly, it remains largely unexplored whether there are *new* types of vulnerabilities or attack surfaces that are specific to the unique enclave protection model (e.g., ABI-level misbehavior, or API-level pointer poisoning in the shared address space). Clearly, the enclave interface represents an important attack surface that so far has not received the necessary attention and thus is the focus of this chapter.

Our contribution. In this chapter, we study the question of how a TEE trusted runtime can securely "bootstrap" from an initial attacker-controlled machine state to a point where execution can be safely handed over to the actual application written by the enclave developer. We start from the observation that TEE runtimes hold the critical responsibility of shielding an enclave application at all times to preserve its intended program semantics in a hostile environment. As part of our analysis, we conclude that the complex shielding requirement for an enclave runtime can be broken down into at least two distinct tiers of responsibilities.

In a first ABI-level tier, we consider that upon enclave entry, the adversary usually controls a significant portion of the low-level machine state (e.g., CPU registers). This requires sanitization, typically implemented through a carefully crafted enclave entry assembly routine to establish a trustworthy ABI state as expected by the compiled application code. Examples of trusted runtime responsibilities at this level include switching to a private call stack, clearing status register flags that may adversely affect program execution, or scrubbing residual machine state before enclave exit.

Secondly, we consider that the enclaved binary itself makes certain API-level assumptions. Here we pay particular attention to pointers and size arguments, because in many TEE designs [47, 189, 153], at least part of the enclave's address space is shared with untrusted adversary-controlled code. Hence, the enclaved binary may assume that untrusted pointer arguments are properly sanitized to point outside of trusted memory, or that `ocall` return values have been scrutinized. Our main contributions are:

- We categorize enclave interface shielding responsibilities into 10 distinct classes, across the ABI and API tiers (cf. Table 2.1).
- We analyze 8 widely used enclave runtimes, revealing a recurring vulnerability landscape, ranging from subtle side-channel leakage to more grave types of memory safety infringements.
- We practically demonstrate according attacks in various application scenarios by extracting full cryptographic keys, and triggering controlled enclave memory corruptions.
- We show that state-of-the-art automated enclave interface sanitization approaches such as `edger8r`, or even the use of safe languages like Rust, fail to fully prevent our attacks, highlighting the need for more principled mitigation strategies.

Responsible disclosure. All of the security vulnerabilities described in this work have been responsibly disclosed through the proper channels for each affected TEE runtime. In each case, the issues have been verified and acknowledged by the developers. In the case of Intel, this can be tracked via CVE-2018-3626 and CVE-2019-14565, and for Microsoft via CVE-2019-0876, CVE-2019-1369, and CVE-2019-1370. The weakness found in Fortanix-EDP led to a security patch in the Rust compiler. For other open-source projects, our reports have been acknowledged in the respective commits or issues on GitHub. We worked with the maintainers of said projects to ensure mitigation of the problems reported in this chapter.

To ensure the reproducibility of our work, and to provide the community with a relevant sample of vulnerable enclave programs for evaluating future attacks and defenses, we published all of our attack code at `https://github.com/jovanbulck/0xbadc0de`.

2.2 Background and related work

This section reviews enclave operation and TEE design, introduces the trusted runtime libraries we analyzed in this work, and finally summarizes related work on TEE memory corruption attacks.

2.2.1 Enclave entry and exit

TEE design. The mechanisms to interface with enclaves vary depending on the underlying TEE being used. Figure 2.1 shows how, from an architectural point of view, we distinguish two types of TEE designs: those that rely on a single-address-space model (e.g., Intel SGX [47] and Sancus [189]) vs. the ones that follow a two-world view (e.g., ARM TrustZone [205] and Keystone [153]). In the former case, enclaves are embedded in the address space of an unprivileged host application. The processor orchestrates enclave entry/exit events, and enforces that enclave memory can never be accessed from outside the enclave. Since the trusted code inside the enclave is allowed to freely access unprotected memory locations outside the enclave, bulk input/output data transfers are supported by simply passing pointers in the shared address space.

In the case of a two-world design, on the other hand, the CPU is logically divided into a "normal world" and a "secure world". A privileged security monitor software layer acts as a bridge between both worlds. The processor enforces that normal world code cannot access secure world memory and resources, and may only call a predefined entry point in the security monitor. Since the security monitor has unrestricted access to memory of both worlds, an explicit "world-shared memory" region can typically be setup to pass data from the untrusted OS into the enclave (and vica versa).

Enclave entry/exit. Given that the runtimes we studied focus mainly on Intel SGX (cf. Section 2.3.2), we now describe `ecall/ocall` and exception handling following SGX terminology [47]. Note that other TEEs feature similar mechanisms, the key difference for a two-world design being that some of the enclave entry/exit functionality may be implemented in the privileged security monitor software layer instead of in the processor.

In order to enter the enclave, the untrusted runtime executes the `eenter` instruction, which switches the processor into enclave mode and transfers execution to a predefined entry point in the enclave's Trusted Runtime System (TRTS). Any metadata information, including the requested `ecall` interface function to be invoked, can be passed as untrusted parameters in CPU registers.

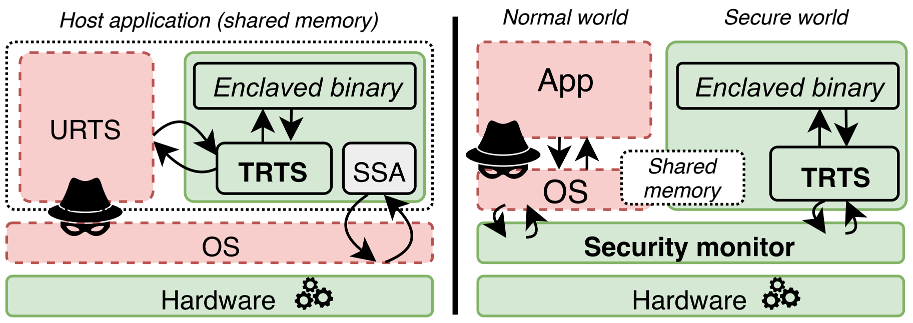


Figure 2.1: Enclave interactions in a single-address-space TEE design (left) vs. two-world design (right). The software components we study are bold, and the TCB is green (solid lines).

TRTS first sanitizes CPU state and untrusted parameters before passing control to the `ecall` function to be executed. Subsequently, TRTS issues an `eexit` instruction to perform a synchronous enclave exit back to the untrusted runtime, again passing any parameters through CPU registers. The process for `ocalls` takes place in reverse order. When the enclave application calls into TRTS to perform an `ocall`, the trusted CPU context is first stored before switching to the untrusted world, and restored on subsequent enclave re-entry.

When encountering interrupts or exceptions during enclaved execution, the processor executes an Asynchronous Enclave Exit (AEX) procedure. AEX first saves CPU state to a secure State Save Area (SSA) memory location inside the enclave, before scrubbing registers and handing control to the untrusted OS. The enclave can subsequently be resumed through the `eresume` instruction. Alternatively, the untrusted runtime may optionally first call a special `ecall` which allows the enclave's TRTS to internally handle the exception by inspecting and/or modifying the saved SSA state.

2.2.2 TEE shielding runtimes

Intel SGX SDK. With the release of the open-source SGX SDK, Intel [116] supports a secure function call abstraction to enable production enclave development in C/C++. Apart from pre-built trusted runtime libraries, a key component of the SDK is the `edger8r` tool, which parses a developer-provided Enclave Description Language (EDL) file in order to automatically generate trusted and untrusted proxy functions to be executed when crossing enclave boundaries.

Microsoft Open Enclave SDK. Microsoft developed the Open Enclave (OE) SDK with the purpose of facilitating TEE-agnostic production enclave development [177]. Currently, OE only supports Intel SGX applications, but in the future TrustZone-based TEEs will also be supported through OP-TEE bindings [177]. The OE runtime includes a custom fork of Intel's `edger8r` tool.

Google Asylo. Google aims to provide a higher-level, platform-agnostic C++ API to develop production enclaves in a Remote Procedure Call (RPC)-like fashion [77]. While the Asylo specification aims to generalize over multiple TEEs, presently only a single SGX back-end is supported, which internally uses Intel's SGX SDK. From a practical perspective, the Asylo runtime can thus be regarded as an additional abstraction layer on top of the Intel SGX SDK.

Fortanix Rust-EDP. As an alternative to Intel's and Microsoft's SDKs written in C/C++, Fortanix released a production-quality SGX toolchain to develop enclaves in the safe Rust language [68]. The combination of SGX's isolation guarantees with Rust's type system aims to rule out memory safety attacks against the trusted enclave code. Similar to libOS-based approaches, Rust-EDP hides the enclave interface completely from the programmer and transparently redirects all outside world interactions in the standard library through a compact and scrutinized `ocall` interface.

Graphene-SGX. This open-source library OS approach allows to run unmodified Linux binaries inside SGX enclaves [246]. The trusted Graphene-SGX runtime transparently takes care of all enclave boundary interactions. For this, the libOS offers a limited `ecall` interface to launch the application, and translates all system calls made by the shielded application binary into untrusted `ocalls`. While Graphene was originally developed as a research project, it is currently meeting increasing industry adaption and thrives to become a standard solution in the Intel SGX landscape [80].

SGX-LKL. This open-source research project offers a trusted in-enclave library OS that allows to run unmodified Linux binaries inside SGX enclaves [207]. Similarly to Graphene-SGX, SGX-LKL intercepts all system calls in the shielded application binary, but the libOS layer is internally based on the Linux Kernel Library (LKL).

Keystone. Keystone [153] is an open-source research framework for developing customized TEEs in RISC-V processors. Keystone adopts a "secure world" view similar to ARM TrustZone [205] where a privileged security monitor software layer separates enclaves in their own address spaces, potentially including explicit shared memory regions. Keystone enclaves feature a trusted runtime which intercepts system calls and transparently tunnels all untrusted world interactions through the underlying security monitor.

Sancus. The Sancus research TEE [189] offers lightweight enclave isolation and attestation on an embedded 16-bit TI MSP430 processor featuring a plain single-address-space without virtual memory. A dedicated C compiler automates enclave creation and includes a small trusted runtime library that is transparently invoked on enclave entry/exit. Trusted software may additionally provide code confidentiality [78] or authentic execution [192] guarantees.

2.2.3 Related work

OS system call interface. During the last decade, significant research efforts have been made to discover and mitigate vulnerabilities in OS kernels, such as missing pointer checks, uninitialized data leakage, or buffer and integer overflows [41]. By exploiting a single vulnerability in a kernel, unprivileged adversaries may read or write arbitrary memory and gain root access. While these vulnerabilities continue to be relevant in modern kernels, they are generally well understood by the OS security community. However, they have received less attention in the context of TEEs.

Checkoway et al. [38] first demonstrated that an *untrusted* OS can perform so called Iago attacks to compromise legacy applications by supplying maliciously crafted pointers or lengths as the return value of a traditionally trusted system call like `malloc()`. These attacks are closely related to a small subset of the vulnerabilities described in this work, specifically attack vector #9, which exploits that pointers or buffer sizes returned by untrusted `ocalls` may not be properly sanitized (cf. Section 2.5.5). Our work generalizes Iago attacks from the OS system call interface to `ocalls` in general, and more broadly shows that Iago attacks are but one instance of adversarial OS interactions. We show, for instance, that legacy applications may also make implicit assumptions on the validity of `argv` and `envp` pointers, which are not the result of system calls.

Memory corruption attacks on ARM TrustZone. ARM TrustZone [205] was one of the first widely deployed TEEs, particularly in mobile devices, and hence received considerable attention from security researchers. The code running in the secure world largely depends on the device manufacturer, with widely used runtimes including Trustonic Kinibi, Qualcomm's QSEE, Google's Trusty, and the open-source project OP-TEE. Over the past years, several vulnerabilities [205, 242] have been discovered in TrustZone runtimes caused by e.g., missing or incorrect pointer range or length checks, or incorrect handling of integer arithmetic. Often, these vulnerabilities rely on the existence of a shared memory region for data exchange between the normal and secure worlds: if an adversary passes a pointer into trusted memory where a pointer to shared memory is expected, memory corruption or disclosure may occur when the pointer is not properly validated by the trusted runtime.

Machiry et al. [165] presented a related class of Boomerang attacks, which leverage the fact that TrustZone's secure world OS has full access to untrusted memory, including the regions used by the untrusted OS. Boomerang exploits that trusted pointer sanitization logic may only validate that pointers lie outside of secure memory, allowing unprivileged code executing in the normal world to read or write memory locations belonging to other applications or the untrusted OS. In a sense, Boomerang vulnerabilities are orthogonal to a subset of the vulnerabilities described in this chapter: both target incorrect pointer checks within trusted code, but while Boomerang attacks relate to checks of pointers into *untrusted* memory, we focus on pointers into *trusted* memory.

Memory corruption attacks on Intel SGX. Lee et al. [154] were the first to execute a completely blind memory corruption attack against SGX by augmenting code reuse attack techniques [228] with several side-channel oracles. To successfully mount this attack, adversaries require kernel privileges and a static enclave memory layout. Recently, these techniques were improved by Biondo et al. [24] to allow even non-privileged adversaries to hijack vulnerable enclaves in the presence of fine-grained address space randomization [227]. Their approach is furthermore made application-agnostic by leveraging gadgets found in the trusted runtime library of the official Intel SGX SDK. In a perpendicular line of research, Schwarz et al. [219] criticized SGX's design choice of providing enclaves with unlimited access to untrusted memory outside the enclave. They demonstrated that malware code executing inside an SGX enclave can mount stealthy code reuse attacks to hijack control flow in the untrusted host application.

Importantly, all previous SGX memory safety research focused on contributing novel exploitation techniques while assuming the prior presence of a vulnerability in the enclave code itself. Hence, those results are *complementary* to the vulnerabilities described in this work. We have indeed demonstrated control flow hijacking for some of the pointer sanitization issues below, and these may further benefit from exploitation techniques developed in prior work.

2.3 Methodology and adversary model

2.3.1 Attacker model

We consider systems with hardware support for a TEE and where a trusted runtime supports the secure, shielded execution of an enclaved binary produced by the application developer. With *enclaved binary*, we specifically mean that the binary is the output of a standard compiler, which is not aware of the TEE. It is the responsibility of the shielding runtime to preserve intended program semantics in a hostile environment. We focus exclusively on vulnerabilities in the TEE runtime and assume that there are no application-level memory safety vulnerabilities in the enclaved binary.

We assume the standard TEE attacker model [166], where adversaries have full control over *all* software executing *outside* the hardware-protected memory region. This is a powerful attacker model, allowing the adversary to, for instance, modify page-table entries [277, 258], or precisely execute the victim enclave one instruction at a time [257]; yet, this is the attacker that TEEs are designed to defend against. It is important to note that some of the attacks we discuss can also be launched by significantly less privileged attackers, *i.e.*, with just user-level privileges to invoke the enclave.

2.3.2 Research methodology

Our objective is to pinpoint enclave shielding responsibilities, and to find vulnerabilities where real-world TEE runtimes fail to safeguard implicit interface assumptions made by the enclaved binary.

TEE runtime code review. We base our research on manual code review, and hence limited our study to open-source TEE runtimes. After reviewing the literature and code repositories, we selected 8 popular runtimes to be audited. Our resulting selection allows to compare tendencies in *(i)* production vs. research code bases; *(ii)* SDK vs. libOS-based shielding abstractions; *(iii)* unsafe C/C++ vs. safe Rust programming languages; and *(iv)* underlying TEE design dependencies. Note that we opted not to include `baidu-rust-sgx`, as it is merely a layer on top of Intel SGX SDK (and hence inherits all vulnerabilities of the latter). After reviewing prior research [242] and relevant code, we found that sanitization in the TrustZone runtime OP-TEE has already been thoroughly vetted and we hence decided not to systematically audit this runtime. For each of the selected TEE runtime implementations, we then reviewed the

Table 2.1: Enclave runtime vulnerability assessment (our contribution, highlighted) and comparison to related work on user-to-kernel exploits. Symbols indicate whether a vulnerability was successfully exploited (★); acknowledged but without proof-of-concept (●); or not found to apply (○). Half-filled symbols (⯪, ◐) indicate that improper sanitization only leads to side-channel leakage.

	Vulnerability \ Runtime	SGX-SDK	OpenEnclave	Graphene	SGX-LKL	Rust-EDP	Asylo	Keystone	Sancus	Linux
ABI	#1 Entry status flags sanitization	★	★	◐	●	◐	●	○	○	[55]
	#2 Entry stack pointer restore	○	○	★	●	○	○	○	★	○
	#3 Exit register leakage	○	○	○	★	○	○	○	○	○
Tier2 (API)	#4 Missing pointer range check	○	★	★	★	○	●	○	★	[41]
	#5 Null-terminated string handling	⯪	★	○	○	○	○	○	○	[41]
	#6 Integer overflow in range check	○	○	●	○	●	○	●	●	[41]
	#7 Incorrect pointer range check	○	○	●	○	○	●	○	●	○
	#8 Double fetch untrusted pointer	○	○	●	○	○	○	○	○	[276, 218]
	#9 Ocall return value not checked	○	★	★	★	○	●	★	○	–
	#10 Uninitialized padding leakage	[155]	★	○	●	○	●	★	★	[45]

sanitizations and defensive checks implemented by the trusted runtime between entering the TEE and transferring control to the enclaved binary, and the symmetrical path when exiting the TEE. We found new vulnerabilities in all studied runtimes. Table 2.1 summarizes our findings, structured according to the respective vulnerability classes, and relating to similar vulnerabilities in the Linux kernel and prior TEE research. Our systematization revealed 10 distinct attack vectors across 2 subsequent tiers of TEE shielding responsibilities, explored in Sections 2.4 and 2.5, respectively.

In our code review, we focus our attention on the assumptions that an enclaved binary makes about two key interfaces, and we consider both integrity and confidentiality concerns. A first level of interface sanitization we inspect is the ABI, which unambiguously specifies function calling conventions regarding the low-level machine state expected by the compiler [65]. We manually locate the trusted runtime entry point, and review how the compact assembly routine establishes a trustworthy ABI state on entry, and similarly scrubs residual CPU state on exit. The second key interface, that we refer to as the API, is the functional interface of the enclaved binary. We review how the TEE runtime validates different kinds of arguments passed in through an `ecall` or as the return value of an `ocall`. We focus in particular on the handling of pointers and strings, where it is the TEE runtime's responsibility to ensure that variable-sized buffers lie entirely outside the enclave before copying them inside and transferring execution to the enclaved binary. For confidentiality, we check again that all memory copied outside the TEE only contains explicit return values, and that no avoidable side-channel leakage is introduced.

TEE design considerations. The communication between enclave and untrusted code for all TEE runtimes considered in this chapter relies on some form of "world-shared memory", *i.e.*, a memory region that is accessible to both trusted and untrusted code. Depending on the specific TEE design (cf. Fig. 2.1), this can be realized by either embedding the enclave in the address space of a surrounding host process, as in Intel SGX [47] or Sancus [189], or by explicitly mapping a dedicated virtual memory region into both worlds as in ARM TrustZone [205] and Keystone [153]. Prior research has mainly explored interface sanitization vulnerabilities in ARM TrustZone TEEs (cf. Section 2.2.3). Given the prevalence of SGX in contemporary Intel processors, our study focuses largely on SGX-style single-address-space TEE designs as used in 7 out of 8 considered runtimes. However, the example of Keystone, and prior research on ARM TrustZone [242, 205], shows that the attack surface studied here is not necessarily limited to TEEs using the single-address-space approach taken by SGX. As part of our analysis, we found that certain TEE-specific design considerations may sometimes significantly impact exploitability. When applicable, such TEE design considerations are discussed throughout the chapter.

2.4 Establishing a trusted ABI

Similarly to traditional user/kernel isolation, TEE-enabled processors typically only take care of switching to a fixed entry point and thereafter leave it up to trusted runtime software to securely bootstrap the enclaved execution. In practice, this implies that adversaries may still control a large fraction of the low-level machine state (e.g., CPU registers) on enclave entry. Hence, a trusted assembly entry routine is responsible to establish an ABI-compliant machine state when transferring control to the shielded application, and to save and scrub low-level machine state on enclave exit.

2.4.1 Sanitizing machine state on entry

After reviewing well-documented ABI-level calling conventions [65] expected by popular C compilers, we concluded that most CPU registers can be left unmodified, apart from the stack pointer explored in the next section. However, a more subtle concern relates to the expected state of certain status register flags on function entry.

Attack vector #1 (status flags): Entry code should sanitize register flags that may adversely impact program execution. ▷ Prevalent in production and research runtimes, but exclusively Intel SGX (x86 CISC).

TEE design. The underlying processor architecture used in the specific TEE design may greatly impact the resulting ABI-level attack surface. That is, in comparison to Intel's notoriously complex x86 CISC architecture [47], simpler RISC-based TEEs such as Sancus [189], Keystone [153], or ARM TrustZone [205] tend to impose less obligations for trusted software to sanitize low-level machine state. For instance, we found that the Sancus runtime should only take care to clear the interrupt flag. Likewise, TrustZone even transparently takes care to save/restore secure world stack pointer registers. Our analysis further reveals the trade-offs for implementing register and status flag clearing in either hardware or software. For instance, we show that the Intel SGX design leaves this responsibility largely to software, exposing a larger attack surface.

We methodically examined all the software-visible flags in the x86 `rflags` register [114] and discovered two potentially dangerous flags that may adversely impact enclaved execution if not properly cleared. First, the Alignment Check (AC) flag may be set before entering the enclave in order to be deterministically notified of every unaligned memory access performed by the trusted enclave software. This novel side-channel attack vector is closely related to well known page fault [277] or segmentation fault [91] controlled channels, but this time abuses x86 `#AC` alignment-check exceptions. Also, note that `#PF` side channels ultimately reflect fundamental hardware-level TEE design decisions that cannot be avoided in software, whereas we argue that `#AC` leakage originates from the trusted runtime's failure to clear the associated status register control flag. A second and more dangerous ABI-level attack vector arises from the Direction Flag (DF), which can be set to change the loop behavior of x86 string instructions (e.g., `rep movs`) from auto-increment to auto-decrement. Commonly used x86 ABIs [65] allow for compiler optimizations by mandating that DF shall always be cleared on function call/return. However, in case this subtle ABI requirement is not explicitly enforced in the assembly entry routine, SGX adversaries may change DF to an unexpected "decrement" direction before the `ecall` and thereby hijack the intended direction of all subsequent x86 string instructions executed by the enclave. This opens a severe vulnerability that can be successfully exploited to trigger enclave memory corruption and erroneous computation results.

Intel SGX SDK. We experimentally confirmed that the trusted runtime in Intel's official SGX SDK [116] does *not* clear AC or DF on enclave entry. The

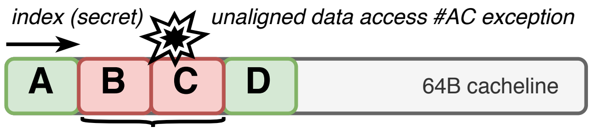


Figure 2.2: Misaligned, intra-cache line secret data access.

latter can be tracked via CVE-2019-14565 (Intel SA-00293), leading to enclave TCB recovery.

While unaligned data accesses (e.g., fetching a 16-bit word at an odd byte address) are explicitly supported in the x86 architecture, the processor may optionally be forced to generate an exception for such accesses when software sets the AC bit in the `rflags` register. We developed a minimal sample enclave to showcase how `#AC` exceptions may in certain scenarios reveal secret-dependent data accesses at an enhanced byte-level granularity as compared to state-of-the-art SGX side-channel attacks that are restricted to a coarser-grained 64 B cache line [225] or 4 KiB page-level [277, 258] granularity. Figure 2.2 illustrates the key idea behind the attack, where a 16-bit word is loaded by specifying a byte-granular index in a small lookup table that has been explicitly aligned to a cache line boundary (e.g., as might also be performed in a streamed data or string processing enclave application). In the example, secret index 0 returns the data `AB`, whereas secret index 1 returns `BC`. Our exploit deterministically reconstructs the intra-cache line secret-dependent data access by observing whether or not the enclaved execution generates an `#AC` alignment-check exception. One of the challenges we encountered is to make the enclave progress after returning from the untrusted signal handler. Since the processor automatically restores the previous value of the `rflags` register (including the set AC bit) from enclave-private SSA memory when resuming the enclave [47], the unaligned data access will never be allowed to complete. To overcome this challenge, we make use of the adversary's root privileges to load a simple kernel module that clears the processor's Alignment Mask (`CR0.AM`) to temporarily disable alignment checking. Combined with a single-stepping attack primitive like SGX-Step [257], this approach allows to determine noise-free alignment side-channel information for *every* single instruction in the victim enclave.

It should be noted that the oversight of not clearing the AC flag in the trusted runtime merely leaks address-related side-channel information, which falls explicitly outside of SGX's threat model [47]. However, this is distinctly *not* the case for the DF flag, which directly intervenes with the semantics of the enclaved execution. We confirmed, for instance, that the popular `gcc v5.4` compiler replaces common `strlen()` and `memset()` invocations with inlined x86 string instructions at optimization level `-Os`. We developed a start-to-end attack scenario to show how forcibly inverting the direction of such string operations when entering the enclave through an `ecall` can lead to controlled heap corruption and memory disclosure. Our PoC exploit targets `edger8r` bridge code that is automatically generated to copy input and output buffers to and from the enclave (cf. Section 2.5.1 and Fig. 2.3). Particularly, we abuse that `edger8r` code allocates the output buffers on the enclave heap and thereafter uses `memset()` to securely initialize the newly allocated buffer to all-zero. However, setting DF before the `ecall` causes the `memset()` direction to be inverted and any preceding heap memory to be corrupted (*i.e.*, zeroed). Due to the way the SGX SDK enclave heap is organized, this will ultimately lead to a crash on the next `free()` invocation in the `edger8r` code. Every heap frame is preceded by a size field and a pointer to a metadata bookkeeping structure. Such pointers are stored in `xor`-ed form with a randomly generated secret constant to harden the code against traditional heap corruption attacks. We confirmed that after erroneously zeroing the preceding heap frames, the resulting pointer will most likely end up as a non-canonical 64-bit address and halt the enclave by means of a general protection fault. However, before finally calling `free()` and detecting the heap corruption, the trusted `edger8r`-generated code still copies the allocated output buffer outside the enclave, potentially leading to secret disclosure (as this buffer has never been properly zeroed). We note that the heap corruption in itself may also be leveraged in application-specific scenarios, e.g., zeroing out a cryptographic key residing in the preceding heap frame.

Microsoft Open Enclave SDK. We experimentally confirmed that OE suffers from the same DF vulnerability described above (tracked via CVE-2019-1370). However, we found that after entering the enclave with the DF flag set, the trusted runtime already crashes early-on in the entry path. The reason for this is that on our machines (`gcc v5.4` using the default Makefile), one of the compiled entry functions uses a `rep` string instruction to initialize a local variable on the call stack. Hence, setting DF leads to memory corruption by overwriting a piece of the *trusted* call stack with zeroes. We have not attempted to further exploit this behavior.

Other SGX runtimes. When reviewing the assembly entry routines of the other SGX-based shielding systems (cf. Table 2.1), we found that *none* of them sanitizes AC, whereas interestingly both Rust-EDP and Graphene-SGX clear DF on enclave entry. Note that Google's Asylo framework is built on top of the Intel SGX SDK and hence inherits all of the vulnerabilities described above.

2.4.2 Maintaining the call stack abstraction

In order to safeguard enclave confidentiality and integrity, it is essential that enclaves feature their own private call stack. When exiting the TEE by means of an `ocall`, the trusted stack pointer should be stored and control flow should continue at a location outside the enclave. After having performed an `ocall`, upon receiving the next `ecall`, the private call stack should be restored so the runtime can "return" into the shielded application.

Attack vector #2 (call stack): Entry code should safeguard the call stack abstraction for `ecalls` and `ocalls`. ▷ Not applicable to TrustZone, well-understood in production SGX SDKs, but not always in research code.

TEE design. We observed that TEE-specific design decisions may largely impact the attack surface arising from call stack switching. That is, in ARM TrustZone [205] the stack pointer CPU register is duplicated and fully transparently stored/restored on secure world context switches. More versatile TEE designs like Intel SGX [47] or Sancus [189], on the other hand, support multiple mutually distrusting enclaves and leave it up to trusted runtime software to store and restore the stack pointer across enclave boundaries. Another illustration of the trade-offs between hardware and software responsibilities arises in SGX's `eexit` instruction, which was designed to explicitly fault when supplying in-enclave continuation addresses [47]. Alternative TEE designs like Sancus [189], on the other hand, expect such continuation pointer checks to be performed by the trusted software, leaving a larger attack surface.

Graphene-SGX. After scrutinizing Graphene's low-level bootstrapping code, we discovered that `enclave_entry.S` does not properly safeguard the `ocall` return abstraction. Listing 2.1 shows how the code unconditionally jumps to the stack pointer restore logic after merely receiving an *unchecked* magic value in the `%rdi` register. We experimentally confirmed that this can be abused to illegally "return" into an enclave thread that is not waiting for a previous `ocall` return. An adversary can exploit this weakness to erroneously initialize the trusted in-enclave stack pointer of a newly started thread with the value of the last `ocall`. The memory content at these locations determine the values popped into registers, and ultimately `ret` control flow.

SGX-LKL. We found a highly similar vulnerability in the way SGX-LKL's low-level entry code distinguishes different `ecall` types. Specifically, we noticed that the unchecked parameter in `%rdi` can be poisoned to trick the entry routine

```
  cmp $RETURN_FROM_OCALL, %rdi     ; %RDI = attacker arg
  je .Lreturn_from_ocall
  ...
.Lreturn_from_ocall
★ mov %gs:SGX_LAST_STACK, %rsp
  ...
  ret
```

Listing 2.1: Low-level `ocall` return path in Graphene-SGX.

into erroneously calling a signal handler for a thread that was never interrupted. This is especially problematic as the signal handler code will then illegally restore the stack pointer register from an uninitialized memory location.

Sancus. We reviewed the assembly code inserted at the entry point of a Sancus enclave, and noticed that the Sancus TEE suffers from similar call stack switching vulnerabilities. Particularly, we experimentally confirmed that it is possible to supply illegal CPU register arguments and trick the enclave into "returning" into a thread that was not waiting for a previous `ocall` return. In such a case, the enclave stack will be falsely restored to the value of the last valid `ocall`, leading to memory-safety violations from incorrect control flow and register values. Sancus's enclave entry assembly routine further expects a CPU register parameter to specify the address where execution is continued after leaving the enclave. The software does not properly validate this parameter. Unlike SGX's `eexit` hardware primitive, which refuses to jump to illegal continuation addresses, Sancus enclaves are exited by means of an ordinary `jmp` instruction. We experimentally confirmed the possibility of code reuse attacks [228] by forcing the vulnerable entry routine to jump to an arbitrary in-enclave continuation address.

2.4.3 Storing and scrubbing machine state on exit

Prior to exiting the TEE, the trusted runtime's assembly routine should save and clear all CPU registers that are not part of the calling convention, and restore them on subsequent enclave re-entry. This is highly similar to how a traditional operating system needs to context switch between processes, and hence we found this to be a generally well-understood requirement.

Attack vector #3 (register state): Exit code should save and scrub CPU registers. ▷ Generally well-understood across runtimes and architectures.

TEE design. Similar to parameter passing across traditional user/kernel boundaries, widespread TEE designs commonly preserve CPU register contents when context switching between the normal and secure worlds. Prior research [154, 24] on exploiting memory safety vulnerabilities in SGX enclaves has, for instance, exploited that the `eexit` instruction does *not* clear register values, leaving this as an explicit software responsibility. Further, while scrubbing CPU registers on enclave interrupt is a hardware responsibility in the Intel SGX design [47], we found that the AEX operation in current SGX processors does *not* clear the x86 DF flag (cf. Section 2.4.1). We experimentally confirmed that this can be exploited as a side channel to learn the direction of private in-enclave string operations.

SGX-LKL. When reviewing the respective assembly routines, we noticed that SGX-LKL is the only SGX runtime which does not properly scrub registers before invoking `eexit`. The reason for this oversight is that LKL attempts to leverage the `setjmp/longjmp` standard C library functions to easily store and restore the execution state on enclave entry/exit without needing dedicated assembly code. While indeed functionally correct, *i.e.*, the integrity of CPU registers is preserved across enclave calls, the approach cannot guarantee confidentiality. This is because `setjmp()` still behaves as a normal C function, which—adhering to calling conventions—does *not* clear all CPU state. We therefore advise to use a dedicated assembly routine which overwrites confidential CPU registers before invoking `eexit`. This issue highlights the necessity to explicate and properly separate ABI and API-level shielding concerns in consecutive stages of the trusted runtime (cf. Section 2.3). We experimentally confirmed this vulnerability by loading an elementary AES-NI application binary inside SGX-LKL, and modifying the *untrusted* runtime to dump x86 `xmm` registers—including the AES state and round keys—after enclave exit.

2.5 Sanitizing the enclave API

Once a trustworthy ABI state has been established, the trusted bootstrapping assembly code can safely transfer control to machine code emitted by a compiler from a program description written in a higher-level language. Remarkably, almost all runtimes [116, 177, 246, 207, 77, 153, 189] we studied are written in C or C++, with the notable exception of Fortanix's EDP platform [68], which is written in the memory-safe Rust language. While the use of safe languages is indeed preferable to rule out an important class of application-level memory-safety vulnerabilities in the trusted runtime implementation, we show that safe languages by themselves cannot guarantee that the enclave interface is safe.

That is, it remains the responsibility of the trusted runtime implementation to marshal and scrutinize untrusted input parameters before passing them on to the shielded application written by the enclave developer. Depending on the specific runtime, developers may communicate trusted API sanitization and marshalling requirements explicitly (e.g., using a domain-specific language like in Intel's `edger8r` or Microsoft's `oeedger8r`), or the enclave interface may be completely hidden from the programmer (e.g., libOS-based approaches).

In this section, we analyze shielding requirements for API sanitization based on the different types of arguments that can be passed across the enclave boundary. We pay particular attention to pointers and (variable-sized) input buffers, given the prevalent weaknesses found in real-world code.

2.5.1 Validating pointer arguments

Whenever an enclave shares at least part of its address space with untrusted code, an important new attack surface arises: malicious (untrusted) code can pass in a pointer to enclave memory where a pointer to untrusted memory is expected. Therefore, it is the responsibility of the shielding system to be careful in never dereferencing untrusted input pointers that fall outside of the shared memory region and point into the enclave. In case such sanity checks are missing, the trusted enclave software may unintentionally disclose and/or corrupt enclave memory locations. This is an instance of the well-known "confused deputy" [93] security problem: the attacker is architecturally prohibited from accessing secure enclave memory, but tricks a more privileged enclaved program to inadvertently dereference a secure memory location chosen by the attacker.

Attack vector #4 (pointers): Runtimes should sanitize input pointers to lie inside the expected shared memory region. ▷ Generally understood, but critical oversights prevalent across research and production code.

TEE design. TEEs commonly support some form of shared memory which allows trusted in-enclave code to directly read or write an untrusted memory region outside the enclave (cf. Section 2.3.2). Input and output data transfers can now easily be achieved by bulk-copying into the shared memory region and passing pointers.

Pointer sanitization is a relatively well-known requirement for enclave applications, and even bears some similarity with traditional user-to-kernel

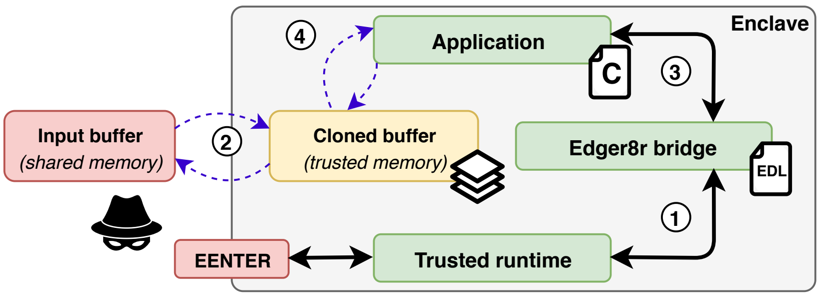


Figure 2.3: Automatically generated `edger8r` bridge code handles shielding of application input and output buffers.

system call validation concerns [41]. However, the kernel system call interface remains largely invisible, fairly stable, and is only modified by a select group of expert developers. SDK-based enclave development frameworks on the other hand expose `ecalls` and `ocalls` much more directly to the application developer by means of a secure function call abstraction.

Intel SGX SDK. In line with trusted runtime shielding requirements, pointer sanitization should preferably not be left to the application developer's end responsibility. As part of the official SGX SDK, Intel [116] therefore developed a convenient tool called `edger8r`, which transparently generates trusted proxy bridge code to take care of validating pointer arguments and copying input and output buffers to/from the enclave. The tool automatically generates C code based on `ecall`/`ocall` function prototypes and explicit programmer annotations that specify pointer directions and sizes in a custom, domain-specific Enclave Definition Language (EDL).

Figure 2.3 gives an overview of the high-level operation of the trusted `edger8r` bridge code. After entering the enclave, the trusted runtime establishes a trusted ABI (cf. Section 2.4), locates the `ecall` function to be called, and finally ① hands over control to the corresponding `edger8r`-generated bridge code. At this point, all input buffer pointers are validated to fall completely outside the enclave, before being copied ② from untrusted shared memory to a sufficiently-sized shadow buffer allocated on the enclave heap. Finally, the `edger8r` bridge transfers control ③ to the code written by the application developer, which can now safely operate ④ on the cloned buffer in enclave memory. A symmetrical path is followed when returning or performing `ocalls` to the untrusted code outside the enclave.

```
1 OE_ECALL void ecall_hello(hello_args_t* p_host_args) {
2   oe_result_t __result = OE_FAILURE;
3   if (!p_host_args || !oe_is_outside_enclave(p_host_args,
4                                               sizeof(*p_host_args)))
5     goto done;
6   ...
7   done:
8 ★   if (p_host_args) p_host_args->_result = __result;
9 }
```

Listing 2.2: Proxy function generated by `oeedger8r` (simplified) with illegal write to arbitrary in-enclave pointer on failure.

Microsoft Open Enclave SDK. Microsoft [177] adopted the sanitization strategy from the Intel SGX SDK by means of their own `oeedger8r` fork. Interestingly, OE uses a "deep copy" marshalling scheme to generalize to TEEs where the enclave cannot directly access host memory and every interaction needs to be mediated in a security kernel with access to an explicit shared memory region (cf. Fig. 2.1). With deep copy marshalling, instead of passing the enclave pointers to the input buffer, the contents of the buffer are first copied into the marshalling structure and then cloned into enclave memory. The pointers in the argument structure are then modified such that they point to the corresponding (cloned) memory buffer.

Nevertheless, we discovered several flaws in the way OE handles pointer validation (tracked via CVE-2019-0876). A first subtle issue was found by reviewing the `oeedger8r`-generated code skeleton itself. Listing 2.2 shows a simplified snippet of the trusted bridge code generated for an elementary `hello()` entry point. The code attempts to properly verify that the untrusted `p_host_args` structure lies outside the enclave, and indeed rejects the `ecall` when detecting a pointer poisoning attempt. However, in the `done` branch at line 8, an error code is still written into the `p_host_args` structure, even if it was found earlier to illegally point inside the enclave. At the time of our review, this could only be exploited when calling the enclave through a legacy `ecall` dispatcher that had unfortunately not been removed from OE's trusted code base (cf. Appendix B.1).

Secondly, we found that enclaves built with OE feature a small number of "built-in" `ecall` entry points for infrastructural functionality directly serviced in the trusted runtime without forwarding to the shielded application. Notably, OE developers decided *not* to route these entry points through `oeedger8r`-generated bridges, but instead opted to manually scrutinize arguments for these special `ecalls`. We audited all eight built-in entry points, and confirmed that most of them were carefully written to prevent pointer sanitization issues, as well as more subtle attack vectors like TOCTOU and speculative execution side channels. However, we found a critical issue in the built-in `_handle_get_sgx_report()` `ecall` involved in crucial attestation functionality (see Appendix B.2 for full code). This function copies the untrusted report input buffer into enclave memory, but never validates whether the argument pointer passed by the untrusted runtime actually lies outside the enclave. This evidently leads to corruption of trusted memory, e.g., when writing the return value in the fall-through branch similar to the `oeedger8r`-generated code discussed above.

Both of the above vulnerabilities allow to write a fixed failure code (`0x03000000` and `0x01000000`) to an arbitrary in-enclave memory location. We developed a PoC based on an existing file-encryptor OE example application, and successfully exploited the above vulnerabilities to forcefully overwrite the first round keys of the AES cipher. This could be extended by overwriting all but the final round keys with known values to perform full key extraction.

Google Asylo. Because Google's Asylo [77] framework is built on top the existing Intel SGX SDK, it also inherits Intel's `edger8r`-based input sanitization scheme. Particularly, the Asylo trusted runtime features a small number of predefined `ecall` entry points, specified in EDL, that implement the necessary functionality to present a higher-level, RPC-like message passing abstraction to the application programmer. Considering that Asylo's runtime extends the trusted computing base on top of Intel's existing SGX SDK, we were interested to assess whether the extra abstraction level may also bring *additional* attack surface. This may, for instance, be the case when making use of the unsafe `[user_check]` EDL attribute [116] that explicitly weakens `edger8r` guarantees and puts the burden of pointer validation on the programmer (e.g., to allow for application-specific optimizations in performance-critical scenarios). Manually scrutinizing the EDL specifications of Asylo's trusted runtime, we found 14 instances of the problematic `[user_check]` attribute. We reviewed these instances and alarmingly found that several of them lacked proper pointer validation, leaving critical vulnerabilities in the compiled enclave (e.g., a write-zero primitive). Notably, the developers took care to validate *second-level* input buffers in the untrusted argument structure, but failed to validate the argument pointer itself (cf. Appendix B.3 for a relevant sample).

Graphene-SGX. While Graphene-SGX's [246] untrusted world interaction and pointer validation concerns are largely limited to `ocalls` (cf. Sections 2.5.3 and 2.5.5), our inspection of the narrow `ecall` interface revealed a rather subtle type of implicit pointer passing that was overlooked. Namely, Graphene's trusted runtime never validates the `argv` and `envp` pointers, which are passed from the untrusted runtime all the way into the main function of the shielded application binary. As a result, adversaries can, for instance, leak arbitrary in-enclave memory when the trusted application outputs `argv` values (e.g., in case of an unknown command line argument). We experimentally confirmed this attack by means of an elementary `echo` program, which unknowingly prints in-enclave secrets after overriding `argv[1]` in the *untrusted* runtime. With respect to mitigations, note that properly sanitizing string arguments can be non-trivial in itself, as explored in Section 2.5.2.

We also found that the special `enclave_ecall_thread_start()` trusted runtime function unconditionally redirects control flow, without performing any validation on the provided untrusted function pointer. We successfully exploited this to jump to arbitrary in-enclave locations, hence allowing code reuse attacks [228].

SGX-LKL. Our analysis of the open-source SGX-LKL `ecall` interface revealed the exact same vulnerability. That is, the trusted `__sgx_init_enclave()` libOS function passes the untrusted `argv` pointer directly to the shielded application without any prior sanitization. We experimentally confirmed that this vulnerability can be abused for information leakage, similar to the above exploit.

Further, the in-enclave signal handler `ecall` entry point does not check that the `siginfo struct` pointer provided by the untrusted runtime lies outside the enclave. This vulnerability can be abused in certain scenarios to leak in-enclave memory contents. For instance, we describe a full exploit for the `SIGILL` signal in Appendix B.4.

Sancus. To demonstrate that untrusted pointer dereference vulnerabilities are *not* limited to advanced virtual memory-based architectures, we also reviewed the trusted runtime and infrastructural enclaves of the low-end open-source Sancus [189] TEE for embedded TI MSP430 devices. As with the above runtimes, we focused our security audit on the enclave boundary code only.

A first critical vulnerability was found in a recent extension [192] to the Sancus compiler infrastructure, which implements a high-level authenticated message passing abstraction to develop distributed event-driven enclave programs. Much like Intel's `edger8r`, the Sancus compiler fully automatically generates `ecall` bridge code to transparently marshal, decrypt, and authenticate input buffers, which can be subsequently processed by the shielded application. We found that the compiler-generated bridge code does *not* sanitize untrusted pointer arguments (cf. Appendix B.5). This may be exploited to forcefully decrypt enclave secrets.

A second input pointer validation vulnerability was found in an infrastructural trusted loader enclave [78] that decrypts third-party application enclaves to preserve code confidentiality. We noticed that the trusted loader enclave code lacks any input pointer validation checks, allowing us to build an arbitrary write primitive in enclave memory. We successfully exploited this vulnerability in a PoC that launches a ROP-style [228] control flow hijacking attack by corrupting the loader enclave call stack.

2.5.2 Validating string arguments

In case the enclave interface is written in a low-level language like C, string arguments do not carry an explicit length and may not even have been properly null-terminated. Thus, shielding runtimes need to first determine the expected length and always include a null terminator when copying the string inside the enclave.

Attack vector #5 (strings): Runtimes should avoid computing untrusted string sizes, and always include a null byte at the expected end. ▷ At least one related instance repeated across two production SDKs.

TEE design. We show below how computing on unchecked string pointers may leak enclave secrets through side channels, even if the `ecall` is eventually rejected. While side channels are generally a known issue across TEE technologies [47, 205, 256, 153] and may even be observed by non-privileged adversaries, for example by measuring overall execution time [180] or attacker-induced cache evictions [225, 161], we show that TEE-specific design decisions can still largely affect the overall exploitability of subtle side-channel vulnerabilities. Particularly, we develop a highly practical attack that abuses several privileged adversary capabilities that have previously proven to be notorious in the Intel SGX design, e.g., untrusted page tables [277, 258], interrupts [156, 257, 256], and storing interrupted CPU register contents in SSA memory frames [249, 39].

Intel SGX SDK. We discovered that `edger8r`-generated code may be tricked into operating on unchecked in-enclave pointers when computing the size of a variable-length input buffer. While such illegal `ecall` attempts will always be properly rejected, we found that adversaries can exploit the unintended size computation as a deterministic oracle that reveals side-channel information

```
1 static sgx_status_t SGX_CDECL sgx_my_ecall(void* pms)
2 {
3   CHECK_REF_POINTER(pms, sizeof(ms_my_ecall_t));
4   ms_my_ecall_t* ms = SGX_CAST(ms_my_ecall_t*, pms);
5   char* _tmp_s = ms->ms_s;
6
7 ★ size_t _len_s = _tmp_s ? strlen(_tmp_s) + 1 : 0;
8   char* _in_s = NULL;
9
10  CHECK_UNIQUE_POINTER(_tmp_s, _len_s);
11  __builtin_ia32_lfence(); // fence after pointer checks
12  ...
```

Listing 2.3: Proxy function generated by `edger8r` for the EDL specification: `public void my_ecall([in,string] char *s)`.

about arbitrary in-enclave memory locations. This vulnerability is tracked via CVE-2018-3626 (Intel SA-00117), leading to enclave TCB recovery and changes in the EDL specification [118]. Prior to our disclosure, EDL allowed programmers to specify a custom `[sizefunc]` attribute that takes as an argument an *unchecked* pointer to an application-specific structure, and returns its size. Likewise, there is a dedicated `[string]` EDL attribute to specify null-terminated string arguments. Essentially, this special case comes down to `[sizefunc=strlen]`.

Consider the code skeleton generated by `edger8r` in Listing 2.3 for an `ecall` that expects a single string pointer argument. In order to verify that the complete string is outside the enclave, the trusted edge routine *first* computes the size of the argument buffer (through either `strlen()` or a dedicated `sizefunc` in general), and only *thereafter* checks whether the entire buffer falls outside of the enclave. It is intended that the edge code first determines the length in untrusted memory, but we made the crucial observation that the `strlen()` invocation at line 7 operates on an arbitrary unchecked pointer, potentially pointing into enclave memory. Any pointer poisoning attempts will subsequently be rejected at line 10, but the unintended computation may have already leaked information through various side channels [156, 257]. In general, leakage occurs whenever there is secret-dependent control or data flow in the specified `sizefunc`. This is most obviously the case for the common `[string]` EDL attribute, since the amount of loop operations performed by `strlen()` reveals the number of non-zero bytes following the specified in-enclave pointer.

Our attack builds on top of the open-source SGX-Step [257] enclave interrupt framework to turn the subtle `strlen()` side-channel leakage into a fully *deterministic oracle* that reveals the exact position of all `0x00` bytes in enclave private memory (thereby, for instance, fully breaking the confidentiality of

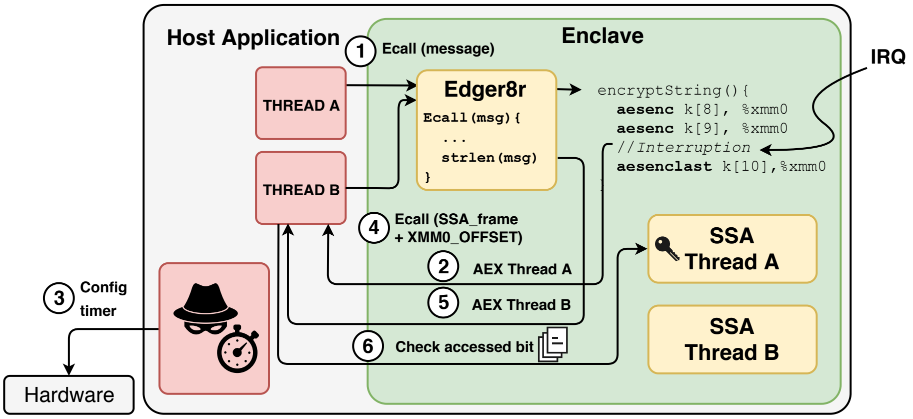


Figure 2.4: Overview of the key extraction attack exploiting `strlen()` side-channel leakage in Intel SGX SDK.

booleans or providing valuable information for cryptanalysis). Particularly, we use SGX-Step to reliably step the `strlen()` execution, one instruction at a time, leveraging the "accessed" bit in the page-table entry of the targeted in-enclave memory location as a noise-free oracle that is deterministically set by the processor for every `strlen()` loop iteration [258]. We confirmed that our single-stepping oracle continues to work reliably even when the victim enclave was compiled to a single, extremely compact `rep movsb` instruction (x86 string operations can indeed be interrupted in between every loop iteration [114]).

We developed a practical end-to-end AES-NI key extraction PoC in an application enclave built with a vulnerable version of `edger8r`. Our victim enclave provides a single, multi-threaded `ecall` entry point that encrypts the first 16 bytes of a given string using side-channel resistant AES-NI instructions with a secret in-enclave key. Since AES-NI operates exclusively on CPU registers (e.g., `xmm0`) and due to the limited nature of the `strlen()` side channel, we cannot perform key extraction by directly targeting the AES state or key in memory. Instead, our attack uses repeated encryption `ecalls`, assuming varying (but not necessarily known) plaintext and known ciphertext. We further abuse that the Intel SGX architecture enables a privileged adversary to precisely interrupt a victim enclave at a chosen instruction-level granularity [257], thereby forcing the processor to write the register state to a fixed SSA location in enclave memory (this includes the `xmm` registers that are part of the `XSAVE` region of the SSA frame). Figure 2.4 depicts the high-level phases of the attack flow, using two threads A and B:

Algorithm 1 `strlen()` oracle AES key recovery where $S(\cdot)$ denotes the AES SBox and $SR(p)$ the position of byte p after AES ShiftRows.

```
while not full key $K$ recovered do
    $(P, C, L)$ ← random plaintext, associated ciphertext, strlen oracle
    if $L < 16$ then
        $K[SR(L)] \leftarrow C[SR(L)] \oplus S(0)$
    end if
end while
```

(a) Invoke the encryption `ecall` from thread *A* ① and interrupt the enclave ② before the final round of the AES (*i.e.*, before the `aesenclast` instruction). To keep the PoC simple, we achieve this requirement by inserting an access to a dummy page at the appropriate point, and catching accesses to this page in a signal handler on the untrusted side. Note that in a real-world attack, the single-stepping feature of SGX-Step could be used to execute the victim enclave exactly up to this point, without relying on a more coarse-grained page fault for interruption.

(b) While the `ecall` in thread *A* is interrupted, prepare the timer used by SGX-Step ③ and launch a second thread *B* ④ to probe the position of the first zero byte (if any) in the intermediate AES state. Concretely, this involves a second `ecall` to the same entry point, but this time supplying an illegal in-enclave target address pointing to the fixed memory location containing the `xmm0` register in the SSA frame of the interrupted thread *A*. Each time when a timer interrupt arrives ⑤, we monitor and clear ⑥ the "accessed" bit of the targeted SSA page-table entry.

(c) After the `strlen()` probing has finished, the obtained leakage is stored alongside the corresponding ciphertext, and thread *A* is resumed by restoring read/write access to the dummy page.

(d) Repeat from step (a) with a different plaintext until the full key has been recovered (see Algorithm 1).

Experimentally, we determined that this attack succeeds with 881 AES invocations on average (over 1000 runs with random keys, minimum: 306, maximum: 3346), given a deterministic, noise-free `strlen()` oracle. Note that this attack could also be adapted to work with noisy measurements, using the so-called zero-value model known from hardware side-channel attacks [75]. Besides, the attack would also be applicable when targeting the first round of the AES in a known-plaintext scenario.

Properly closing this side channel requires profound changes in the way `edger8r` works. Notably, the bridge code includes an `lfence` instruction at line 11 to rule out advanced Spectre-v1 misspeculation attacks that might still speculatively compute on unchecked pointers before they are architecturally rejected. However, our attack is immune to such countermeasures because we directly observe side effects of normal, non-speculative execution. Further, early rejecting the `ecall` when detecting that the start pointer falls inside the enclave does not suffice in general. In such a case, adversaries might still pass pointers below the enclave base address, and observe secret-dependent behavior based on the first bytes of the enclave. Intel implemented our recommended mitigation strategy by dropping support for the superfluous `[sizefunc]` EDL attribute entirely, and further abstaining from computing untrusted buffer sizes inside the enclave. Instead, alleged buffer sizes are computed outside the enclave, and passed as an *untrusted* argument, such that the `CHECK_UNIQUE_POINTER` test can take place immediately. For the `strlen()` case, the untrusted memory can simply be copied inside, and an extra null byte inserted at the alleged end. This solution conveniently moves all secret-dependent control flow from the enclave into the untrusted application context.

Microsoft Open Enclave SDK. After Intel had properly patched the `strlen()` side-channel vulnerability in the SGX SDK, OE appears to have tried to adopt our proposed mitigation strategy of passing an *untrusted* alleged string length into the enclave. However, after reviewing the generated code, we found that `oeedger8r` fails to include a `0x00` terminator byte after copying the untrusted string inside enclave memory (cf. Appendix B.7). This critical oversight can be exploited to trick the shielded enclave application into operating on non-null-terminated strings. The trusted user function will incorrectly assume that the string is properly terminated and may perform out-of-bounds memory read/writes, hence turning a mitigation for a subtle and functionally correct side-channel issue into a more dangerous source of enclave memory corruption. This OE vulnerability is tracked via CVE-2019-0876 and specific to enclaves that expect EDL string arguments, and output or manipulate them in-place, e.g., `strcpy()`.

We experimentally demonstrated this vulnerability by means of a minimal PoC application enclave which overwrites all non-alphanumeric chars in a string with `0x20`, until the null terminator is encountered. If this enclave operates on an unterminated string, the length field of the subsequent heap frame is corrupted, which subsequently can be further leveraged in more complex exploits.

2.5.3 Validating variable-sized buffers

Multi-byte input buffers are commonly specified by passing a pointer to the start of the buffer and an associated size. In order to properly validate such buffers, the trusted runtime should first compute the end pointer by adding the alleged size argument, and thereafter assert that the complete input buffer address range falls outside the enclave. However, since the buffer size is an adversary-controlled parameter, care should be taken to prevent the pointer addition from overflowing and silently wrapping around the address space.

Attack vector #6 (integer overflow): Runtimes should use safe arithmetics when computing addresses in a buffer with untrusted size. ▷ Relatively well-understood in production SDKs, not in research code.

TEE design. We found that the address-related vulnerabilities in this section are significantly more exploitable in TEE designs that provide increased attacker control over the shared memory and enclave memory layouts. For instance, some integer overflow vulnerabilities require the adversary to control the enclave base address in a shared address space, as is the case for the Intel SGX [47] and Sancus [189] designs, but not for ARM TrustZone [205] or Keystone [153]. Further, we found that logical errors may arise when checking variable sized buffers in a shared address space. As detailed below, the exploitability of such logic bugs depends heavily on the ability of the adversary to trigger certain edge cases (e.g., passing a pointer that lies just before the enclave base address), which might also be considerably easier in single-address space TEE designs like Intel SGX or Sancus.

Fortanix Rust-EDP. In contrast to the other runtimes described in this chapter, Fortanix's EDP [68] leverages the type system of the safe Rust language to disallow inadvertent untrusted pointer dereferences apart from the dedicated `UserSafe` type, which transparently sanitizes any pointers passed into the enclave. Rust-EDP's shielding system has been explicitly designed to avoid known enclave boundary attacks and implements libOS-like functionality through a deliberately very narrow `ocall` interface that is kept invisible to the application programmer. However, our analysis shows that the promising approach of enforcing pointer sanitization through the use of a type system may evidently still suffer from security issues if the implementation in the type itself is incorrect.

We manually scrutinized the implementation of the confined `UserSafe` type (part of the Rust compiler's SGX-EDP target [68]) and found a potentially exploitable

```
1 /// `true` if the specified memory range is in userspace.
2 pub fn is_user_range(p: *const u8, len: usize) -> bool {
3   let start = p as u64;
4 ★ let end = start + (len as u64);
5   end <= image_base() || start >= image_base() +
6          (unsafe { ENCLAVE_SIZE } as u64) // unsafe ok: link-time constant
7 }
```

Listing 2.4: Pointer validation in the Rust-EDP `UserSafe` type.

integer overflow vulnerability in the pointer validation logic. Listing 2.4 shows the relevant `is_user_range()` function, which checks whether an untrusted memory range specified by a pointer and length falls completely outside the enclave. Concretely, we observed that the 64-bit integer addition to compute the `end` pointer at line 4 may overflow. Note that Rust can automatically detect integer overflows, but these runtime checks are only enabled in debug mode, meaning that in production builds (e.g., `rustc -C debug-assertions=off`), integer overflows do not cause an error by default [157].

We confirmed (after isolating the validation function in a dummy Rust test program) that said function can be made to early-out and return `true` at line 5 even when passing an illegal in-enclave pointer if the enclave base is near the top of the address space. Note that Intel SGX leaves the enclave base address under explicit attacker control [47], so this requirement may be satisfied by real-world attackers. For example, the untrusted runtime can return a specially-crafted pointer from the `alloc()` usercall, potentially leading to in-enclave memory disclosure or corruption, depending on how the pointer is further used within the enclave. After our disclosure, the EDP trusted runtime now explicitly asserts that untrusted sizes returned by `alloc()` do not overflow.

Google Asylo. Apart from the aforementioned `[user_check]` issues, the entry points in Asylo's trusted runtime take care to validate all second-level input buffers. However, our code review also revealed a subtle logic mistake in the input validation logic itself. That is, we observed that many of the trusted runtime functions (cf. Appendix B.3 for a relevant sample) rely on the `TrustedPrimitives::IsTrustedExtent(input, input_size)` library function returning `true` to reject the `ecall` attempt when detecting that an untrusted input buffer is *completely* contained within enclave memory.

Attack vector #7 (outside $\neq$ $\neg$inside): In a shared address space, input buffers should not fall *partially* inside the trusted memory region. ▷ Generally understood in production SDKs, not always in research code.

While this function itself translates to the corresponding `sgx_is_within_-enclave()` primitive from the SGX SDK, which is indeed correct and free from integer overflow vulnerabilities, the logic mistake occurs when considering malicious input buffers that only partly *overlap* with untrusted and enclave memory. For instance, `IsTrustedExtent()` will properly return `false` and the `ecall` will still be allowed when passing a lengthy adversarial input buffer that starts one byte before the enclave base address but continues into the enclave memory range. Evidently, this may subsequently lead to trusted enclave memory corruption or disclosure. Hence, the trusted runtime should instead make use of the proper `sgx_is_outside_enclave()` SGX SDK primitive.

Graphene-SGX. We discovered a critical integer overflow vulnerability in the widely used pointer range validation function that often computes on untrusted attacker-provided sizes (similar to the Rust-EDP issue described above). We further found that Graphene-SGX suffers from the same subtle logic mistake that we spotted in the Asylo code base: at the time of our review, there was no `sgx_is_outside_enclave()` primitive, and all instances of the intended "abort if not completely outside" were erroneously checked for "abort if completely inside enclave" (cf. Listing 2.5 for a relevant sample). A related type of pointer validation vulnerabilities arises when the libOS allocates variable-sized output buffers in untrusted memory outside the enclave to be able to exchange data for `ocall` arguments and return values. For performance reasons, Graphene-SGX allocates such shared memory buffers directly on the untrusted host stack. While the untrusted host stack pointer is indeed validated to lie outside of enclave memory upon enclave entry, we observed that the trusted libOS does *not* properly check whether the untrusted stack does not overflow into enclave memory after allocating a new shared memory buffer in the widely used `OCALLOC` macro. Depending on the specific `ocall` implementation, the enclave will subsequently copy data to/from the inappropriately allocated buffer, leading to information disclosure and/or memory corruption.

Keystone. While Keystone [153] is still a research prototype and lacked essential functionality when we reviewed its code, we discovered and reported a potential integer overflow vulnerability (cf. Appendix B.8) in the trusted security monitor's `detect_region_overlap()` function, which is used during the creation of an enclave. However, this overflow was not directly exploitable due to certain restrictions on region sizes in the Keystone codebase.

Sancus. We found both logical errors and integer overflow vulnerabilities in the `sancus_is_outside_sm()` function provided by the trusted runtime.

Particularly, the current implementation does not properly detect an untrusted buffer that spans the entire enclave address range, or a rogue length specifier that triggers an integer overflow to wrap around the 16-bit address space.

2.5.4 Pointer-to-pointer validation pitfalls

While the previous sections have focused on the *spatial* aspect of untrusted pointer dereferencing, we also found more subtle vulnerabilities related to the *temporal* aspect. That is, whenever a pointer points to an untrusted address or size (as it is often the case, for instance, in marshalling structs), the runtime should take care to first copy the second-level pointer value to a trusted location in enclave memory before applying the sanitization logic. If this is not the case, adversaries may overwrite the second-level pointer in untrusted memory after the validation has succeeded but before the pointer is dereferenced in the enclave code. This class of vulnerabilities is also referred to as "double fetch" bugs in operating system kernels [276, 218].

Attack vector #8 (double fetch): Untrusted pointer values should be copied inside the enclave *before* validation to avoid time-of-check time-of-use. ▷ Relatively well-understood (once pointer sanitization is applied).

TEE design. Double fetch bugs typically rely on a very narrow vulnerability time window and hence can be notoriously hard to exploit in traditional user-to-kernel contexts. However, recent research demonstrated how some TEE design decisions may considerably simplify exploitation of synchronization bugs in enclaves. AsyncShock [266] exploits that Intel SGX adversaries may provoke page faults in the enclaved execution, and SGX-Step [257] similarly abuses that privileged SGX adversaries may abuse system timers to very precisely interrupt a victim enclave after *every* single instruction. Finally, Schwarz et al. [218] use a cache side channel to expose double fetch bugs in both Intel SGX and ARM TrustZone TEEs.

Graphene-SGX. Scrutinizing Graphene-SGX's `ocall` interface, we found several instances of exploitable double fetch vulnerabilities. Listing 2.5 provides a relevant code snippet that attempts to sanitize the result of the `sock_accept` system call. First, at line 1, a buffer `ms` is allocated in *untrusted* memory outside the enclave. The `struct` buffer pointed to by `ms` contains another pointer `ms->ms_addr` that will be initialized by the untrusted runtime to point to the socket address returned by the system call. As `ms->ms_addr` is an untrusted pointer, the libOS shielding system attempts to properly validate

```
1    OCALLOC(ms, ms_ocall_sock_accept_t *, sizeof(*ms));
2    ...
3    retval = SGX_OCALL(OCALL_SOCK_ACCEPT, ms);
4    if (retval >= 0) {
5 ★    if (len && (sgx_is_within_enclave(ms->ms_addr, len)
6                  || ms->ms_addrlen > len)) {
7        OCALL_EXIT();
8        return -PAL_ERROR_DENIED;
9      }
10     ...
11 ★   COPY_FROM_USER(addr, ms->ms_addr, ms->ms_addrlen);
```

Listing 2.5: Double fetch vulnerability in Graphene-SGX.

that it lies outside the enclave at line 5 (modulo the logic bug described in Section 2.5.3) before dereferencing `ms->ms_addr` a second time when copying the socket address buffer inside at line 11. However, since the parent `ms struct` was allocated in untrusted memory and has never been copied inside, SGX adversaries can interrupt the enclave in between lines 5 and 11 and trivially overwrite the `ms_addr` field with an arbitrary in-enclave address, potentially leading to trusted memory disclosure.

2.5.5 Validating ocall return values

Apart from validating `ecall` arguments, the enclave trusted runtime should also take care to properly scrutinize `ocall` return values when passing pointers or sizes back into the enclave.

Attack vector #9 (Iago): Pointers or sizes returned through `ocalls` should be scrutinized [38]. ▷ Understood, but still prevalent in research libOSs that shield system calls; one instance in a production SDK.

TEE design. We found that the complexity of the shielding system may largely affect this attack surface. That is, SDK-based approaches typically do not feature a large built-in `ocall` interface, whereas libOSs should safeguard against Iago attacks [38] by scrutinizing return values from the complex system call interface before passing them on to the shielded application.

Microsoft Open Enclave SDK. OE's trusted runtime includes a `oe_get_-report()` function which is used to provide attestation functionality to the enclaved binary. Internally, this function performs the same `ocall` twice; the first time specifying the output buffer as a null pointer in order to obtain the required quote size. Based on this size, a buffer is allocated on the enclave heap, and subsequently filled through a second `ocall` invocation. We found, however, that the untrusted runtime can return different sizes for the two `ocall` invocations (tracked via CVE-2019-1369). Particularly, the in-enclave buffer is allocated based on the size obtained from the first `ocall`, whereas the size returned by the second `ocall` is passed on to the caller of `oe_get_report()`. Hence, returning an unexpectedly large size in the second `ocall` invocation may cause the enclave application to read or write out of bounds. We experimentally confirmed that OE's remote attestation example enclave can leak up to 10 kB of trusted heap memory (this upper bound is due to an internal limit), possibly at multiple heap locations depending on other memory allocations.

LibOS-based runtimes. We discovered several exploitable instances of Iago attacks [38] in Graphene-SGX's `ocall` interface. For example, an untrusted system call return value `len` is later used to copy `len` bytes from untrusted memory into a fixed-size buffer inside the enclave, leading to arbitrary write-past the in-enclave buffer. To demonstrate this vulnerability, we developed a PoC where the `readdir()` system call in the untrusted runtime returns an unexpected length, causing an out-of-bounds write in the enclave.

Similarly, in SGX-LKL's `ocall` interface, we found several instances of Iago vulnerabilities where for example the untrusted pointers returned by `mmap()` are not checked to lie outside of enclave memory, or the untrusted length returned by `write()` is passed unsanitized back to the shielded application. To demonstrate how this can be successfully exploited, we developed an elementary victim application featuring a common programming idiom where `write()` is used to output a buffer piecewise, each time advancing a pointer with the number of bytes successfully written (*i.e.*, the system call's return value). We modified the untrusted runtime to unexpectedly increment the return value of the `write()` system call, causing the shielded application binary to output secret enclave memory beyond the buffer bounds. Finally, we also confirmed and reported the existence of similar issues in Google Asylo.

Keystone. Similar to the above SGX runtimes, Keystone provides system call wrappers to simplify porting of existing code to an enclave. While Keystone documentation indicates that the developers are aware of potential issues, the codebase currently lacks mitigations against Iago attacks. Hence, we developed an exploit using the `write()` system call, similar to the SGX-LKL PoC.

2.5.6 Scrubbing uninitialized structure padding

Apart from pointers and size arguments, enclaves may also pass composite `struct` types to the untrusted world. While, as with all output buffers, we assume that enclave applications do not intentionally disclose secrets through the program-visible state (*i.e.*, the `struct`'s individual members), prior research on operating system kernel [45] and SGX enclave [155] interfaces has shown that padding bytes silently added by the compiler may still unintentionally leak uninitialized secret memory.

Attack vector #10 (uninitialized padding): Scrubbing program-visible state may not suffice for `struct` outputs [155]. ▷ Especially relevant for production SDKs that expose the enclave interface to the programmer.

TEE design. This subtle attack vector cannot be easily mitigated by sanitizing program-visible API state. Possible mitigations include securely initializing the entire output `struct` using `memset()` and/or doing a member-wise deep-copy, or declaring the output `struct` as "packed" so the compiler does not unknowingly introduce padding. However, both solutions require application-specific knowledge about the exact `struct` types being passed. As an important insight, we therefore found that this attack vector can only be transparently shielded when the enclave interface is predefined and fixed. That is, the fixed `ocall` interface in libOS-based runtimes can indeed be manually scrutinized for this type of vulnerabilities. However, this is not the case for SDK-based runtimes that offer a *generic* enclave interface defined by the programmer, and hence (opposed to their shielding responsibility) ultimately outsource the responsibility of scrubbing uninitialized `struct` padding to the application developer.

SDK-based runtimes. Lee et al. [155] first demonstrated how uninitialized `struct` padding may pose a subtle information leakage source in the `edger8r`-generated code of the Intel SGX SDK. Building on their findings, we generalized this attack vector to also demonstrate its applicability to `oeedger8r`-generated code in Microsoft's Open Enclave SDK, as well as in the Sancus TEE. Similarly, we confirmed that padding leakage can also occur in Keystone, e.g., through the padding of `calc_message_t` in the demo enclave.

LibOS-based runtimes. We reviewed the `ocall` interfaces in the libOS-based runtimes we studied (Graphene-SGX, LKL, Rust-EDP). Rust-EDP appears to be free of such issues, and Graphene-SGX explicitly enforces `struct` packing through a compiler `#pragma`. However, SGX-LKL contains at least two instances of an `ocall` using a `struct` with potentially vulnerable padding bytes (`sigaction` and `siginfo_t`). In Google Asylo, most `structs` passed through an `ocall` are explicitly declared as packed, however, we found one instance of a padded `struct BridgeSignalHandler` used in the syscall interface.

2.6 Discussion and general mitigations

The most intuitive solution to defend against our attacks is to incorporate additional checks in the enclave code to properly sanitize ABI state and API arguments/return values. When properly implemented, such checks suffice to block all of the attacks described in this work, and they have indeed been adopted by the various projects we analyzed. However, leaving the decision of whether (and how) to correctly implement numerous interface validation checks to enclave developers, who are likely unaware of this class of vulnerabilities, may be problematic. Moreover, even when developers think about inserting the necessary checks, our analysis has revealed several recurring pitfalls, including subtle logical bugs, side channels, double fetches, and integer overflows. This highlights the need for more principled approaches to rule out this class of vulnerabilities at large, as well as defense-in-depth code hardening measures that may raise the bar for successful exploitation.

Code hardening. Interface sanitization vulnerabilities are closely related to a wider class of memory safety issues [154, 24], and their exploitation may hence be partially hindered by established techniques such as heap obfuscation (cf. Section 2.4.1). Furthermore, SGX-Shield [227] aims to obstruct memory corruption attacks by randomizing the memory layout of enclaved binaries shielded by the Intel SGX SDK. However, prior research [24] has shown that SGX-Shield does *not* randomize the trusted runtime, meaning that the code we studied would still feature a deterministic and static memory layout, and may offer numerous gadgets for mounting code reuse attacks. Further, as the trusted runtime also forms an integral part of SGX-Shield's loader [227], any memory safety or side-channel vulnerabilities in the trusted runtime itself may also be used to disrupt the preliminary randomization stage. While randomizing the memory layout of the trusted runtime would indeed be desirable, this constitutes a non-trivial task [24, 227] given its low-level nature, including hand-written assembly code and static memory addresses expected by SGX's `eenter` and `eresume` instructions. In this respect, we want to emphasize that some of the attacks we presented are free from non-static address dependencies, and hence remain inherently immune to software randomization schemes. For example, the SGX SDK `strlen()` oracle in Fig. 2.4 depends solely on the *fixed* address of the victim's SSA frame, which is deterministically dictated by the SGX hardware and immutable from software.

As a perpendicular code hardening avenue, we recommend to implement more aggressive responses when detecting pointer violations in the trusted runtime. That is, most of the runtimes we studied merely reject the `ecall` attempt when detecting pointer poisoning. In the SGX SDK `strlen()` oracle attack of Section 2.5.2, we for example abused this to repeatedly call a victim enclave, each time passing an illegal pointer and making side-channel observations before the `ecall` is eventually rejected. To rule out such repeated attacks, and reflecting that in-enclave pointers represent clear adversarial or buggy behavior, we recommend to immediately destroy secrets and/or initiate an infinite loop upon detecting the *first* pointer poisoning attempt in the trusted runtime.

Hardware-assisted solutions. As a more principled approach to rule out the confused deputy attacks described in this chapter, solutions could leverage finer-grained memory protection features in the processor. In particular, tagged memory [267] or capability architectures [275] appear to be a promising approach to inherently separate the memory domains of untrusted and trusted code. On a capability machine [275], pointers are represented at run-time as unforgeable objects carrying associated permissions and length fields. The machine ensures that untrusted code can never create a valid capability that points inside enclave-private memory and pass it as an argument to an `ecall`, thereby eradicating an entire class of pointer dereference vulnerabilities architecturally.

As an example of an alternative tagged memory design, the recently proposed Timber-V [267] architecture provides lightweight and strong enclaved execution on embedded RISC-V platforms. Timber-V processors offer enhanced MPU isolation by keeping track of a 2-bit tag for every memory word, allowing individual memory locations to be associated with one out of 4 possible security domains. The CPU further restricts tag updates, and offers checked memory load/store operations, which take an *expected* tag as an argument and trap whenever the actual memory location being dereferenced does not match the expected tag. Hence, any pointer poisoning attempts by untrusted code outside the enclave would be immediately caught by the hardware.

The untrusted pointer dereference issues we identified in this work bear some similarities with how privileged OS kernel code needs to properly sanitize user space pointers in e.g., system call arguments. As a defense-in-depth mechanism, recent x86 processors support Supervisor Mode Access Prevention (SMAP) features to explicitly disallow unintended user space pointer dereferences in kernel mode [114]. We encourage further research to investigate porting such CPU features to enclave mode.

Safe programming languages. The combination of TEEs and safe programming languages, such as Rust, has been proposed as a promising research direction to safeguard enclave program semantics, but still requires additional interface sanitizations [72]. The approach of Fortanix's Rust-EDP [68] shows how the compiler's type system can be automatically leveraged to limit the burden of pointer sanitization concerns from a cross-cutting concern throughout the enclave code base to the correct implementation of a single untrusted pointer type. However, it is important to note that safe languages by themselves are not a silver bullet solution to our attacks. That is, the trusted runtime code remains responsible to bootstrap memory safety guarantees by *(i)* establishing expected ABI calling conventions in the low-level entry assembly code, and *(ii)* providing a correct implementation of sanitization in the untrusted pointer type. In this respect, the subtle integer overflow vulnerability in Fortantix's EDP, presented in Section 2.5.3, demonstrates that developing both the trusted runtime libraries and the enclave in safe Rust may still not suffice to fully eradicate pointer sanitization vulnerabilities.

Finally, as an alternative to Intel's `edger8r` tool, the use of separation logic has been proposed to automatically generate secure wrappers for SGX enclaves [73]. This approach aims to provide the advantages of safe languages, and even formal verification guarantees, but still relies on explicit developer annotations.

2.7 Conclusions and future work

Our work highlights that the shielding responsibilities in today's TEE runtimes are not sufficiently understood, and that various security issues exist in the respective trusted computing bases. We showed that this attack surface is large and often overlooked: we have identified 35 interface sanitization vulnerabilities in 8 open-source TEE runtimes, including production-quality SDKs written by security-savvy developer teams. Our analysis further reveals that the entry points into this attack surface are more pervasive than merely argument pointers: we contributed a classification of 10 recurring vulnerability classes spanning the ABI and API tiers.

In the defensive landscape, our work emphasizes the need to research more principled interface sanitization strategies to safeguard the unique TEE shielding responsibilities. We particularly encourage the development of static analysis tools, and fuzzing-based vulnerability discovery and exploitation techniques to further explore this attack surface.

Chapter 3

Stealthy page-table-based attacks on enclaved execution

This chapter was previously published as:

> J. Van Bulck, N. Weichbrodt, R. Kapitza, F. Piessens, and R. Strackx. "Telling your Secrets Without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution". In: *26th USENIX Security Symposium*. Aug. 2017, pp. 1041–1056

Preamble

This chapter presents a new type of stealthy side-channel attack techniques that enable an untrusted operating system to observe enclave page accesses without resorting to page faults. Particularly, we contribute two novel attack vectors that infer enclaved memory accesses from page-table attributes, as well as from the caching behavior of unprotected page-table memory. We demonstrate the effectiveness of our attacks by recovering full EdDSA session keys with little to no noise from the popular Libgcrypt cryptographic software suite.

This research was the first to recognize the dangers of traversing enclave page tables in untrusted operating system memory, beyond observing obvious page fault exceptions. Our work, therefore, highlights the deficiency of initial controlled-channel defenses [230, 229], which focus on merely detecting page-fault events, and calls attention to the untrusted page-table walk itself. This observation makes page-table-based attacks very challenging to mitigate in a principled way. We discuss various proposals and their trade-offs in Chapter 8.

In the context of a wider study of SGX memory access side channels, Wang et al. [263] concurrently developed similar stealthy page-table attacks, yet only monitoring A/D page-table attributes and at a significantly lower temporal resolution without exploring the cache timing channel.

The stealthy page-table spy techniques described in this chapter have since been leveraged in an ongoing line of SGX side-channel attacks [257, 256, 254, 139, 6, 182]. Notably, the observation that the processor's page-miss handler may cause contention in the CPU cache has even been generalized to non-SGX attack scenarios as well [81, 259]. With our innovative application of FLUSH+FLUSH [87] to monitor enclave page-table accesses, we established a novel high-resolution and low-noise cache timing attack technique which for the first time exploits increased timing differences from writing back *modified* cache lines. Our work furthermore pioneered the security analysis of "accessed" and "dirty" page-table attributes, which later turned out to be effective not only for side-channel analysis, but also as a microcode assist primitive to trigger transient-execution data sampling attacks [223, 35]. Likewise, as explained in Chapter 6, the observation that SGX permissions are only applied *after* the untrusted page-table walk has completed (cf. Fig. 3.1) was later leveraged to evade SGX abort page semantics in the Foreshadow attack.

In anticipation of SGX-Step, introduced in Chapter 4, this chapter for the first time explored the temporal dimension of the paging channel by showing that enclaves can be precisely preempted at instruction-level granularity using inter-processor interrupts. The tandem between interrupts and page-table interactions has since become a reoccurring element in the SGX attack landscape, as discussed in Chapter 8. Our recent work on CopyCat [182] made this even more explicit by combining the techniques described in this chapter with SGX-Step single-stepping interrupts to deterministically reconstruct enclave control flow *within* a single 4 KiB code page. Intuitively, CopyCat shows that the stealthy page-table spy techniques introduced in this chapter are not only equivalent to, but strictly stronger than page-fault controlled channels.

To mitigate the EdDSA attack described in this chapter, we contributed a minimal patch that has been merged in Libgcrypt v1.7.7 [247]. We furthermore publicly released all of the attack code and experimental data, and we have been in contact with several independent research groups that successfully reproduced our findings. At least one published work [198] explicitly ran our open-source attack code to evaluate an improved defense technique.

3.1 Introduction

Enclaved execution is a promising new security paradigm that makes it possible to execute application code on a platform without having to trust the underlying operating system or hypervisor. With the advent of Intel SGX [176], support for Trusted Execution Environments (TEEs) is now available on mainstream consumer hardware, and can be used to defend against malicious or compromised system software, both in an untrustworthy cloud environment [20, 217] as well as for desktop applications [99]. In particular, one line of research has developed techniques and supporting software to make it relatively easy to run unmodified legacy applications within an enclave [20, 17, 231, 246].

An essential aspect of enclaved execution is that the hardware prevents privileged system software from reading or writing an enclave's private memory directly, or from tampering with its internal control flow. However, the OS remains in charge of allocating platform resources (memory pages and CPU time), such that the platform can be protected against misbehaving or buggy enclaves. One consequence of this interaction between privileged system software and enclaves is an entirely new class of powerful, *indirect* attacks on enclaved applications. Xu et al. [277] first showed how a malicious OS can use page faults as a noise-free *controlled channel* to extract rich information (full text and images) from a single run of a victim enclave. This is particularly dangerous when legacy software is running within an enclave, as these applications have not been hardened against side-channel attacks. As a result, several authors have expressed their concerns on side-channel vulnerabilities in a TEE setting in general, and the page fault channel in particular [63, 47, 245, 229, 42].

The research community has since proposed a number of compile-time and hardware-enabled defense techniques [230, 48, 229] that hide enclave page accesses from the OS. We argue, however, that page faults are but one side effect of the address translation process that is observable by untrusted system software. More specifically, the main contribution of this chapter is that we show that an adversarial OS can infer page accesses from an enclaved execution that *never* suffers a page fault. Our attacks exploit the key property that the SGX design leaves page-table memory under explicit control of the untrusted OS. As such, other side effects of the page-table walk in enclave mode can be observed by the OS with very little to no noise. We identify and successfully exploit straightforward effects such as the setting of "accessed" and "dirty" bits, as well as less obvious effects such as the caching of page-table memory itself. An important consequence is that our novel attack vectors bypass recent defenses that focus exclusively on suppressing page faults [230, 229].

In summary, the contributions of this chapter are:

- We advance the state-of-the-art by defeating recently proposed defense techniques, showing that we can infer page accesses without resorting to page faults.
- We present a page-table-based technique to precisely interrupt an enclave at instruction-level granularity.
- We implement our novel attack vectors as an extension to Graphene-SGX's untrusted runtime, facilitating eavesdropping on unmodified applications.
- We demonstrate the effectiveness of our attacks by extracting private EdDSA session keys from the widely used Libgcrypt cryptographic library.

Our attack framework and evaluation scenarios are available as free software, licensed under GPLv3, at `https://github.com/jovanbulck/sgx-pte`.

3.2 Background and state-of-the-art

In this section, we provide the necessary background on Intel SGX, refine the attacker model, and discuss previous research results on controlled-channel attacks.

3.2.1 Intel SGX

Recent Intel x86 processors from Skylake onwards are being shipped with Software Guard Extensions (SGX) [176, 14, 114] that enable strong, hardware-enforced trusted computing guarantees in an untrusted execution environment. SGX extends the instruction set and memory access logic of the Intel architecture to allow the execution of security-sensitive application logic in protected *enclaves* in isolation from the remainder of the system, including privileged OS or hypervisor.

Memory protection. An SGX-enabled processor sets aside a contiguous physical memory area, referred to as Processor Reserved Memory (PRM). A hardware-level memory encryption engine guarantees the confidentiality, integrity, and freshness of PRM memory while it resides outside of the processor package. The PRM region is subdivided into two data structures: the Enclave Page Cache (EPC) and the Enclave Page Cache Map (EPCM). Protected 4 KiB enclave code and data pages are allocated from the EPC, while every EPC page has a shadow entry in the EPCM to track ownership, type, address

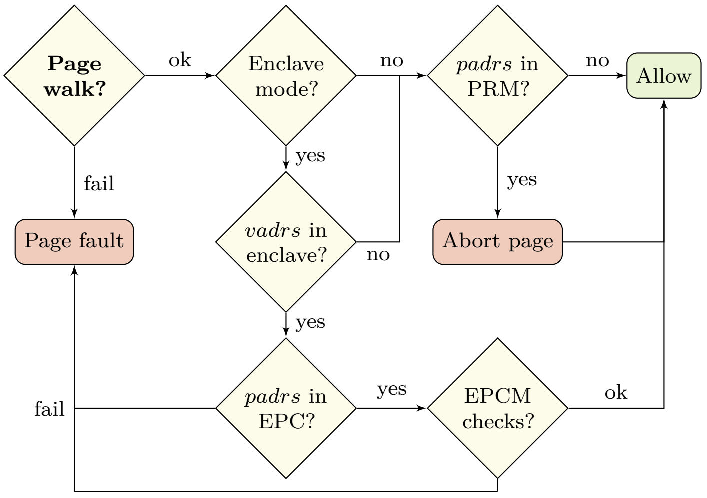


Figure 3.1: Additional memory access checks performed by SGX for a virtual address *vadrs* that maps to a physical address *padrs*.

translation, and permission metadata. EPCM memory is exclusively managed by the processor, and is never directly accessible to software.

SGX enclaves are instantiated as part of the virtual address space of a conventional OS process. Since PRM is a limited system resource, untrusted system software is in charge of assigning protected memory pages to enclaves, and is allowed to oversubscribe the EPC. At enclave creation time, the OS can instruct the processor to initialize newly allocated EPC pages with unprotected code or data. After finalizing the enclave, and before it can be entered, the hardware calculates a secure hash of the enclave's initial state. This allows the integrity of the untrusted loading process to be attested to a remote stakeholder [14]. SGX furthermore offers dedicated ring-zero instructions to securely evict and reload enclave pages between EPC memory and untrusted storage.

An important design decision of SGX is that it leaves page tables under explicit control of the untrusted operating system. Instead, SGX implements an additional, independent layer of access control on top of the legacy page-table-based memory protection mechanism. Figure 3.1 summarizes the additional checks performed when accessing enclave memory. First, in order to translate the provided virtual address to a physical one, the processor traverses the OS-managed page tables, as well as the extended page tables set up by the hypervisor, if any. As usual, a page fault is signaled to the untrusted OS in case of a permission mismatch or missing page-table entry during address translation.

Any attempt to access the PRM region in non-enclave mode results in abort page semantics, *i.e.*, read `0xFF` and ignore writes. Likewise, in enclave mode, the processor is allowed to reference all memory that falls outside of the executing enclave's virtual address range, but abort page semantics apply when such an address resolves into PRM memory. Furthermore, a page fault is signaled to the untrusted OS for EPC accesses that either do not belong to the currently executing enclave, are accessed through an unexpected virtual address, or do not comply with the read/write/execute permissions imposed by the EPCM.

To speed up subsequent memory accesses, SGX employs the processor's Translation Lookaside Buffer (TLB) as a trusted cache of already checked page permissions. That is, SGX's memory access protection is entirely implemented in the Memory Management Unit (MMU) hardware that consults the untrusted page tables and the EPCM whenever a provided virtual address was not found in the TLB [176, 47]. SGX's security argument is based on the key observation that untrusted system software needs to interrupt the logical processor core before it can affect TLB entries. SGX therefore flushes the TLB and internal paging-structure caches whenever entering or exiting an enclave, and requires the OS to engage in a hardware-verified protocol that ensures proper TLB invalidation before evicting an EPC page.

SGX's dual permission lookup scheme prevents malicious system software from mounting active memory mapping attacks [47]. The output of the address translation process is considered untrusted, and the most restrictive of both permissions is applied. However, this design also implies that an attacker controlling page-table permissions can cause enclave code to cause page faults, and be notified when certain pages are accessed. This property lies at the basis of the page fault attacks described in Section 3.2.3.

Enclave entry and exit. SGX enclaves are embedded in the address space of an untrusted user mode application, and can be internally multithreaded. They have to be explicitly entered by means of a dedicated `eenter` instruction that switches the logical processor to enclave mode, and transfers control to a predetermined entry point in the enclave's code section. The untrusted application context can exchange data with the enclave via unprotected memory. A processor running in enclave mode can be switched back programmatically by invoking the `eexit` instruction, or in case of a fault or external interrupt, through a process known as Asynchronous Enclave Exit (AEX). Upon AEX, the processor securely stores the execution context and exit reason (exception number) in a predetermined State Save Area (SSA) inside the enclave, and replaces CPU registers with a synthetic state before transferring control to the untrusted OS exception handler specified in the Interrupt Descriptor Table

(IDT). In case of a page fault, SGX also takes care of zeroing out the twelve least significant bits of the faulting address, revealing only the page number, but not the 12-bit offset within that page.

Importantly, SGX enclave threads are unaware of interrupts by design, and have to be resumed explicitly by invoking `eresume` from the unprotected application context. The `eresume` instruction takes care of restoring the previously saved processor state, and redirects control flow to the instruction pointer specified in the SSA frame. SGX allows an enclave to register trusted in-enclave exception handlers with a cooperative OS. For this to work, `eenter` has to be explicitly called before `eresume`, so as to allow the previously interrupted enclave to inspect and modify its internal SSA frame. Since `eresume` cannot be intercepted however, an enclave has no way of enforcing its internal exception handler to be actually called.

SGX's exception model ensures that the untrusted operating system remains in control of shared platform resources such as memory or CPU time, and prevents direct information leakage of register contents. However, partial information on the enclave's internal state still leaks to the OS via exception vectors, and the access type and page base address in case of a page fault.

3.2.2 Attacker model and assumptions

The adversary's goal is to derive sensitive application data processed in an enclave. We assume the standard SGX threat model where an attacker has full control over privileged system software including the operating system and hypervisor. The attacker has full control over OS scheduling decisions; she can pin specific threads to specific CPU cores, and interrupt enclaves repeatedly. She can furthermore modify all non-enclaved parts of the application. Like previous SGX attacks [277, 230, 79, 225, 156], we finally assume knowledge of the (compiled) source code of the target application.

At the system level, we assume a classical MMU-based architecture where the system software maintains a multi-level page-table data structure in OS memory to control virtual to physical page mappings. We assume the OS is in control of enclave page mappings, whereas the TEE guarantees the confidentiality and integrity of enclave pages, and properly verifies address translations to protect against page remapping attacks. Importantly, in contrast to previously published controlled-channel attacks discussed below, we assume a *PF-oblivious* attacker model where any page faults in enclave mode are hidden from untrusted system software. Our notion of stealthiness thus requires an attacker to infer page access patterns from an enclaved execution that *never* suffers a page fault. In addition, to stay under the radar of remote attestation schemes [229] that require the user's approval for each enclave invocation, our stealthy adversary should extract information from a *single* run of the victim enclave.

3.2.3 Controlled-channel attacks

This section briefly revisits previous research on page fault-driven attacks and defenses. We first explain how sensitive information can be derived from an enclave's page fault behavior, and thereafter elaborate on recently proposed state-of-the-art defense techniques.

Tracking page faults. As explained above, a page fault during enclave execution triggers an AEX that hands over control to the untrusted operating system, revealing the base address of the faulting page. A malicious OS can exploit this property to obtain a page-level trace of enclave execution by clearing the "present" bit in the Page Table Entrys (PTEs) that form the enclave's virtual address space. For maximal information leakage, an adversary allocates at most one code page and up to two operand data pages at all times. Furthermore, the access type can be inferred by manipulating the "writable" and "execute disable" PTE attributes.

Seminal work by Xu et al. [277] first showed how to exploit the page fault side channel in a deterministic way. Their *controlled-channel* attacks exploit secret-dependent control flow and data accesses in unmodified legacy applications running on top of the SGX-based Haven [20] architecture. To overcome the coarse-grained page-level granularity, they observe that the *sequence* of preceding page faults can be used to uniquely identify a specific memory access. The controlled-channel attack relies on an exhaustive offline analysis of the target application binary to identify page fault sequences, and afterwards uses this information to extract rich information (full text and images) without noise from a single run of the victim enclave. Subsequent work by Shinde et al. [230] demonstrated that the page fault channel is sufficiently strong to extract cryptographic key bits from unmodified versions of OpenSSL and Libgcrypt.

Proposed defenses. Ferraiuolo et al. [63] propose the use of dedicated CPU instructions to prevent certain pages from being swapped out of the protected memory area. This defense technique overlooks however that page faults can also be caused by directly modifying PTE attributes controlled by the OS. Shinde et al. [230] introduce the notion of *PF-obliviousness* which requires that any information leaked via page fault patterns can also be learned from running the program without inducing any page faults. They propose a compiler-based solution called *deterministic multiplexing* to generate PF-oblivious programs that unconditionally access all code and data pages at the same level of the execution tree. Without developer-assisted optimizations however, their approach exhibits unacceptable performance overheads [230] in practical application scenarios, which is why they also propose a hardware-assisted solution. In the *contractual execution* model, an enclave agrees with the untrusted OS that a number of sensitive pages remain mapped in its address space. The hardware is modified to report page faults directly to the enclave, without OS intervention, so as to enable protected enclave programs to detect contract violations. The enclave's fault handler can decide to either forward the page fault to the OS, abort the enclave program, or perform a fake execution to hide the page fault completely.

It seems that Intel made a first step towards supporting contractual execution on SGX platforms. As per revision 2 of the SGX specification [114], AEX can optionally store information about page faults in the interrupted enclave's SSA frame. This allows an SGX enclave to register a trusted exception handler for page faults. As explained in Section 3.2.1, however, the unprotected application can trivially `eresume` an enclave without first calling its designated exception handler. That is, the SGX v2 design still leaves enclaves explicitly unaware of interrupts or page faults. In response, Shih et al. [229] present a pragmatic approach to contractual execution on SGX platforms. Their solution, called T-SGX, leverages hardware support for Transactional Synchronization Extensions (TSX) in recent Intel processors [114]. TSX was designed to synchronize the critical sections of multiple threads without the overhead of software-based locks. Code executing in a TSX transaction is aborted and automatically rolled back whenever encountering a cache conflict or exception. The security argument of T-SGX relies on the important property that a page fault during a TSX transaction immediately transfers control to a user-level transaction abort handler, without first notifying the OS. In case of an external interrupt on the contrary, the normal AEX procedure vectors to the OS, but TSX ensures that the in-enclave transaction abort handler is called on `eresume`. The T-SGX compiler wraps each basic block in a TSX transaction, and uses a carefully designed springboard page to hide page faults across transactions. Since TSX lacks hardware support to distinguish between page faults and regular interrupts in the abort handler, T-SGX restarts transactions by default, and only terminates the enclave program after counting too many consecutive aborts of the same transaction. Since the OS is made unaware of page faults, an adversary learns at most one page access by observing early program termination. T-SGX prevents reruns by requiring the remote enclave owner's consent before starting the enclaved application.

Note that T-SGX does *not* consider frequent enclave preemptions suspicious (up to 10 consecutive transaction aborts are allowed for each individual basic block).

After acceptance of our work, however, more recent research was published [42] that leverages TSX to not only hide page faults, but also monitor suspicious interrupt rates. We discuss this heuristic defense technique and its implications for our attacks in more detail in Section 3.6.

Finally, Costan et al. [48] present a hardware-software co-design called Sanctum that represents a more radical approach to eliminate controlled-channel attacks. Not only does Sanctum dispatch page faults directly to enclaves, but it also allows them to maintain their own virtual-to-physical mappings in a separate page-table hierarchy in enclave-private memory. As further explored in Section 3.6, this design decision effectively prevents directed page-table-based attacks from the OS. While Sanctum explicitly identifies information leakage from "accessed" and "dirty" page-table attributes as a motivation for enclave-private page tables, we are the first to provide an exploitation strategy and to explore the implications of this side channel.

3.3 Stealthy page-table-based attacks

In this section, we present the design of our novel page-table-based attacks. We first introduce two distinct ways in which a PF-oblivious attacker can detect page accesses after the enclave has programmatically been exited. Next, we present our approach to dealing with cached TLB entries for subsequent accesses to the same page. We finally explain how to infer conditional control or data flow in large programs by correlating subsequent page accesses in *page sets* as a more stealthy alternative to the page fault sequences introduced by Xu et al. [277].

3.3.1 Monitoring page-table entries

As a running example, consider the leftmost code snippet in Listing 3.1, where we assume that a and b reference different data pages. In the classical controlled-channel attack [277, 230], an adversary would revoke access rights on both pages before entering the enclave, and learn the secret input by observing a page fault on either a or b.

Our attacks are based on the important observation that a processor in enclave mode accesses unprotected page-table memory during the address translation process. The key intuition is to exploit side effects of the page-table walk to identify which page has been accessed. In the following, we show that an adversary with access to unprotected page-table memory can learn the secret

```
void inc_secret (int s) {
    if (s)
        *a += 1;
    else
        *b += 1;
}
```

```
int cmp_and_swap (int old, int new)
{
    if (*a == old)
        return (*a=new);
    else return *a;
}
```

Listing 3.1: Example code with secret-dependent data flow.

input without resorting to page faults, either explicitly via page-table attributes, or implicitly by observing cache misses.

A/D bits. Since memory is a limited system resource, swapping out pages is benign OS behavior. To help memory-management software make an informed decision, Intel x86 processors [114] explicitly provide insight into an application's memory usage via page-table attributes. The CPU's address translation logic sets a dedicated Accessed (A) bit whenever reading a page-table entry, and takes care to set the Dirty (D) flag the first time a page has been written to. A/D attributes are stored in kernel-space memory, alongside the physical address of the page being referenced by the corresponding PTE entry, and need to be explicitly cleared by software.

We experimentally confirmed that A/D bits are also updated in enclave mode. An adversary inspecting these PTE attributes after enclave execution is thus provided with a perfect, noise-free information channel regarding the accessed memory pages. She can furthermore unambiguously distinguish between read and write accesses to the same page. In our `inc_secret` example, the secret input is directly revealed through the "accessed" bit of the PTEs referenced by a respectively b. The right-hand side of Listing 3.1 provides a more subtle example where the data page referenced by a is first accessed, and thereafter either written to, or read again. An adversary can distinguish between these cases using the "dirty" PTE attribute. Note that a page fault-based attack could derive the same information using the "writable" attribute, if stealthiness is not a concern.

Cache misses. Since modern CPUs can process data an order of magnitude faster than it can be fetched from DRAM, they rely on on an intricate cache hierarchy to speed up repeated code and data accesses. Contemporary Intel CPUs [114] feature three levels of multi-way, set-associative caches for instruction/data memory, and a separate TLB plus specialized paging-structure caches to accelerate address translation. Cache memories introduce a measurable timing difference for DRAM accesses and enable a powerful class of microarchitectural side-channel attacks, for they are shared among all software running on the platform.

A reliable and powerful class of access-driven cache attacks based on the Flush+Reload [280] technique exploits the availability of physical shared memory between the attacker and the victim, as is often the case with shared libraries. Flush+Reload relies on the `clflush` instruction that invalidates from the entire cache hierarchy all entries corresponding to a specified virtual address. To spy on a victim application, an adversary explicitly flushes a specified address in the shared memory region. Afterwards, she carefully measures the amount of time it takes to reload the data, so as to determine whether or not the address has been accessed by the victim in the meantime.

One cannot directly apply Flush+Reload techniques to SGX enclaves, since the `clflush` instruction requires read permissions on the provided memory location [114]. So it seems that properly implemented SGX enclaves do not share physical memory with their untrusted environment. We make the important observation, however, that an SGX enclave still *implicitly* shares unprotected page-table memory with the operating system. Since page-table entries are stored in regular DRAM, they are subject to the same caching mechanisms as any other memory location [114, 86] Additionally, modern Intel CPUs employ an internal paging-structure cache for page-table entries that reference other paging structures (but not those that map pages), and cache physical addresses in the TLB. As explained in Section 3.2.1, the processor's internal TLB and paging-structure caches are cleared whenever entering or exiting an enclave. However, since the data cache hierarchy remains explicitly untouched, an adversarial OS can perform a Flush+Reload-based cache timing attack on the page table itself.

In our `inc_secret` running example, a kernel-space attacker uses `clflush` to evict the last-level PTEs referenced by a as well as b, before entering the enclave. After the enclave has returned, she learns the secret input by carefully recording the amount of time it takes to reload the relevant PTEs. The latter can easily be achieved on x86 processors using the `rdtsc` instruction. We experimentally ascertained a timing penalty of at least 150 cycles for PTE entries that miss the cache, practically turning our Flush+Reload page-table attack into a reliable way to decide enclave page accesses.

Discussion. Cache timing attacks on page-table memory reveal a fundamental flaw in the SGX design. That is, walking the untrusted-page table during enclave execution discloses memory accesses at page-level granularity, even when faults would be suppressed and A/D bits are masked. However, as compared to the A/D channel, a cache-based attack suffers from a few limitations. First, one

```
point ec_mul (int d, point P) {
   point Q = 0; int n = nbits(d);
   for (int i = n-1; i >= 0; i--) {
       Q = point_double(Q);
       if (d & (0x1 << i))
           Q = point_add(Q, P);
   }
   return Q;
}
```

Listing 3.2: Elliptic curve scalar point multiplication.

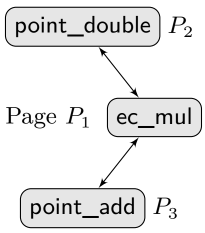


Figure 3.2: Code page layout for scalar point multiplication.

cannot distinguish between read and write accesses to the same page. This is not really a practical concern, however, since previous fault-based attacks [277, 230] do not rely specifically on write accesses. A second limitation considers the processor's prefetch unit [113, 88] that loads adjacent data speculatively into the cache. Specifically, during the reload phase of Flush+Reload, subsequent measurements might be destroyed. We develop a strategy to robustly infer page access patterns in the presence of false positives in Section 3.3.3.

A more severe limitation affects the granularity at which we can see page accesses. Since CPU caches exploit spatial locality, they fetch data from DRAM more than one byte at a time. The atomic unit of cache organization is called a *cache line* and measures 64 bytes on recent Intel processors [114]. A PTE entry on the other hand occupies only 8 bytes, implying that eight adjacent PTEs share the same cache line. PTE monitoring at a cache line granularity can thus conveniently be modelled as spying on enlarged ($8 * 4\,\text{KiB} = 32\,\text{KiB}$) pages.

3.3.2 Monitoring repeated accesses

So far, we only described how to detect memory page accesses after the enclave program has returned to its untrusted execution context. This suffices to extract secrets from the elementary code snippets in Listing 3.1. More realistic scenarios, however, repeatedly operate over the same code or data in a single start-to-end run.

As an example, consider the pseudocode for elliptic curve scalar point multiplication in Listing 3.2, where a provided point P is multiplied with a secret scalar d to obtain another point Q. The algorithm uses the double-and-add method, a variation of square-and-multiply used for modular exponentiation in RSA, and widely studied in side-channel analysis research [145, 44, 279, 280, 230]. We elaborate more on elliptic curve cryptography, and successfully attack Libgcrypt's implementation of the algorithm in Section 3.5.2. For now, we assume the `ec_mul` function is situated on code page P_1, whereas the subroutines `point_double` and `point_add` are located on distinct pages P_2 and P_3 (cf. Fig. 3.2). Previous fault-driven attacks [230] recovers the private scalar by observing different page fault sequences for iterations corresponding to a one $(P_1, P_2, P_1, P_3, P_1, P_2)$ or zero (P_1, P_2, P_1, P_2) bit.

The key difference in our stealthy attacker model, as compared to the page fault channel, is that we are *not* notified in case of a memory access. Instead, page-table entries should be explicitly monitored to establish whether they have been accessed or not. If the adversary only probes PTEs after enclave execution, she is left with aggregated information only (e.g., all pages P_1, P_2, and P_3 have been accessed). We therefore introduce a dedicated spy thread that monitors PTE entries in real-time, while the victim executes. The main challenge now becomes that SGX caches address translations in the TLB, implying that only the first access to a specific page results in a page-table walk. Subsequent accesses to the same page most likely hit the TLB, and will not be observed by a spy thread monitoring page-table memory. In the following, we present our approach to overcoming this challenge.

Flushing the TLB. We explicitly interrupt the enclaved victim application in order to reliably evict cached address translations without provoking page faults. Note that we don't even have to invalidate TLB entries explicitly, since an SGX-enabled processor automatically takes care of this during the AEX process. An adversary is left with two choices. She can either periodically interrupt the enclave with a timer-based preemption, or she can conditionally interrupt the victim CPU from a snooping thread. The timer-based approach would have to interrupt the victim enclave at a high frequency to minimize the risk of missing page accesses. Since SGX leaves enclaves interrupt-unaware by design, they have no way of detecting these frequent preemptions. Some of the enhanced TEE designs [48, 229] targeted by our stealthy attacker, however, redirect interrupts as well as page faults to a trusted enclave entry stub. Such fortified enclaves could recognize suspicious interrupt rates as an artifact of the attack, defeating our argument for stealthiness. We therefore opted for the second option that conditionally interrupts the victim CPU minimally. In this respect, note that concurrent research [263] has demonstrated that Simultaneous Multithreading (SMT) technology can be abused to evict TLB entries from a co-resident logical processor in real-time, *without* interrupting the victim enclave.

Our spy thread monitors one or more page-table entries in a tight loop, preempting the victim enclave CPU after a page access has been detected.

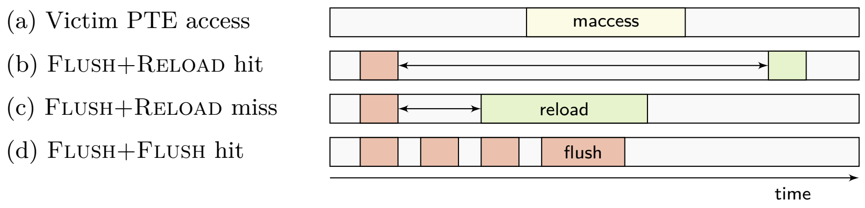


Figure 3.3: Flush+Flush as a high-resolution, low-latency channel to spy on victim PTE memory accesses.

The latter can be easily achieved in multiprocessor systems through a directed Inter Processor Interrupt (IPI), specifically designed to synchronize address translations across cores. From the point of view of the enclave, IPIs are directly handled by the CPU's local Advanced Programmable Interrupt Controller (APIC), and are thus indistinguishable from regular interrupts sent by a benign operating system.

Monitoring A/D bits. We experimentally confirmed that the "accessed" PTE attribute is only updated during the first page walk, since subsequent accesses hit the TLB. Furthermore, we found that the "dirty" attribute is independently set once for the first subsequent write access to that page. In the A/D implementation of our spy thread, an IPI is sent as soon as the A bit of the monitored PTE entry flips. Alternatively, an adversary can choose to only interrupt the victim enclave when the D flag changes. This might allow for a slightly stealthier attack, which interrupts the victim minimally, as pages are typically more often read than written to.

Monitoring PTE memory accesses. In a classical Flush+Reload attack [280], time is divided into slots. The spy program flushes the monitored cache lines at the start of each time slot, and reloads them at the end to find out whether they have been accessed by the concurrent victim program executing independently. When the victim's memory access overlaps with the flush or reload phases of the spy thread however, the measurement might be lost, as illustrated in Fig. 3.3c. Naturally, the probability of an overlapping victim access increases as the length of the time slot decreases, whereas a longer time slot increases detection latency and might miss subsequent memory accesses by the victim. As such, a trade-off is presented between attack resolution and accuracy.

When reloading PTEs after the enclave has been exited, as in the start-to-end examples of Listing 3.1, our measurement cannot be destroyed by a concurrent victim access. This is not the case, however, when monitoring page-table memory in real-time from a spy thread. Moreover, the victim only makes a single memory access to the monitored PTE entry, for subsequent accesses to the same page hit the TLB. In a classical Flush+Reload attack on the other hand, a missed memory access can be compensated for by subsequent accesses in the next time slot. We therefore chose to adopt a novel technique called Flush+Flush [87] that abuses microarchitectural timing differences in the execution time of the x86 `clflush` instruction, which depends on whether the data is cached or not. A spy thread that repeatedly flushes a specific PTE entry will observe a slightly higher execution time when the page has been accessed by the victim, as illustrated in Fig. 3.3d. Spying on page-table memory the Flush+Flush way thus ensures we can see all page accesses with a minimal detection latency.

Flush+Flush also confronts us with a new challenge however, since the microarchitectural timing differences of the `clflush` instruction are inherently more subtle than the apparent timing penalty for a DRAM access in Flush+Reload [87]. On the bright side, `clflush` does not trigger the processor's prefetcher, and therefore does not destroy subsequent measurements, a known concern for Flush+Reload [88]. We furthermore remark that, if needed, the spy thread can be made more robust by monitoring multiple code or data PTEs that each should be accessed before sending the IPI.

3.3.3 Inferring page access patterns

An essential ingredient of the attack procedure outlined so far, is that we interrupt the victim enclave via a targeted IPI from the spy thread. Some time passes however before the victim is interrupted, since the spy CPU cannot instantaneously detect PTE accesses and send the IPI. During this time interval, the victim enclave continues to execute instructions that may access additional code and data pages. Previous controlled-channel attacks on the contrary instantaneously trap to the OS in case of a page fault. This enables a PF-aware adversary to unambiguously distinguish two successive enclave instructions, whereas the accuracy at which we can see subsequent page accesses is constrained by IPI latency. In this respect, a fault-driven attack can be modelled as having zero latency between detecting a page access and interrupting the victim.

Page fault sequences. Naturally, page-table-based attacks have to deal with the limitation that they can only see memory accesses at a page-level granularity. Since functions as well as data objects typically share the same memory page with other functions or data objects, one cannot directly identify specific function or data accesses in a large enclave program. Xu et al. [277] overcome this challenge by identifying unique *page fault sequences* that lead to a particular code or data access. Since a PF-aware attacker does not have to cope with latency in the measurement process, she may construct page access sequences at instruction-level granularity.

In the running example of Listing 3.2, the `ec_mul` function on P_1 serves as a trampoline to redirect control flow to either `point_double` on page P_2 or `point_add` on page P_3, based on the secret scalar bit under consideration. A one bit can be identified by the sequence $(P_2, P_1, P_3, P_1, P_2)$. An observed page fault sequence of (P_2, P_1, P_2) on the other hand, corresponds to an iteration with a zero bit. One approach would be to implement a state machine in the spy thread to recognize such sequences. However, as the intermediate P_1 accesses are only a few instructions long, they could be easily missed by a stealthy spy that has to take IPI latency into account. Moreover, page fault sequences presuppose a completely noise-free way of establishing enclave page accesses. Recall from the above discussion, however, that FLUSH+RELOAD may suffer from occasional false positives by triggering the processor's prefetcher.

Page sets. To correlate subsequent page accesses in large enclave programs, we introduce the notion of *page sets* as a robust alternative to page fault sequences. Our spy thread continuously monitors one or more PTEs, from here on referred to as the *trigger page(s)*, and interrupts the victim enclave as soon as an access is detected. Upon IPI arrival, the spy establishes the set of pages (not) accessed by the victim, using one of the techniques from Section 3.3.1. Since the TLB is cleared whenever entering or exiting the enclave, these pages must have been accessed at least once by the victim from the previous interrupt up to now. We make the key observation that specific points in the execution trace of a large enclave program can be uniquely identified by matching the pattern of all pages accessed or not accessed in between two successive accesses to a trigger page. Note that information recovery via page sets is inherently stealthier than the previously proposed page fault sequences [277, 230] in that victim enclaves are only interrupted when accessing the trigger page. Where a page fault only leaks one bit of information (*i.e.*, the trigger page was accessed), our notion of page sets allows a spy to capture the maximum information for every trigger page interrupt.

Applying our page set theory to the running example of Listing 3.2, the spy thread monitors the trigger page P_2 holding `point_double`, and matches the page set $\{P_1, P_3\}$ on every interrupt. If both P_1 (`ec_mul`) and P_3 (`point_add`) have been accessed, the iteration corresponds to a one bit. Likewise, if P_1 has been accessed, but not P_3, the iteration processed a zero bit. Finally, in case

P_1 as well as P_3 were both not accessed, P_2 must have been accessed from an execution context other than the targeted `point_double` invocation, and we classify the interrupt as a false positive.

After identifying secret-dependent control flow or data accesses in the victim application, a successful attack comes down to designating specific pages to be tracked in the spy thread, and recognizing the associated page set patterns. Analogous to previous fault-based attacks [277, 230], we first perform a detailed offline analysis of the enclaved application binary to extract an ideal trace of instruction-granular page accesses for a known input. From this ideal trace, we select a suitable candidate trigger page, and we construct the sets of all pages accessed or not accessed in between two hits on the trigger page. By comparing the resulting page sets, we are left with a page set pattern that (uniquely and robustly) identifies a specific point in the victim's execution trace.

3.4 Implementation

Similar to previous controlled-channel attacks [277, 230], our exploits target unmodified legacy applications running under the protection of a TEE. The enclaved application binary is protected from the untrusted host operating system by means of a *shielding system* that provides trusted library services, and interposes on system calls. Previous controlled-channel attacks on Intel SGX were implemented for the Haven [20] shielding system. Since Haven is not publicly available, we implemented our attacks on the open-source[1] Graphene-SGX library OS [246]. We first briefly overview the internals of Graphene-SGX, and thereafter explain how we extended the untrusted runtime with a reusable attacker framework.

Graphene-SGX. Library OSs such as Graphene [246] repackage conventional OS kernel services into a user-mode application library. System calls made by the legacy application are transparently transformed into libOS function calls, which are then either processed locally, or translated into a minimal host kernel ABI that provides core OS primitives. The libOS relies on a small Platform Adaptation Layer (PAL) to translate platform-independent host ABI calls into a narrow set of system calls to the underlying host operating system, which remains, however, explicitly trusted from a security perspective.

Graphene-SGX [246]—like other recently proposed SGX-based shielding systems including Haven [20], Panoply [231], and SCONE [17]—improves over this

[1]`https://github.com/oscarlab/graphene`

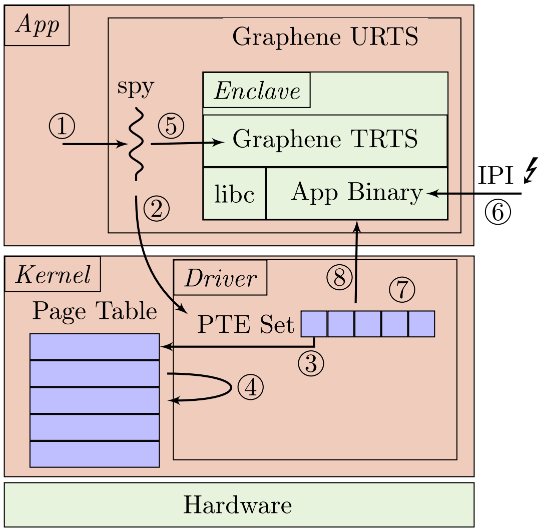


Figure 3.4: Graphene-SGX attack framework interaction.

situation by not only protecting libOS instances from each other, but also from a malicious host operating system. To this end, Graphene-SGX encapsulates the entire libOS, including the unmodified application binary and supporting libraries, inside an SGX enclave. Graphene also inserts a trusted runtime with a customized C library and ELF loader in the enclave. Since SGX prohibits enclaves from making system calls directly, the PAL is split into a trusted part that calls out to an untrusted runtime in the containing application to perform the system call to the untrusted host OS. Graphene-SGX furthermore relies on an untrusted Linux driver for enclave creation/tear down and protected memory management via the dedicated ring-zero SGX instruction set.

Attack framework. We implemented our attacks as an extension to Graphene's untrusted runtime, leaving the trusted in-enclave components unchanged. Our implementation is conceived as a reusable framework to facilitate eavesdropping on different application binaries.

Figure 3.4 summarizes the steps undertaken by our attack framework. ① The untrusted user space runtime creates a separate spy thread just before entering the enclave's main function. We affinitize the spy and victim threads to their own physical CPU cores to avoid any noise from page-table shootdowns by the OS scheduler. ② The newly created spy thread continues its execution in kernel space by calling to our modified Graphene-SGX driver. We run our core attacker code in kernel mode to be able to easily send IPIs, inspect PTE attributes, and monitor page-table memory. ③ The spy first goes through a pluggable, attack-specific initialization phase that creates the page sets to be monitored. ④ After synchronizing with the victim thread, which is still waiting to enter the enclave, the spy enters a tight probing loop that measures either `clflush` execution time, or A/D attributes of one or more page-table entries. ⑤ Victim thread enters the enclave. ⑥ Upon detecting an access on the trigger page, the spy interrupts the victim thread as soon as possible. ⑦ The IPI handler on the victim CPU now establishes the access pattern for the monitored page set using either the noise-free FLUSH+RELOAD or A/D mechanism. Page set access patterns are logged for later parsing by an attack-specific post-processing script. ⑧ Spy and victim threads synchronize once more before resuming the enclave.

So far, we assumed the attacker obtained the page addresses to be monitored from an `objdump` of the application binary. Graphene, like other SGX-based shielding systems [20, 231, 17], does not randomize the base address of loaded executables. Instead, applications and supporting libraries (including libc) are loaded at deterministic memory locations. To easily discover executable base addresses, we propose to first deploy the target application binary in an attacker-controlled libOS instance that we minimally modified to leak load addresses. SGX's remote attestation scheme properly prevents us from deploying the modified libOS instance when running the application for the remote stakeholder, but the observed load addresses will be identical. Note that it has been shown [277] that hypothetical support for conventional address space layout randomization, which only randomizes the application's base address, could be easily defeated by observing page access patterns.

Inter-processor interrupts. In a page fault-driven attack, the victim enclave is exited immediately when accessing a monitored page. For our PF-oblivious attacks on the contrary, we define IPI latency as the number of instructions executed by the victim enclave after accessing a trigger page, and before being interrupted by the spy thread. Reducing IPI latency is an important implementation consideration in that it defines the accuracy at which we can see subsequent page accesses. Before quantifying latency in the evaluation section, we present some general implementation techniques to minimize IPI latency.

Our driver hooks into an unused IPI vector of Linux's KVM subsystem by registering the address of our interrupt handler in the system-wide IDT. This allows us to send the IPI promptly from assembly code in the spy thread by writing to the relevant memory-mapped APIC address, instead of having to rely on Linux's IPI subsystem that performs bookkeeping on shared data structures before sending the interrupt. To further reduce IPI latency, we considered a previously proposed [156] technique that sets the "cache disable" bit in the `CR0` control register to disable the L1, L2, and L3 cache on the CPU running the victim enclave. We experimentally confirmed that this technique dramatically slows down the victim thread, and substantially reduces the number of instructions executed after accessing a trigger page. However, setting `CR0.CD` on the victim CPU invalidates our cache-based PTE timing attack vector. Moreover, the aforementioned T-SGX defense [229] would be able to detect this technique, for TSX relies on the CPU cache to start transactions [114].

Analyzing page sets. With our attack framework in place, the main challenge left is to select the pages that need to be tracked in the spy thread. To study the behavior of target applications, previous controlled-channel attacks [277] record a complete, byte-granular trace of page fault addresses by running the application outside of the enclave with at most one code and data page allocated at all times. We simplify this process via a GNU debugger script that extracts an instruction-granular code page trace by single-stepping through the *unprotected* application binary, recording the symbolic name and virtual page address of the instruction pointer. Furthermore, by placing strategic breakpoints, the debugger script can easily be instrumented to mark individual loop iterations.

To construct the most stealthy attack, we select a trigger page that is minimally accessed in the extracted trace, and we compose a set of remaining pages that unambiguously identifies the code page access of interest. When running the attack on an enclaved application binary, our driver dumps page set patterns for all accesses on the trigger page. Afterwards, we use a small, attack-specific post-processing script to match the desired patterns in the driver output. If needed, the pattern to be matched, can also include the page sets of previous or succeeding trigger page accesses, and can be made more robust by means of a regular expression.

3.5 Evaluation

In this section, we evaluate our attack framework. We first provide microbenchmarks to quantify IPI latency, and thereafter demonstrate the effectiveness of our attacks by extracting EdDSA session keys from an unmodified binary of the widely used Libgcrypt cryptographic library.

All experiments were conducted on publicly available off-the-shelf SGX hardware. We used a commodity Dell Inspiron 13 7359 laptop with a Skylake dual-core Intel i7-6500U processor and 8 GiB of RAM. The machine runs Ubuntu 15.10, with a generic 64-bit Linux 4.2.0 kernel. To prevent any noise from OS scheduling

Table 3.1: IPI latency in terms of the number of instructions executed by the victim after accessing the trigger page.

	Accessed		Flush+Flush		
Experiment	**Mean**	σ	**Mean**	σ	**Zero %**
`nop`	431.70	34.11	0.65	17.65	99.84
`add` register	176.30	14.60	0.15	6.18	99.94
`add` memory	32.45	2.79	0.06	1.92	99.88
`nop` nocache	0.02	0.39	–	–	–

decisions, we disabled SMT and reserved a dedicated CPU for the spy thread using Linux's `isolcpus` boot option. We based our attack framework on a recent `master` checkout of the Graphene project, compiled with `gcc` v5.2.1.

3.5.1 IPI latency microbenchmarks

Recall from Section 3.4 that we want to minimize the number of instructions executed by the victim enclave after accessing a trigger page, and before being interrupted by a targeted IPI from the spy thread. In order to reliably quantify IPI latency, we wrote a small microbenchmark application that first accesses an isolated memory page, and immediately thereafter starts executing an instruction slide of 5,000 identical x86 instructions. For the microbenchmark experiments, we instrumented our driver to retrieve the instruction pointer stored in the SSA frame of the interrupted debug enclave through the `edbgrd` SGX instruction. The exact number of instructions executed in the microbenchmark application can be inferred by comparing the retrieved instruction pointer with the known start address of the instruction slide.

Interrupt granularity. Table 3.1 records IPI latencies for different x86 instructions. We repeat all experiments 10,000 times for a spy thread that monitors the trigger page through the "accessed" PTE attribute, as well as for a spy that repeatedly flushes page-table memory locations. We present the mean and the standard deviation (σ) to characterize IPI latency distributions. In the first experiment, we prepare an instruction slide with ordinary no-operations. The upper row of Table 3.1 reveals a first important result. That is, our benchmark enclave can only be interrupted by an A/D spy at a relatively coarse-grained granularity of about 430 `nop`s, whereas the novel Flush+Flush technique immediately interrupts the victim thread. Note that interrupts with zero IPI latency arrive *within* the instruction that accessed the trigger page, even before the next enclave instruction started executing. The last column, which lists the percentage of interrupts with zero IPI latency, distinctly shows that a victim thread monitored by a FLUSH+FLUSH spy is interrupted within the trigger instruction with very high probability (99.84%). As such, FLUSH+FLUSH represents a *precise*, instruction-granular, technique to interrupt victim enclaves, improving significantly over related state-of-the-art enclave execution control proposals [266, 156, 181]. We furthermore found the technique to be *reliable*, for FLUSH+FLUSH recorded all 10,000 page accesses, without false positives, and with significantly less noise (smaller standard deviation) than an A/D spy.

The increased advantage of a FLUSH+FLUSH spy, as opposed to a spy monitoring A/D bits, can be understood from the effects on the caching behavior of the page-table walk. A PTE memory location that is continuously probed by an A/D spy will be cached when the victim CPU performs the page-table walk, whereas a FLUSH+FLUSH spy actively ensures the victim CPU misses the cache. As such, instructions that access the trigger page will take longer to complete, providing a wider time frame for IPI arrival. This effect is further aggravated when the processor needs to update the "accessed" page-table attribute. For the victim CPU needs to perform another memory access to reload the PTE entry from DRAM when the A bit was not set, and the corresponding cache line has been flushed by a concurrent spy thread. Interestingly, we found that the victim's second PTE memory access, where the A bit is updated, is more noticeable from a FLUSH+FLUSH spy thread. Intel's software optimization manual [113] indeed confirms that "flushing cache lines in modified state are more costly than flushing cache lines in non-modified states".

Instruction latency. The second and third experiments investigate the influence of the microbenchmark instruction type on IPI latency. We start from the intuition that an individual `nop` instruction is trivial to execute and can easily be pipelined, allowing many instructions to be executed in the limited time period after accessing the trigger page and before IPI arrival. The second row of Table 3.1 confirms that a victim program can make significantly less progress on an instruction slide with `add` instructions that sequentially increment a processor register. Likewise, the third row shows that IPI latency drops even further when the victim executes a sequence of `add` instructions that increment a memory location. The latter can be explained from the additional page-table walk that retrieves the physical memory address of the data operand for the first `add` instruction.

Finally, we performed an experiment that entirely disables instruction and data caching on the victim CPU by setting the `CR0.CD` bit, as explained in Section 3.4. The last row of Table 3.1 clearly shows that this approach can almost completely eliminate IPI latency (mean and standard deviation near zero) for an A/D spy. This confirms our hypothesis that the observed IPI latency differences stem from the caching behavior of the page-table walk. Of course, a FLUSH+FLUSH spy cannot see page accesses when the cache is disabled on the victim CPU.

3.5.2 Attacking Libgcrypt EdDSA

To illustrate the applicability of our attacks on real-world applications, we extract private EdDSA session keys from a general purpose cryptographic library Libgcrypt, which used in the popular GnuPG cryptographic software suite. More specifically, we reproduce a previously published [230] page fault-driven attack on Libgcrypt, showing that our stealthy attack vectors can extract the same information without triggering any page faults. Since Libgcrypt is officially distributed from source code, we built unmodified binaries for Libgcrypt v1.6.3 and v1.7.5 as well as the accompanying error-reporting library Libgpg-error v1.26 through the default `./configure && make` invocation, using `gcc` v5.2.1.

EdDSA implementation. The Edwards-curve Digital Signature Algorithm (EdDSA) [22] is an efficient, high-security signature scheme over a twisted Edwards elliptic curve with public reference point G. The security of elliptic curve public key crypto systems critically relies on the computational intractability of the elliptic curve discrete logarithm problem: given an elliptic curve with two points A and B, find a scalar k such that $A = kB$. Recall that our running example in Listing 3.2 provides an efficient algorithm for the inverse operation, *i.e.*, multiply a point with a known scalar. EdDSA uses scalar point multiplication for public key generation, as well as in the signing operation. The private key d is derived from a randomly chosen large scalar value, and the corresponding public key is calculated as $Q = dG$. To sign a message M, EdDSA first generates a secret *session key* r, also referred to as *nonce*, by hashing the long-term private key d together with M. Next, the signature is calculated as the tuple $(R = rG, S = r + hash(R, Q, M)d)$. It can be seen that an adversary who learns the secret session key r from side-channel observations during the signing process, can easily recover the long-term private key as $d = (S - r)/hash(R, Q, M)$, with (R, S) a valid signature for a known message M [22, 279].

Listing 3.3 provides the relevant section of the scalar point multiplication routine in Libgcrypt v1.6.3. Lines 14 to 18 are a straightforward implementation of Listing 3.2, and have previously been successfully targeted in a page fault-aware attacker model [230]. We remark however that Libgcrypt provides some

```
 1 if (mpi_is_secure (scalar)) {
 2     /* If SCALAR is in secure memory we assume that it is the
 3        secret key we use constant time operation.  */
 4     point_init (&tmppnt);
 5
 6     for (j=nbits-1; j >= 0; j--) {
 7         _gcry_mpi_ec_dup_point (result, result, ctx);
 8         _gcry_mpi_ec_add_points (&tmppnt, result, point, ctx);
 9         if (mpi_test_bit (scalar, j)) /* ← eliminated in v1.7.5 */
10             point_set (result, &tmppnt);
11     }
12     point_free (&tmppnt);
13 } else {
14     for (j=nbits-1; j >= 0; j--) {
15         _gcry_mpi_ec_dup_point (result, result, ctx);
16         if (mpi_test_bit (scalar, j))
17            _gcry_mpi_ec_add_points (result, result, point, ctx);
18     }
19 }
```

Listing 3.3: Scalar point multiplication in Libgcrypt v1.6.3.

protection against side-channel attacks by tagging sensitive data, including the EdDSA long-term private key, as "secure memory" [143]. Lines 1 to 12 show how a hardened, add-always scalar point multiplication algorithm is applied when the provided scalar is tagged as secure memory. However, while the hardened algorithm of Libgcrypt v1.6.3 greatly reduces the attack surface by cutting down the amount of secret-dependent code, we show that even the short if branch on line 9 remains vulnerable to page-table side-channel attacks during the public key generation phase. We verified that this defect has been addressed in the latest version v1.7.5 by replacing the if branch with a truly constant time swap operation. We also found, however, that Libgcrypt v1.6.3 as well as v1.7.5 do *not* tag the secret EdDSA session key as secure memory, resulting in the non-hardened path being taken during the signing phase.[2]

Monitoring A/D bits. We first explain how we attacked the hardened multiplication (lines 6 to 11) in Libgcrypt v1.6.3. We found that every loop iteration accesses 21 distinct code pages, regardless of whether a one or a zero bit was processed. Our stealthy spy thread monitors the A attribute of the trigger page-table entry holding the physical page address of `point_set`, which is accessed 126 or 127 times each iteration, depending on the scalar bit under consideration. We rely on a robust PTE set of nine additional code pages whose

[2] To address this shortcoming, we contributed a patch that has been merged in Libgcrypt v1.7.7.

combined A bits unambiguously identify an unconditional execution point in `add_points` as well as the conditional `point_set` invocation on line 10. Our post-processing script reliably recovers the full 512-bit EdDSA session key by counting the number of IPIs (*i.e.*, trigger page accesses) in between two page set pattern hits. PTE set hits are classified as belonging to a different iteration when the number of IPIs in between them exceeds a certain threshold value. As such, iterations that processed a one bit are easily recognized by two page set hits, whereas zero iterations hit only once. Our A/D attack on Libgcrypt v1.6.3 interrupts the victim enclave about 60,000 times.

To attack the standard multiplication (lines 14 to 18) in the latest Libgcrypt v1.7.5, we spy on the A attribute of the PTE that references the `test_bit` code page. Our offline analysis shows that the trigger page is accessed 93 or 237 times for iterations that respectively process a zero or a one bit. The spy thread records a PTE set of four additional code pages whose combined access patterns uniquely identify the if branch on line 16. We reliably recover all 512 secret scalar bits at post-processing time by observing that the PTE set pattern repeats exactly once every loop iteration, and the page set value for the first subsequent trigger page access depends on whether the if branch was taken or not. We counted only about 40,000 IPIs for our A/D attack on Libgcrypt v1.7.5.

Monitoring cache misses. Recall from Section 3.3 that spying on page-table memory at a cache line granularity is challenging in that we can only see accesses for conceptually enlarged 32 KiB pages. Our offline analysis on Libgcrypt v1.7.5 shows that every loop iteration accesses 22 code pages, belonging to three different application libraries: Libgcrypt, Libgpg-error, and the trusted libc included by Graphene. Only 11 of these 22 code pages fall in distinct cache lines. Interestingly, we found that the `free` wrapper function used by Libgcrypt stores/restores the `errno` memory location of the trusted in-enclave libc 46 or 102 times for zero respectively one iterations. The address of the error number for the current thread can be retrieved via the `__errno_location` function, residing at a remote location within the libc memory layout.

Our stealthy FLUSH+FLUSH spy uses the code page for the `__errno_location` libc function as a reliable trigger page that does not share a cache line with any of the other pages accessed in the loop. Our cache-based attack on Libgcrypt interrupts the victim enclave about 130,000 times for a single, start-to-end run. We furthermore construct a page set covering 7 distinct PTE cache lines that are recorded by the spy on every trigger page access, using the FLUSH+RELOAD technique after interrupting the enclave. While the extracted page set value sequences themselves appear quite noisy at first sight, we found that certain values unmistakably repeat more often in iterations that processed a one bit. Furthermore, the number of IPIs (*i.e.*, `errno` accesses) in between these values exhibit clear repetitions. Our post-processing script uses a regular expression to identify a robust pattern that repeats once every iteration. Again, key bits can be inferred straightforwardly from the number of IPIs in between pattern hits. Using this technique, we were able to correctly recover 485 bits of a 512-bit secret EdDSA session key in a single run of the victim enclave. Moreover, using the number of IPIs in between two recovered scalar bits as a heuristic measure, our post-processing script is able to give an indication of which bit positions are missing.

3.6 Discussion and mitigations

Frequent enclave preemption. Our work shows that enclave memory accesses can be learned by spying on unprotected page tables, without triggering any page faults. This observation is paramount for the development of defenses against page-table-based threats. Specifically, state-of-the-art PF-oblivious defenses [230, 229] do not achieve the required guarantees. We only interrupt the enclave when successive accesses to the same page need to be monitored. Importantly, our attacks remain undetected by T-SGX [229], since it allows up to 10 consecutive transaction aborts (interrupts) for each individual basic block. We do acknowledge, however, that the number of interrupts reported for our Libgcrypt attacks in Section 3.5.2 is substantially higher than what is to be expected under benign circumstances. We can therefore see improved, heuristic defenses using suspicious interrupt rates as an artifact of an ongoing attack.

Indeed, Déjà Vu [42], which was first published after we submitted this work, explores the use of TSX to construct an in-enclave reference clock thread that cannot be silently stopped by the OS. The enclave program is instrumented to time its own activity, so as to detect the execution slowdown associated with an unusual high number of AEXs. While Déjà Vu would likely recognize frequent enclave preemptions as a side effect of our current attack framework, we argue that heuristic defenses do not address the *root* causes of page-table-based information leakage. That is, our novel attack vectors are still applicable, and depending on the victim program, interrupts may not even be required. The knowledge that a specific page is accessed, can reveal security-sensitive information directly, or enable an attacker to launch a second phase of her attack [266]. Furthermore, as part of the continuous attacker-defender race, we expect the contributed attack vectors to trigger improved, stealthier attacks that remain under the radar of Déjà Vu-like defenses.

In this regard, Wang et al. [263] concurrently developed similar page-table-based attacks. In contrast to our work, they explore a perpendicular approach which does not aim to maximize the temporal resolution, but instead only focuses on the A/D channel, rather than PTE caching, and minimizes enclave interrupts by exploiting SMT-based contention from a concurrent, attacker-controlled sibling core to evict TLB entries *without* interrupting the victim enclave. As such, this observation effectively demonstrates that Déjà Vu-like defenses are inherently insufficient to eliminate page-table-based threats.

Hiding enclave page accesses. At the system level, some lightweight embedded TEEs [189, 147] avoid page-table-based threats altogether by implementing hardware-enforced isolation in a single-address-space. Alternatively, some higher-end TEE research prototypes [48, 58, 173, 236] place enclave page tables out of reach of an attacker. Unfortunately, we believe such an approach is unacceptable for Intel SGX, especially when protecting sensitive application data from potentially malicious cloud providers [20, 217]. In such use cases, the cloud provider must be able to quickly regulate different cloud users competing for scarce platform resources including EPC memory. Fortified TEE designs such as Sanctum [48] on the other hand move page tables within the enclave, and require the OS to engage in a lengthy protocol whenever reclaiming a physical page. Furthermore, when applying Sanctum's enclave-private page table design to modern x86 processors [114], an adversary could still leverage the Extended Page Tables (EPTs) set up by the hypervisor. That is, any access to guest-physical pages, including the enclave and its private page tables, results in an EPT walk that sets accessed and dirty bits accordingly. Masking A/D attributes in enclave mode is neither sufficient nor desirable, as it cannot prevent our cache-based attacks, and disrupts benign OS memory management decisions.

At the application level, we believe the academic community should investigate different defense strategies based on the type of enclave. For small enclaves that must be offered the highest security guarantees, automated compiler-based solutions [44] are to be considered. Good practices applied to cryptographic software (e.g., not branching on a secret) may be extended to more general approaches, such as the deterministic multiplexing defense proposed by Shinde et al. [230]. For uses cases where unmodified application binaries are loaded in an enclave, however, such approaches would likely lead to unacceptable performance overhead. In such situations, the use of more probabilistic security measures may be acceptable. Note that previous page fault-driven research [277] successfully defeated conventional Address Space Layout Randomization (ASLR) schemes that randomize an application's base address. SGX-Shield [227], on the other hand, implements fine-grained ASLR by compiling enclaved application code into small 32- or 64-byte randomization units that can subsequently be re-shuffled at load time.

3.7 Related work

A recent line of work has developed TEE security architectures that support secure isolated execution of enclaves with a minimal trusted computing base, either via a small hypervisor [174, 173, 236, 100], or with trusted hardware [166, 176, 58, 48, 189, 147]. Intel SGX represents the first widespread TEE solution, included in off-the-shelf consumer hardware, and has recently been put forward to protect sensitive application data from untrusted cloud providers [20, 217]. As such, SGX has received considerable attention from the research community, and one line of work, including Graphene-SGX [246], Haven [20], Panoply [231], and SCONE [17] has developed small libOSs that facilitate running unmodified legacy applications in SGX enclaves. However, Xu et al. [277] recently pointed out that enclaved execution environments are vulnerable to a new class of powerful controlled-channel attacks conducted by an untrusted host operating system. We have discussed previous research results on page-table-based attacks and defenses extensively in Section 3.2.3. Iago attacks [38] furthermore exploit legacy applications via the system call interface, and AsyncShock [266] demonstrates that an adversarial OS can more easily exploit thread synchronization bugs within SGX enclaves. Finally, the SGX research community has witnessed a steady stream of microarchitectural side-channel attacks; either by abusing the branch prediction unit [156], or in the form of fine-grained PRIME+PROBE [79, 225, 29, 181] cache attacks.

In a more general, non-TEE context, there exists a vast amount of research on microarchitectural cache timing vulnerabilities [202, 280, 88]. Especially relevant to our work is the FLUSH+FLUSH [87] channel which was only proposed very recently, and attack research [279] that applies FLUSH+RELOAD to partially recover OpenSSL ECDSA nonces. Furthermore, timing differences from TLB misses have been exploited to break kernel space ASLR [104]. More recently, it has been shown that kernel ASLR can also be bypassed by exploiting timing differences in the `prefetch` instruction [86], or by leveraging TSX [132]. Finally, recent concurrent work [81] on JavaScript environments has independently demonstrated a page-table-based cache side-channel attack that completely compromises application-level ASLR.

3.8 Conclusion

Our work shows that page-table walks in unprotected memory leak enclave page accesses to untrusted system software. We demonstrated that our stealthy attack vectors can circumvent current state-of-the-art defenses that hide page faults from the OS. As such, page-table-based threats continue to be worrisome for enclaved execution.

Chapter 4

SGX-Step: A practical attack framework for precise enclave execution control

This chapter was previously published as:

> J. Van Bulck, F. Piessens, and R. Strackx. “SGX-Step: A Practical Attack Framework for Precise Enclave Execution Control”. In: *2nd Workshop on System Software for Trusted Execution (SysTEX)*. ACM, Oct. 2017, 4:1–4:6

Preamble

This chapter presents SGX-Step, an open-source framework to facilitate side-channel attack research on Intel x86 processors in general and Intel SGX platforms in particular. SGX-Step consists of two components, the first one being an adversarial Linux kernel driver that exports traditionally privileged operating system powers to user space. The second component is a small user-space attack library that allows to configure page-table entries and x86 APIC timer interrupts directly from the untrusted enclave host process. We contribute and evaluate an improved approach to single-step enclaved execution at instruction-level granularity, and we show how SGX-Step enables several new or improved attacks. Finally, we discuss its implications for the design of effective defense mechanisms.

This research won the best paper award at SysTEX 2017. Since the original release of our open-source framework, SGX-Step gained widespread recognition in the TEE community and has been leveraged in our own research [91, 249, 256, 223, 35, 254, 188, 182, 251], as well as by several independent researchers [270, 6, 105, 130, 269, 8, 208, 211, 5, 94], to enable a long line of new and improved high-resolution enclave attacks. From a defensive perspective, SGX-Step permanently refined the TEE threat model by showing that enclaves can be precisely interrupted exactly one instruction at a time. This observation defeats any side-channel mitigations [156, 69, 119] that are based on partial atomic behavior of the instruction stream and has furthermore informed several recent defensive works [103, 9, 34, 110]. which now properly take single-stepping adversary capabilities into account. For instance, our single-stepping attack against the Zigzagger [156] branch-shadowing mitigation has directly inspired an improved compile-time hardening technique based on randomization [103]. Likewise, Intel explicitly mentions SGX-Step in the security analysis deep dive of their recently released LVI mitigations [110].

SGX-Step originates from initial experimental efforts to avoid compilation overheads when creating a custom Linux kernel for measuring interrupt latency in early prototypes of Nemesis (Chapter 5) and to avoid kernel panics when running varying page-table spy code in kernel mode (Chapter 3). In this respect, SGX-Step significantly eases enclave attack prototyping by offering a mini user-space operating system library on top of an unmodified stock Linux kernel. This is in notable contrast to alternative attacks that require developing a custom kernel driver module [277, 258, 181, 92, 39] or even patching and re-compiling the entire Linux kernel [156, 95, 233].

Over the past three years, we have maintained SGX-Step as an active open-source project on GitHub, expanding the framework with new features and keeping it up-to-date with recent Linux kernel and SGX SDK versions. Table 4.1 lists the principal x86 features supported by modern SGX-Step distributions, together with an overview of demonstrated interrupt-driven SGX attacks. Note that we defer a more systematic overview of the SGX attack landscape to Chapter 8. Table 4.1 only focuses on the relevant subset of attacks, namely the ones that leverage interrupts or page faults to frequently preempt a victim enclave. The highlighted works leverage SGX-Step, and the table clearly shows increasing adoption of the framework over time in a line of high-resolution attacks. Indeed, to date SGX-Step remains the first and only framework of its kind to offer true, noiseless single-stepping capabilities. The table furthermore reveals that, apart from single-stepping, a considerable number of attacks have benefited from SGX-Step's convenient page-table manipulation interface to preempt victim enclaves at a coarser-grained, page-level granularity. As a last interesting tendency, some recent works [188, 223, 35, 281] have adopted SGX-

Table 4.1: Overview of demonstrated interrupt-driven SGX attacks in terms of the achieved temporal resolution (number of instructions) and x86 features used: APIC device (timer interrupts, inter-processor interrupts), page-table manipulations (page faults, accessed/dirty bits, physical page number), global- and interrupt-descriptor-table manipulations. The last column indicates whether the attack was mounted after dynamically loading a driver vs. patching and re-compiling a custom OS kernel. Attacks using the user-space SGX-Step framework are in the highlighted rows.

			APIC		**PTE**			**Desc**		
Yr	**Attack**	**Temporal resolution**	IRQ	IPI	#PF	A/D	PPN	GDT	IDT	**Drv**
'15	Ctrl channel [277]	~ Page	○	○	●	○	○	○	●	✓ ⊞
'16	AsyncShock [266]	~ Page	○	○	●	○	○	○	○	– 🐧
'17	CacheZoom [181]	✗ >1	●	○	○	○	○	○	○	✓ 🐧
'17	Hahnel et al. [92]	✗ 0 - >1	●	○	○	○	○	○	●	✓ ⊞
'17	BranchShadow [156]	✗ 5 - 50	●	○	○	○	○	○	○	✗ 🐧
'17	Stealthy PTE [258]	~ Page	○	●	○	●	○	○	●	✓ 🐧
'17	DarkROP [154]	~ Page	○	○	●	○	○	○	○	✓ 🐧
'17	SGX-Step [257]	✓ 0 - 1	●	○	●	●	○	○	○	✓ 🐎
'18	Off-limits [91]	✓ 0 - 1	●	○	●	○	○	●	○	✓ 🐎
'18	Single-trace RSA [270]	~ Page	○	○	●	○	○	○	○	✓ 🐎
'18	Foreshadow [249]	✓ 0 - 1	●	○	●	○	●	○	○	✓ 🐎
'18	SgxPectre [39]	~ Page	○	○	●	○	○	○	○	✓ 🐧
'18	CacheQuote [50]	✗ >1	●	○	○	○	○	○	○	✓ 🐧
'18	SGXlinger [95]	✗ >1	●	○	○	○	○	○	○	✗ 🐧
'18	Nemesis [256]	✓ 1	●	○	●	●	○	○	●	✓ 🐎
'19	Spoiler [130]	✓ 1	●	○	○	●	○	○	●	✓ 🐎
'19	ZombieLoad [223]	✓ 0 - 1	●	○	●	●	○	○	●	✓ 🐎
'19	Tale of 2 worlds [254]	✓ 1	●	○	●	●	○	○	●	✓ 🐎
'19	MicroScope [233]	~ 0 - Page	○	○	●	○	○	○	○	✗ 🐧
'20	Bluethunder [105]	✓ 1	●	○	○	○	○	○	●	✓ 🐎
'20	Big troubles [269]	~ Page	○	○	●	○	○	○	○	✓ 🐎
'20	Viral primitive [6]	✓ 1	●	○	●	●	○	○	●	✓ 🐎
'20	CopyCat [182]	✓ 1	●	○	●	●	○	○	●	✓ 🐎
'20	LVI [251]	✓ 1	●	○	●	●	●	○	●	✓ 🐎
'20	A to Z [8]	~ Page	○	○	●	○	○	○	○	✓ 🐎
'20	Frontal [208]	✓ 1	●	○	●	●	○	○	●	✓ 🐎
'20	CrossTalk [211]	✓ 1	●	○	●	○	○	○	●	✓ 🐎
'20	Online template [5]	~ Page	○	○	●	○	○	○	○	✓ 🐎
'20	Déjà Vu NSS [94]	~ Page	○	○	●	○	○	○	○	✓ 🐎

Step for rapid x86 attack prototyping without necessarily relying on frequent enclave preemptions and ultimately even without targeting SGX enclaves at all.

Importantly, while the single-stepping technique provided by SGX-Step considerably amplifies existing leakage sources, it does in itself not directly break enclave security. By confronting the ultimate consequences of a privileged adversary threat model, however, we consider the primary role of SGX-Step to be accordingly raising the bar for adequate defenses in an enclave setting. In this respect, SGX-Step reveals the inevitable tension between processor-level enclave isolation and the operating system's traditional role as a resource manager. Hence, silver-bullet defenses should not be expected, but we outline a number of potential hardening techniques, which may hinder exploitation with SGX-Step, and their trade-offs in Chapter 8.

4.1 Introduction

Today's computing platforms rely on privileged system software to separate applications, and to govern the interactions between them. Commodity monolithic Operating System (OS) kernels, however, consist of millions of lines of code written in unsafe languages, exposed to both logical bugs and low-level software vulnerabilities. In response to these concerns, the past years have seen a significant research effort [166, 114, 48] on Trusted Execution Environments (TEEs) that support isolated execution of security-sensitive application components or *enclaves* with a minimal Trusted Computing Base (TCB). These proposals have in common that they enforce security primitives directly in hardware, or in a small hypervisor, so as to prevent the untrusted OS from accessing enclaved code or data directly, while still leaving it in charge of shared platform resources such as system memory or CPU time. With the arrival of Intel's Software Guard Extensions (SGX) [114, 119], such strong hardware-enforced trusted computing guarantees are now available on mainstream consumer devices.

Recent research demonstrated, however, that the increased capabilities of a privileged TEE attacker allow her to construct high-resolution, low-noise channels to spy on enclaved execution. Specifically, the past months have seen a steady stream of kernel-level SGX attacks exploiting information leakage from page tables [277, 258], CPU caches [181, 92], or branch prediction units [156]. These attacks commonly exploit the OS's control over timer devices to gain fine-grained side-channel observations from frequent enclave preemptions. As such, the precision at which one can interrupt a victim enclave, determines the temporal resolution of the attack.

This chapter shows that enclaved execution can be reliably monitored at a *maximal* temporal resolution (*i.e.*, instruction per instruction). Specifically, we present and evaluate SGX-Step, which is the first framework of its kind to achieve true single-stepping for arbitrary enclave programs. We furthermore lower the bar for enclave preemption attacks considerably by exporting user space memory mappings for the local APIC timer device and enclave page tables. As part of our evaluation, we defeat a recently proposed branch prediction defense [156], demonstrating SGX-Step's enhanced precision over previous proposals. Summarized, we make the following contributions:

- We show that enclaved execution can be precisely single-stepped using a novel APIC timer manipulation.
- We implement SGX-Step as an open-source[1] Linux kernel driver and runtime library, and explain how it improves the temporal resolution of existing attacks.
- We evaluate our approach on two different SGX processors, and provide evidence that SGX-Step enables new attacks that were previously deemed infeasible.

4.2 Background and related work

4.2.1 Attacker model and Intel SGX

Ongoing concerns on protecting sensitive data from software running at higher privilege levels have led to the Software Guard Extensions (SGX) [114, 119] included in recent Intel x86 processors. SGX enables hardware-enforced isolation and attestation of security-critical code in *enclaves*, embedded in the virtual address space of a conventional OS process. Legacy page tables are left under explicit control of the untrusted OS, but the processor's Memory Management Unit (MMU) enforces that enclave-private memory can never be directly accessed from outside. Hardware-level cryptography furthermore allows the untrusted OS to initialize enclaves, and swap in/out protected pages to untrusted storage.

Enclave code is restricted to user mode, and has access to all its protected pages, as well as to the unprotected part of the application's address space. Dedicated CPU instructions switch the processor in or out of *enclave mode*. The `eenter` instruction transfers control from the unprotected application context to a predetermined location inside the enclave, and `eexit` can be used to exit an

[1] `https://github.com/jovanbulck/sgx-step`

enclave programmatically. Alternatively, in case of a fault or external interrupt, the processor executes an Asynchronous Enclave Exit (AEX) procedure that saves the execution context securely in a preallocated State Save Area (SSA) inside the enclave, and replaces the CPU registers with a synthetic state to avoid direct information leakage to the untrusted Interrupt Service Routine (ISR). The AEX procedure also takes care of pushing a predetermined Asynchronous Exit Pointer (AEP) on the unprotected call stack, so as to allow the OS interrupt handler to return transparently to unprotected trampoline code outside the enclave. From this point, an interrupted enclave can be continued by means of the `eresume` instruction.

To aid enclave development, SGX differentiates between *debug* and *production* enclaves, where private memory of the former is accessible from outside via special ring-0 `edbgrd` and `edbgwr` instructions. Debug operations are ignored for production enclaves, however, such that they are provided with strong isolation of code and data memory. SGX furthermore includes measures against obvious interference with production enclaves. Specifically, in enclave mode, the processor ignores performance counters, hardware breakpoints, and the single-step trap flag (`rflags.tf`).

4.2.2 Enclave preemption attacks

Given SGX's strong adversary model, several recent studies have looked into its side-channel attack surface. Given the scope of this chapter, we focus exclusively on attacks that preempt the enclaved execution, but it is worth noting that some recent L1 cache attacks [225, 29] can be mounted from a co-resident logical processor, without interrupting the victim enclave. Enclave preemption attacks on the other hand either leverage page faults or interrupts to inspect enclave behavior.

Fault-driven attacks. Seminal work by Xu et al. [277] first showed how carefully revoking access rights on enclave pages and observing the associated page faults, allows an adversarial OS to extract large amounts of sensitive data (full text, and images) from SGX enclaves. Subsequent work [266] has leveraged page faults as an enclave execution control technique to more easily exploit thread synchronization bugs in enclaves. Since page faults are triggered deterministically by the hardware, fault-driven attacks generally suffer from very little to no noise. A fundamental limitation of this channel, however, concerns the relatively coarse-grained (4 KiB) granularity at which page faults reveal memory accesses. Moreover, in order for the enclaved execution to continue, access rights on the faulting pages should be restored.

Interrupt-driven attacks. More recent SGX attacks improve over the spatial resolution of the page fault channel by exploiting information leakage at a cache line granularity. Not all, but a significant fraction of these attacks suspend the victim enclave to obtain precise side-channel observations. Earlier proposals such as CacheZoom [181] rely on a rather coarse-grained kernel patch to interrupt the victim enclave more frequently. More recent work by Hähnel et al. [92] significantly improves the temporal resolution of enclave cache attacks by directly configuring the local APIC timer in kernel space. While their approach approximately interrupts enclaved execution every three instructions, true single-stepping is not achieved, since *(i)* they focus on instructions with memory operands only, and *(ii)* the approach was implemented and evaluated in a software simulator, leaving intricate microarchitectural interactions with real SGX hardware fundamentally unclear.

Recent research [156] on *branch shadowing* attacks demonstrated that enclave-private control flow can be inferred by abusing cache collisions in the CPU-internal Branch Target Buffer (BTB). Such attacks critically rely on the periodic interleaved execution of the victim enclave with carefully aligned spy shadow code. Lee et al. [156] employ a kernel patch to achieve a relatively coarse-grained enclave interrupt granularity of about 50 instructions, which can be further improved to about 5 instructions by disabling the CPU cache hierarchy entirely (`CR0.CD`). Note however that disabling caching of course also invalidates aforementioned CPU cache attacks.

Finally, our own previous work [258] on stealthy page-table-based attacks relies on frequent enclave preemptions to measure page-table access patterns. This work also introduced a highly accurate PTE Flush+Flush technique, where a concurrent spy thread running on another logical core continuously monitors a specific page-table entry, and sends an inter-processor interrupt upon detecting an access. Note that this approach is distinct from single-stepping in that the enclaved execution is only preempted when a specific trigger page was accessed, whereas SGX-Step interrupts *each* instruction sequentially.

4.3 Design and implementation

Our single-stepping objective is to execute an enclave one instruction at a time. Note that advanced x86 hardware debug assistance features such as the single-step trap flag (`rflags.tf`) or hardware breakpoints are explicitly suppressed in enclave mode [114]. Our implementation therefore leverages the OS's control over hardware timer devices to *emulate* this behavior with frequent enclave interrupts.

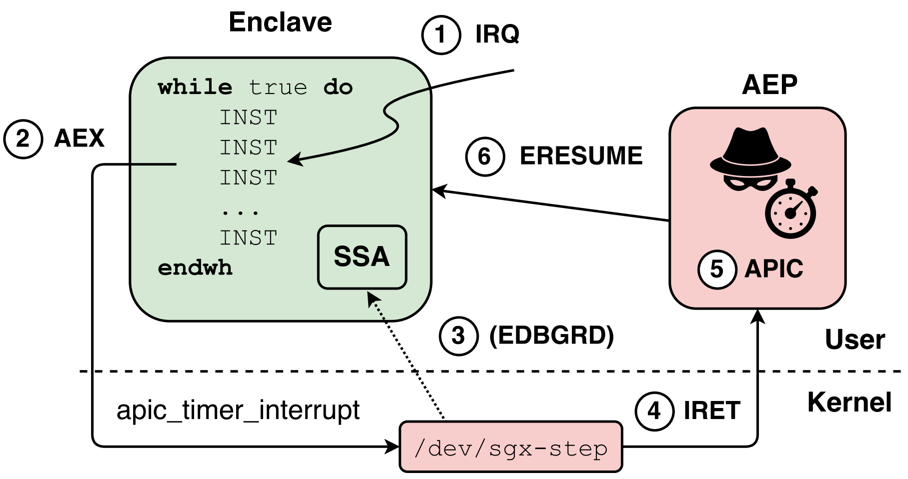


Figure 4.1: Framework for single-stepping SGX enclaves.

APIC timer configuration. Every Intel processor comes with a local Advanced Programmable Interrupt Controller (APIC) [114] to configure and deliver interrupts destined for that core. The APIC also contains a timer that can be operated in one of three modes. In one-shot or periodic mode, the timer is configured through memory-mapped I/O registers. Specifically, by writing into an initial-count register, an internal current-count register can be initialized. The local APIC decrements the current-count at the CPU's bus frequency, divided by the value specified in the divide-configuration register, and generates an interrupt whenever the current-count reaches zero. In one-shot mode a single interrupt is generated, whereas in periodic mode the initial-count is automatically copied back into the current-count register. Alternatively, in TSC-deadline mode, an interrupt is generated when the CPU's internal timestamp counter reaches the absolute value specified in a dedicated model-specific register. This mode is substantially more precise, since the timestamp counter operates at the processor's nominal frequency, instead of the much slower external bus frequency. The Skylake CPUs used in the evaluation, for instance, run at a base frequency of 2.5 GHz and 3.4 GHz, whereas the fixed external bus frequency is only 100 MHz (25/34 times slower).

To facilitate APIC configuration, SGX-Step comes with a runtime library that creates user space virtual memory mappings for the physical APIC memory I/O configuration registers. By writing into the exported memory locations, the untrusted host process can easily configure the APIC timer one-shot/periodic interrupt source or trigger inter-processor interrupts directly from user space. Figure 4.1 summarizes the sequence of hardware and software steps when interrupting and resuming an SGX enclave through our framework. ① The local APIC timer interrupt arrives within an enclaved instruction. ② The processor executes the AEX procedure that securely stores execution context in the enclave's SSA frame, initializes CPU registers, and vectors to the kernel-level interrupt handler. ③ Our `/dev/sgx-step` loadable kernel module registered itself in the APIC event call back list to make sure it is called on every timer interrupt. At this point, any attack-specific, kernel-level spy code can easily be plugged in. Furthermore, to enable precise evaluation of our approach on attacker-controlled debug enclaves, SGX-Step can *optionally* be instrumented to retrieve the stored instruction pointer from the interrupted enclave's SSA frame using the `edbgrd` instruction. ④ The kernel returns to the user space AEP trampoline. We modified the untrusted runtime of the official SGX SDK to allow easy registration of a custom AEP stub. ⑤ At this point, any attack-specific user mode spy code can again easily be run, before the single-stepping adversary configures the APIC timer for the next interrupt, just before executing ⑥ `eresume`.

Timer interval prediction. With our framework in place, the only remaining challenge is to establish a suitable, platform specific timer interval so as to interrupt the first instruction executed by the enclave after `eresume`. The timer interrupt should not systematically arrive too soon, within the monolithic `eresume` instruction, as then no progress would be made (*i.e.*, *zero-step*). Alternatively, should the interrupt arrive too late after completion of `eresume`, more than one instruction would be executed (*i.e.*, *multi-step*). The single-stepping adversary is therefore required to accurately predict the duration between the moment the timer is configured and completion of `eresume`. Naturally, due to modern processor optimizations, execution time prediction becomes increasingly difficult the more code is actually executed in the timer interval. In this respect, previous enclave preemption attempts [156, 181, 92] all configure the APIC timer in kernel space, whereas enclaves have to be resumed in user mode. Consequently, these approaches suffer from significant timer jitter stemming from the considerable amount of code and a privilege level switch in the interrupt return path.

An important contribution of our framework therefore is that we drastically cut the amount of code in the timer interval path by directly configuring the APIC timer from user space. As a result, SGX-Step reduces the timer configuration challenge to prediction of `eresume` execution time, which we found to be relatively deterministic on our evaluation platforms. Our user-space APIC timer trick only works for the aforementioned single-shot or periodic timer modes, however, since TSC deadline configuration requires the privileged `wrmsr` instruction. We thus improve timer interval predictability at the cost of a lower timer frequency. Note that this inconvenience can be overcome, however, for instance by executing a deterministic amount of `nop` instructions between timer configuration and `eresume`.

In our experimental setup, we operate the APIC timer in one-shot mode with division 2. As explained above, timer configuration depends on CPU frequency, and hence remains inherently platform-specific. We established suitable timer intervals for both our evaluation platforms through an empirical approach that leverages SGX-Step to retrieve the interrupted instruction pointer from an attacker-controlled debug calibration enclave. We leave exploration of fully automated timer configuration approaches as future work.

Monitoring page-table entries. Single-stepping enclaved execution incurs a substantial slowdown, and is often only desired for some specific functions of interest. SGX-Step therefore allows an adversary to initiate single-stepping mode after a specific code or data page has been accessed, using enclave preemption from either page faults [277] or a dedicated spy thread [258]. Specifically, analogous to the APIC configuration trick above, SGX-Step establishes user space virtual memory mappings for the *unprotected* physical memory containing the victim enclave's Page Table Entrys (PTEs). By manipulating PTEs directly from user space, an adversary can provoke page faults ("present" bit), or gain insight in enclave memory usage ("accessed" and "dirty" attributes).

4.4 Evaluation

We evaluate the effectiveness of SGX-Step on both a mid-end laptop and a higher-end desktop CPU. We first provide microbenchmarks, and afterwards demonstrate the enhanced attack potential of SGX-Step in two scenarios that are not exploitable with current, state-of-the-art techniques.

All experiments were conducted on real, off-the-shelf SGX hardware. Our first evaluation platform is a commodity Dell Inspiron 13 7359 laptop running a generic Linux 4.2.0 kernel on a Skylake dual-core Intel i7-6500U CPU with a base frequency of 2.5 GHz. Our Dell Optiplex 7040 desktop, on the other hand, features a generic Linux 4.4.0 kernel and a Skylake quad-core i7-6700 processor running at 3.4 GHz. Like previous SGX preemption attacks [92, 181, 156, 258] and conformant to our attacker model, we disabled TurboBoost plus dynamic frequency scaling (C-States, SpeedStep), and affinitized the victim enclave thread to a specific logical core to increase predictability on both machines.

Table 4.2: Interrupts categorized according to the number of instructions executed in the victim enclave (*i.e.*, zero-step, single-step, or multi-step). When laptop/desktop experimental results differ, we present the laptop measurements first.

Experiment	0-Step	1-Step	> 1	1-Step Ratio
`nop`	2,083 / 1,617	100,000	0	97.96 / 98.41%
`strlen`	8,829 / 4,982	460,000	0	98.12 / 98.93%
Zigzagger	5,739 / 2,872	210,000	0	97.34 / 98.65%

4.4.1 Single-stepping microbenchmark

Our objective is to reliably single-step *arbitrary* enclave programs, including inexpensive instructions without memory operands. To evaluate how accurately SGX-Step realizes such true single-stepping, we constructed a challenging microbenchmark experiment featuring a test enclave with a long slide of successive `nop` instructions. At the microarchitectural level, a 1-byte `nop` is the lowest cost instruction, consuming only a single micro-op without memory or register dependencies [113]. As such, many `nop`s can be executed in a limited time window, and even a relatively small amount of jitter on timer interrupt arrivals can lead to the execution of multiple `nop`s in the benchmark enclave. Hence, we argue that an approach that reliably single-steps a `nop` instruction slide, can easily single-step arbitrary instructions as well.

Our benchmark enclave executes a slide of 100,000 successive `nop` instructions. As part of the experiment, we instructed the SGX-Step driver to retrieve the instruction pointer from the state save area of the interrupted debug enclave using the `edbgrd` instruction, so as to infer the exact number of instructions executed in between two successive enclave interrupts.[2] In the evaluation on our laptop/desktop platforms, we measured a total of respectively 102,083 and 101,617 interrupts for the instruction slide. We confirmed that exactly 2,083/1,617 out of these did not change the enclave instruction pointer (*i.e.*, zero-step), whereas the remaining 100,000 interrupts caused a *single* increment of the enclave instruction pointer. We thus conclude that SGX-Step was able to reliably single-step all 100,000 `nop`s, without ever executing more than one `nop` at a time. A small fraction of interrupts (2.04% on the laptop and 1.59% for the desktop) actually arrived too early, within the `eresume` instruction. These interrupts are superfluous, but rather harmless as they do not result in enclaved code being executed.

[2] Note that `edbgrd` only serves evaluation purposes, to establish the number of instructions executed in the benchmark enclave, and would not be used in real attacks against production enclaves.

```c
size_t strlen (char *str)
{
  char *s;

  for (s = str; *s; ++s);
  return (s - str);
}
```

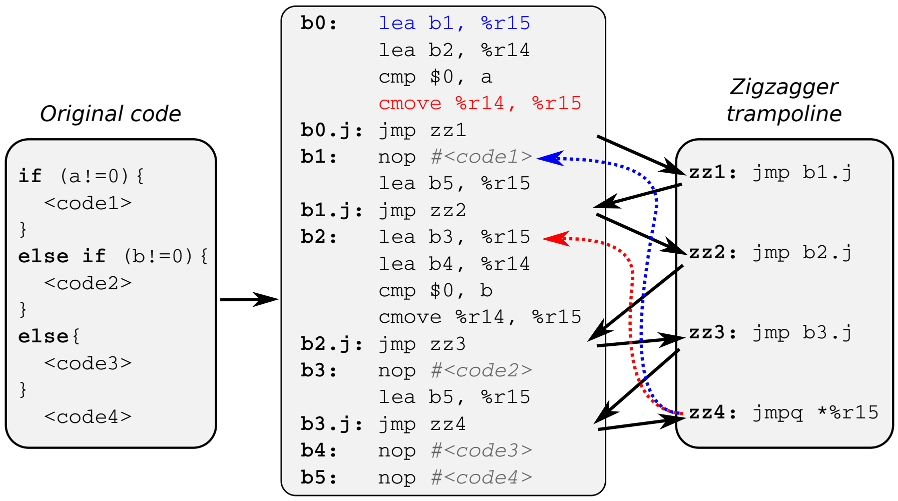
```
    mov   %rdi,%rax
1:  cmpb  $0x0,(%rax)
    je    2f
    inc   %rax
    jmp   1b
2:  sub   %rdi,%rax
    retq
```

Listing 4.1: Example of secret-dependent data accesses in a tight loop (source code and compiled assembly form).

4.4.2 Precise enclave execution control attacks

Determining string length. Previous work [92] explored the temporal resolution limits of the page fault channel, discussed in Section 4.2.2. That is, since an attacker needs to restore access rights on faulting pages in order to guarantee progress, fault-driven attacks cannot infer information from enclaved functions that access a single code and data page. As an example of such a function, consider the elementary `strlen` implementation in Listing 4.1. Assuming the compiler uses a CPU register for the loop counter, the entire loop easily fits within a single code page, and every iteration accesses only one data page (containing the string). As such, progress can only be made if both the `strlen` code page and secret string data page are accessible. That is, the length of the secret string cannot be inferred from page fault sequences. Previous research [258] has shown, however, that page accesses can be observed without page faults, for instance by querying the PTE "accessed" bit after interrupting the enclave. We thus leverage SGX-Step to single-step the tight `strlen` execution loop, each time recording/clearing the "accessed" bit of the PTE referencing the string being processed. Note that accurate single-stepping results themselves also allow the string length to be inferred from the number of interrupts (*i.e.*, instructions executed by the victim enclave).

The right hand side of Listing 4.1 provides the assembly version of the `strlen` C source code on the left. We explicitly compiled the code with optimizations set for size (`-Os`) to ensure a very compact loop with only 4 assembly instructions and a single memory operand. Note that precisely single-stepping this loop is considerably more challenging than the case without optimizations (totalling 5 instructions and 3 memory operands). In our experimental setup, we single-stepped a benchmark enclave that processed the string `"SysTEX 2017"` (11 characters) 10,000 successive times. On every interrupt, just before resuming the enclave, we queried the PTE "accessed" bit from the user space AEP trampoline handler. We correctly recognized the string length for all 10,000 `strlen` invocations. Additionally, we analyzed the full enclave instruction



```
if (a!=0){
  <code1>
}
else if (b!=0){
  <code2>
}
else{
  <code3>
}
  <code4>
```

```
b0:   lea b1, %r15
      lea b2, %r14
      cmp $0, a
      cmove %r14, %r15
b0.j: jmp zz1
b1:   nop #<code1>
      lea b5, %r15
b1.j: jmp zz2
b2:   lea b3, %r15
      lea b4, %r14
      cmp $0, b
      cmove %r14, %r15
b2.j: jmp zz3
b3:   nop #<code2>
      lea b5, %r15
b3.j: jmp zz4
b4:   nop #<code3>
b5:   nop #<code4>
```

```
zz1: jmp b1.j
zz2: jmp b2.j
zz3: jmp b3.j
zz4: jmpq *%r15
```

Figure 4.2: Example code snippet protected by Zigzagger. The final target address in `r15` is obfuscated with `cmov` and a tight trampoline sequence of `jmp` instructions (from [156]).

pointer trace, retrieved with `edbgrd`, to categorize interrupts according to the amount of instructions executed in the victim enclave. The results are in Table 4.2. A first important finding, in line with our microbenchmark observations, is that SGX-Step reliably single-stepped all 460,000 instructions on both the laptop and desktop processors, and without ever executing more than one instruction per interrupt. Only a relatively small fraction of the total number of interrupts ($\leq$ 1.88%) arrived within `eresume` and did not result in an enclaved instruction being executed. These zero-step observations can be easily filtered out, as we confirmed that they never falsely triggered the "accessed" bit of the string PTE.

Defeating Zigzagger. Section 4.2.2 introduced *branch shadowing attacks* that rely on frequent enclave preemptions to execute shadow probing code for inferring enclave-private control flow via targeted BTB cache collisions. This recent work [156] also includes a compile-time defense scheme called Zigzagger. The key idea, illustrated in Fig. 4.2, is to obfuscate the target address of a conditional jump via a `cmov` instruction, followed by a tight trampoline sequence of unconditional jumps that ends with a single indirect branch instruction. By rapidly jumping back and forth between the instrumented code and the trampoline, Zigzagger makes recognizing the current branch instruction considerably more challenging. Its security argument states that "since all of the unconditional branches are executed almost simultaneously in sequence, recognizing the current instruction pointer is difficult" [156]. Moreover, the branch shadowing attack in itself cannot directly infer the secret target address of the indirect branch at `zz4`. We show, however, that even Zigzagger-instrumented code can be reliably single-stepped. Specifically, an attacker leveraging SGX-Step can reliably probe *each* intermediate unconditional trampoline jump (*i.e.*, `zz1` to `zz3`). Observe that after branching to the secret target address, execution continues at one of only two possible target addresses, and eventually lands on the Zigzagger trampoline at either `zz2` or `zz3`. As such, a single-stepping adversary can infer the secret if condition, after detecting execution of the indirect branch at `zz4`, by probing the unconditional `zz2` jump—which is only executed for the first code block.

We evaluate a proof-of-concept Zigzagger attack by repeatedly single-stepping the hardened assembly code[3] from Fig. 4.2. Specifically, we single-stepped a benchmark enclave that executes the 21-instruction code snippet 10,000 successive times, and afterwards analyzed the `edbgrd` instruction pointer trace to establish the number of instructions executed on every interrupt. In line with our previous findings, Table 4.2 shows that SGX-Step *never* executes more than one instruction in the victim enclave per interrupt, allowing precise execution of the shadow code on both evaluation platforms. Hence, these benchmarks can be considered clear evidence that SGX-Step enables *new* attacks, previously deemed infeasible. The superfluous zero-step interrupt fractions (2.66% for the laptop and only 1.35% on the desktop) also keep on par with previous observations, and do not impede a real attack since the BTB cache remains unaffected by the victim.

4.5 Discussion

Attack resolution and implications. We showed that enclaves can be reliably interrupted one instruction at a time. In this, SGX-Step improves significantly over related state-of-the-art enclave preemption schemes that only approximate such instruction-level granularity after either disabling the CPU cache [156], or focusing exclusively on instructions with memory operands in a simulator [92]. From a practical perspective, SGX-Step furthermore lowers the bar for precise enclave preemption attacks from user space.

[3]Since Zigzagger and the attack code were not made public, we repeat the example assembly code snippet from that paper [156]. For the same reason, we could not launch the actual branch shadowing attack, only showing its feasibility with our single-stepping results.

These findings have profound consequences for the design of effective defenses. Specifically, compiler-based techniques are fundamentally insufficient when they rely on (partial) *atomic behavior* of the instruction stream, as effectively demonstrated for the Zigzagger [156] branch obfuscation technique above. Our precise `strlen` attack furthermore highlights the inadequacy of defenses that focus on "aligning specific code and data blocks to exist entirely within a single page", as still officially recommended by Intel [119].

Detecting suspicious interrupts. Heuristic compiler defenses, on the other hand, could focus on detecting high interrupt rates as an artifact of an ongoing attack. Importantly, in contrast to enhanced TEE designs such as Sanctum [48], SGX enclaves are explicitly left *interrupt-unaware*, since they ought to be resumed through a dedicated `eresume` instruction. However, a contemporary line of research [229, 42] leverages x86 Transactional Synchronization Extensions (TSX) to detect page faults or interrupts in enclave mode.

T-SGX [229] protects against page fault-based attacks by wrapping each basic block in a TSX transaction, and aborting the enclave after counting too many consecutive transaction aborts. Déjà Vu [42] instruments an enclave program to detect frequent preemptions through a reliable in-enclave reference clock thread that uses TSX to ensure it cannot be silently stopped by an untrusted OS. Both solutions would recognize the frequent interrupt rates generated by SGX-Step, but also suffer from several important limitations, however. First, an SGX-enabled processor (e.g., the laptop we used in our experiments) is not always shipped with TSX extensions, ruling out this defense for critical infrastructural software such as Intel's Launch and Quoting Enclaves. Second, TSX defenses incur a significant run-time performance overhead [229, 42]. Third, these defenses cannot offer full protection as they rely on heuristics to recognize suspicious interrupt rates, which could also be caused by repeated cache conflicts or benign interrupts under heavy system load.

4.6 Conclusion

Our work shows that enclaved execution can be accurately single-stepped one instruction at a time. We demonstrated SGX-Step's improved temporal resolution over state-of-the-art preemption schemes in two challenging attack scenarios, highlighting the need for adequate defense mechanisms.

Chapter 5

Nemesis: Studying microarchitectural timing leaks in rudimentary CPU interrupt logic

This chapter was previously published as:

> J. Van Bulck, F. Piessens, and R. Strackx. "Nemesis: Studying Microarchitectural Timing Leaks in Rudimentary CPU Interrupt Logic". In: *25th ACM Conference on Computer and Communications Security (CCS)*. Oct. 2018, pp. 178–195

Preamble

This chapter presents Nemesis, a previously overlooked side-channel attack vector that abuses the CPU's interrupt mechanism to leak microarchitectural instruction timings from enclaved execution environments. We show that by measuring the latency of a carefully timed interrupt, an attacker controlling the system software is able to infer instruction-granular execution state from hardware-enforced enclaves. Our novel attack vector is applicable to the whole computing spectrum, from small embedded sensor nodes to high-end commodity x86 hardware. We present practical interrupt timing attacks against the open-source Sancus embedded research processor, and we show that interrupt latency reveals microarchitectural instruction timings from off-the-shelf Intel SGX enclaves. Finally, we discuss challenges for mitigating Nemesis-type attacks at the hardware and software levels.

This research originated from our early attempts to create a deterministic, real-time compatible enclave processor with a strict upper bound on the worst-case interrupt latency [253]. When implementing this design on top of the 16-bit Sancus processor, we noticed that interrupt response time is not constant, but varies depending on the instruction executing in the enclave at the time of interrupt arrival. After confirming that these subtle timing differences can be successfully abused to extract sensitive information from Sancus enclaves, we set out to explore the same side-channel attack vector on off-the-shelf Intel SGX platforms. As part of this research, we created several Linux kernel prototypes, contributing to the discovery of the stealthy page-table cache attacks, introduced in Chapter 3, and ultimately leading to the creation of the SGX-Step framework, discussed in Chapter 4. The earliest Nemesis prototypes demonstrating side-channel leakage from Intel SGX enclaves date back to June 2016, which, at that time, would have placed Nemesis among the first demonstrated microarchitectural attacks on SGX platforms. In this respect, Nemesis also pioneered the security analysis of CPU pipeline behavior and exception handling logic, which became defining for today's transient-execution era. We indeed argue that, at its core, Nemesis abuses the same subtle microarchitectural behavior that enables Meltdown and Foreshadow, *i.e.*, handling of exceptions and interrupts is delayed until instruction retirement.

We publicly released all of the code and experimental data for the attacks described in this chapter, and we have been in contact with several independent research groups that successfully reproduced our findings. We are currently aware of at least two public results that use our open-source Nemesis interrupt latency framework to investigate related microarchitectural behaviors on Intel SGX processors [130, 208]. The Sancus interrupt latency code base has furthermore been extended in several master theses [49] and has sparked an ongoing line of follow-up defensive projects. To assist defenses at the software level, one recent work [206] has extended and improved a prior static analysis tool [52] to automatically verify MSP430 binaries to be free of instruction-granular information leakage from Nemesis-style interrupt latency measurements. As an alternative, fully transparent defense at the hardware level, we recently designed and implemented a provably secure Sancus processor with a Nemesis-resistant enclave interrupt mechanism, relying on an intricate two-level execution time padding scheme [34]. We further discuss this defense and its (non-)applicability to SGX platforms in Chapter 8.

5.1 Introduction

Information security is essential in a world with a growing number of ever-connected embedded sensor nodes, mixed-criticality systems, and remote cloud computing services. Today's computing platforms isolate software components belonging to different stakeholders with the help of a sizeable privileged software layer, which in turn may be vulnerable to both logical bugs and low-level vulnerabilities. In response to these concerns, recent research and industry efforts developed Trusted Execution Environments (TEEs) [235, 166] to safeguard security-sensitive application components or *enclaves* from an untrusted operating system. TEEs enforce isolation and attestation primitives directly in hardware, or in a small hypervisor, so as to ensure protected execution with a minimal Trusted Computing Base (TCB). The untrusted operating system is prevented from accessing enclaved code or data directly, but continues to manage shared platform resources such as system memory or CPU time. Enclaved execution is a particularly promising security paradigm in that it has been explicitly applied to establish trust in both low-end embedded microcontrollers [238, 56, 191, 147, 51, 28] as well as in higher-end desktop and server processors [174, 173, 100, 58, 232, 48]. With the arrival of the Software Guard Extensions (SGX) [176, 14] in recent Intel x86 processors, strong hardware-enforced TEE guarantees are now available on mainstream consumer hardware.

TEEs pursue a black box view on enclaves. That is, a kernel-level attacker should only be able to observe input-output behavior, and is prevented from accessing an enclave's private memory directly. While such interactions are generally well-understood at the architectural level, including successful TCB verification efforts [142, 62], enclave-internal behavior may still leak through the CPU's underlying microarchitectural state. Over the past decade, microarchitectural side channels have received considerable attention from academics [202, 2, 280, 71], but their disruptive real-world impact only recently became clear with the Meltdown [162], Spectre [146], and Foreshadow [249] attacks that rely on side channels to steal secrets from the microarchitectural transient-execution domain. We therefore argue that it is essential for the research community to deepen its understanding in microarchitectural CPU behavior and to identify potential side-channel attack vectors. In this respect, recent research on *controlled channels* [277] has shown that conventional side-channel analysis changes drastically when TEEs are targeted, for the operating system itself has become an untrusted agent. The increased attacker capabilities bring about two major consequences.

First, with an untrusted operating system, an adversary gains full control over the unprotected part of the application, and over system events such as interrupts, page faults, cache flushes, scheduling decisions, etc. These types of events introduce considerable noise in traditional cross-application, or even cross-virtual machine side channels. Noise is traditionally compensated for with statistical analysis over data acquired from multiple runs of the victim program. In a controlled-channel setting on the other hand, one prevailing research line is exploring the possibility of *amplifying* conventional side channels so as to extract sensitive information in a single run, with limited noise. Recent work on Intel SGX platforms has practically demonstrated such side-channel amplification for the usual suspects: CPU caches [79, 225, 29, 181, 92] and branch prediction machinery [156, 59]. These results have prompted Intel to release an official statement, arguing that "in general, these research papers do not demonstrate anything new or unexpected" [133].

A second, more profound consequence of the TEE attacker model, however, is the emergence of an entirely new class of side channels that were never considered relevant before. To date only page-table-based attacks [277, 230, 258, 263] have been identified as one such innovative controlled channel for high-end MMU-based architectures. By carefully revoking access rights on protected memory pages and observing the associated page accesses, an adversarial operating system is able to extract large amounts of sensitive data (cryptographic keys, full text, and images) from SGX enclaves. Several authors [63, 245, 47, 257, 237, 158] have since expressed their concerns on controlled-channel vulnerabilities in a TEE setting. An important research question therefore is to determine which novel controlled channels exist, and to what extent they endanger the TEE protection model.

This chapter contributes to answering this question. We present an innovative class of Nemesis[1] controlled-channel attacks that exploit subtle timing differences in the rudimentary fetch-decode-execute operation of programmable instruction set processors. We abuse the key microarchitectural property that hardware interrupts/faults are only served upon instruction retirement, *after* the currently executing instruction has completed, which can take a variable amount of CPU cycles depending on the instruction type and the microarchitectural state of the processor. Where Meltdown-type "fault latency" attacks [162, 249] exploit this time window in modern out-of-order processors to transiently leak unauthorized memory through a microarchitectural covert channel, Nemesis-type interrupt latency attacks abuse a more fundamental observation that equally affects non-pipelined processors. Namely, that delaying interrupt handling until instruction retirement introduces a subtle timing difference that *by itself* reveals side-channel information about the interrupted instruction and the microarchitectural state when the interrupt request arrived. Intuitively, an untrusted operating system

[1] From the ancient Greek goddess of retribution who inevitably intervenes to balance out good and evil; an inescapable agent much like a pending interrupt request.

can exploit this timing measurement when interrupting enclaved instructions to differentiate between secret-dependent program branches, or to extract information for different side-channel analyses (e.g., trace-driven cache [1], address translation [258], or false dependency [180] timing attacks).

We are the first to recognize the threat caused by instruction set architectures with variable interrupt latency. Previous TEE research has overlooked this subtle attack vector, claiming for instance that "timing of external interrupts does not depend on secrets within compartments, and does not leak confidential information" [63]. We show that Nemesis attacks affect a wide range of security architectures, covering the whole computing spectrum. In this, we are the first to identify a remotely exploitable microarchitectural side-channel vulnerability that is *both* applicable to embedded, as well as higher-end enclaved execution environments.

Summarized, the main contributions of this chapter are:

- We leverage interrupt latency as a novel, non-conventional side channel to extract information from enclaved applications, thereby advancing microarchitectural understanding.
- We present the first controlled-channel attack vector for embedded enclaved execution processors, and extract full application secrets in practical Sancus attack scenarios.
- We provide clear evidence that interrupt latency reveals microarchitectural instruction timings on modern Intel SGX processors, and illustrate Nemesis's increased instruction-granular potential in macrobenchmark evaluation scenarios.
- We explain how naive hardware-level defense strategies cannot defend against advanced Nemesis-style interrupt attack variants, demonstrating the consequential impact of our findings for provably side-channel resistant processors.

Our attack framework and evaluation scenarios are available as free software at `https://github.com/jovanbulck/nemesis`.

5.2 Background and basic attack

We first refine the threat model and the class of security architectures affected by our side channel. Next, we explain how interrupt latency can be leveraged in ideal conditions to extract sensitive data from secure enclaves.

5.2.1 Attacker model and assumptions

The adversary's goal is to derive information regarding the internal state of an enclaved application. In this respect, trusted computing solutions including Intel SGX have been explicitly put forward to protect sensitive computations on an untrusted attacker-owned platform, both in an untrustworthy cloud environment [20, 217], as well as to enforce enterprise right management on consumer hardware [99, 193]. Analogous to previous enclaved execution attacks [277, 258, 156, 91], we therefore consider an adversary with *(i)* access to the (compiled) source code of the victim application, and *(ii)* full control over the Operating System (OS) and unprotected application parts. This means she can modify BIOS options, load kernel drivers, configure hardware devices such as timers, and control scheduling decisions. Note that although TEEs can be leveraged [78, 217] to protect the confidentiality of sensitive code, this is not the default case in the security architectures analyzed in this work and for many of the TEE use cases [235, 99, 20].

At the architectural level, we assume the untrusted OS can securely interrupt and resume enclaves. Such interruptible isolated execution is supported by a wide range of mature embedded [147, 51, 28] as well as higher-end [176, 100, 58, 232, 48] TEEs that employ a trusted security monitor to preserve the confidentiality and integrity of an enclave's internal state in the presence of asynchronous interrupt events. In this chapter we focus exclusively on hardware-level security monitors, but our timing channel may also be relevant for architectures where enclave interruption proceeds through a small trusted software layer [100, 28, 48, 62]. We assume that enclaves can be interrupted repeatedly within the same run, and for the Intel SGX application scenarios, can be made to process the same secret-dependent input repeatedly over multiple invocations.

Importantly, in contrast to previous controlled-channel attacks referenced above, our attack vector does *not* necessarily require advanced microarchitectural CPU features, such as paging, caching, branch prediction, or out-of-order execution. Instead, Nemesis-type interrupt timing attacks only assume a generic stored program computer with a multi-cycle instruction set, where each individual instruction is uninterruptible (*i.e.*, executes to completion). This is the most widespread case for major embedded (e.g., TI MSP430, Atmel AVR) as well as higher-end (e.g., x86, openRISC, RISC-V) instruction set architectures.

5.2.2 Fetch-decode-execute operation

Figure 5.1 summarizes the basic operational process of a CPU, traditionally referred to as the *fetch-decode-execute* operation. A dedicated Program Counter

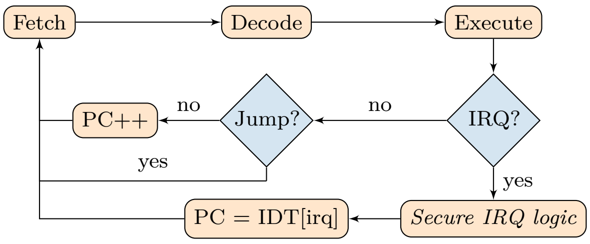


Figure 5.1: A processor fetches, decodes, and executes the instruction referred by the Program Counter (PC) register.

(PC) register holds the address of the next instruction to fetch from memory. PC is automatically incremented after every instruction in the program, and can be explicitly changed by means of jump instructions. Hardware devices furthermore have the ability to halt execution of the current program by means of Interrupt Requests (IRQs) that notify the processor of some asynchronous external event that requires immediate attention. Whenever the current instruction has completed, before fetching the next one, the processor checks if there are IRQs pending. If so, the PC is loaded from a predetermined location in the Interrupt Descriptor Table (IDT) that holds the address of the corresponding Interrupt Service Routine (ISR). Typical processor architectures only take care of storing the minimal execution context (e.g., PC and status register) before vectoring to the ISR. The trusted OS interrupt handling code then stores any remaining CPU registers as needed. However, when interrupting an enclave, the TEE hardware is responsible to securely store and clear *all* CPU registers, which is abstracted in the "secure IRQ logic" block of Fig. 5.1.

While the simplified fetch-decode-execute description above is representative for a class of low-end CPUs such as the TI MSP430 [243], optimizations found in modern higher-end processors considerably increase the complexity. A pipelined architecture improves throughput by parallelizing the fetch-decode-execute stages of subsequent instructions. In case of a complex instruction set such as Intel x86 [114, 47], individual instructions are first split into smaller *micro-ops* during the decode stage. Thereafter, an out-of-order engine schedules the micro-ops to available execution units, which may be duplicated to further increase parallelism. To minimize pipeline stalls from program branches, the processor will try to predict the outcome of conditional jumps. Simultaneous multithreading technology can interleave the execution of multiple independent instruction streams on the same physical CPU core to maximize the use of available execution units. Repeated memory accesses are furthermore sped up by means of an intricate cache hierarchy for among others micro-ops, instructions, data, and address translation. However, despite all these optimizations, Intel [114] confirms that the basic property remains that "all interrupts are guaranteed to be taken on an instruction boundary [...] located during the retirement phase of instruction execution".

5.2.3 Basic Nemesis attack

We consider processors that serve interrupts after the execute stage has completed,[2] which can take multiple clock cycles depending on the microarchitectural behavior of the instruction. Our attacks are based on the key observation that *an IRQ during a multi-cycle instruction increases the interrupt latency with the number of cycles left to execute*—where interrupt latency is defined as the number of clock cycles between arrival of the hardware IRQ and execution of the first instruction in the software ISR. When interrupt arrival time is known (e.g., generated by a timer), untrusted system software can infer the duration of the interrupted instruction from a timestamp obtained on ISR entry.

Figure 5.2 illustrates our basic attack for an enclaved execution that branches on a secret. After the conditional jump *jz* in the victim enclave, either the two-cycle instruction $inst_1$ or the three-cycle instruction $inst_2$ is executed. In an ideal environment, a kernel-level attacker proceeds as follows to determine private control flow. First, before executing the enclave, a cycle-accurate timer is configured to schedule an IRQ at the beginning of the first clock cycle $x + 1$ after the conditional jump instruction. Next, the enclave is entered and the timer fires, interrupting either $inst_1$ or $inst_2$. After instruction completion, the secure hardware stores and clears protected execution state, and hands over control to the untrusted interrupt handler code. Here, the adversary compares the value of a timestamp counter with the known IRQ arrival time to yield a timing difference of one clock cycle, depending on whether the conditional jump in the enclaved execution was taken or not.

The above scenario is a clear example of how an untrusted OS can leverage interrupt latency to break the black box view on enclaves. In line with previous enclaved execution attacks [277, 29, 258, 156], Nemesis-type interrupt timing attacks exploit secret-dependent control flow. Specifically, we require a different execution time for at least one instruction in the if/else branch. The adversary furthermore relies on *(i)* a timer device capable of generating cycle-accurate IRQs, and *(ii)* a Time Stamp Counter (TSC) peripheral that is incremented every CPU cycle. The main difficulty for a successful attack lies in determining a suitable timer value so as to interrupt the instruction of interest. This is non-trivial in that it requires one to predict the duration between the moment the

[2] While not the focus of this chapter, there are also issues with cancelling the currently executing instruction upon IRQ arrival, as outlined in Section 5.6.

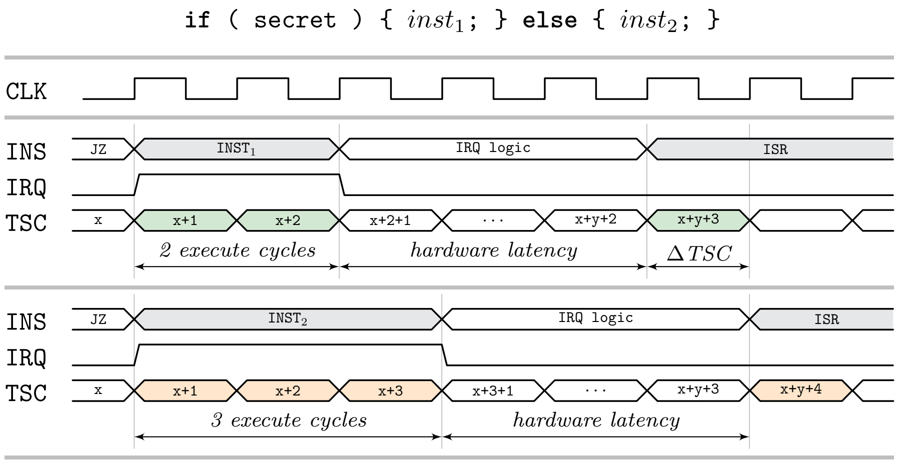


Figure 5.2: Interrupt latency leaks information about the instruction that was executing at the time of IRQ arrival.

timer is configured and the desired interruption point. For reasons pointed out above, it is challenging to precisely predict the execution time of an instruction stream on modern processors. We present our approach to configuring the timer and dealing with noise in Section 5.4.

Note that IRQ latency measurements capture an instruction-granular measurement of the CPU's microarchitectural state, such that the instruction opcode ($inst_1$ vs. $inst_2$) is only one of many properties that influence latency on modern processors. We will show in Section 5.5 that Nemesis adversaries can also distinguish instructions based on, for instance, CPU caching behavior, address translation, or data operand dependencies.

5.3 Case study platforms and attacks

We implemented and evaluated Nemesis-type interrupt timing attacks for both a representative embedded, as well as for an off-the-shelf higher-end enclaved execution processor. To illustrate the wide applicability of conditional control flow side-channel attacks, beyond common cryptographic key extraction [230, 258, 79, 225, 181], we follow a line of enclaved execution attacks [277, 29, 156, 92, 257] that target non-cryptographic case study applications. Such applications cannot be hardened straightforwardly using vetted crypto libraries, as secrets are generally non-trivial to identify and conditional control flow is more prevalent plus harder to eliminate.

5.3.1 Sancus and embedded TEEs

Given the rise of tiny embedded devices in recent years, a new line of research [238, 56, 191, 147, 28] employs a lightweight program counter based memory isolation technique to secure small microcontrollers that lack hardware support for established security measures, such as virtual memory and processor privilege levels. The Sancus [191, 189] research prototype extends the memory access logic and instruction set of a low-end TI MSP430 microcontroller to allow the creation, authentication, and destruction of enclaved software modules with a hardware-only TCB. Furthermore, enclaves residing on the same device can securely link to each other using caller and callee authentication primitives. A dedicated LLVM-based C compiler hides low-level concerns such as secure linking, inter-enclave calling conventions, and private call stack switching by inserting short assembly code stubs to be executed whenever an enclave is entered or exited. Finally, recent research [192, 252] has shown that, in contrast to Intel SGX platforms, Sancus' memory isolation primitive can also be used to provide software enclaves with exclusive access to Memory Mapped I/O (MMIO) hardware peripheral devices. However, since Sancus enclaves only feature a single contiguous private data section, secure I/O on Sancus requires the use of a small driver enclave entirely written in assembly code, using only registers for data storage.

The original Sancus architecture presumes uninterruptible isolated execution. Secure interruption of hardware-enforced embedded enclaves was pioneered by the TrustLite [147] TEE. More specifically, TrustLite modifies the processor to push all CPU registers onto the private call stack of the interrupted enclave, before clearing them and vectoring to the untrusted ISR. Subsequent research [51] has since implemented a comparable hardware-level interrupt mechanism for a prototypic Sancus-like TEE with a single secure domain, and recent work-in-progress [253] reports on hardware and compiler support for fully interruptible and reentrant Sancus enclaves. For the work presented in this chapter, we have implemented TrustLite's secure interrupt mechanism as an extension to the original Sancus architecture. Furthermore, we extended the compiler-generated entry stubs to restore private execution context on the next invocation of a previously interrupted enclave.

We selected Sancus as the case study architecture representative for the lowest end of the computing spectrum with strict security requirements for mutually distrusting stakeholders. A recent exhaustive TEE overview [166] indicates that Sancus is the only embedded architecture with a fully open-source[3] hardware design and tool chain, which allowed us to develop the secure interrupt

[3] `https://distrinet.cs.kuleuven.be/software/sancus` and `https://github.com/sancus-tee`

extensions. In contrast to modern SGX processors, Sancus' openMSP430-based implementation embodies an elementary programmable microcontroller without advanced architectural features such as paging, caches, or out-of-order instruction pipelining. Given the simplistic design of the security extensions, as well as the underlying processor, the existence of remotely exploitable side channels was considered rather unlikely by the original designers [190, §7.5.3]. To the best of our knowledge, we present the first controlled-channel attack vector for embedded enclaved execution processors.

Bootstrap loader. We illustrate the applicability of our basic attack with a code snippet from an actual password comparison routine in Texas Instruments' MSP430 serial Bootstrap Loader (BSL) implementation. The BSL software is executed on platform reset, and enables remote, in-field firmware updates. To enforce that only legitimate device owners can reprogram the microcontroller, sensitive BSL commands are protected with a 32-byte password. Our first Sancus application scenario employs hardware-enforced isolation to shield critical BSL password-protected functionality from untrusted embedded firmware.

```
   cmp.b @r6+, r12
   jz    1f
   bis   #0x40, r11
1: ...
   ...
```

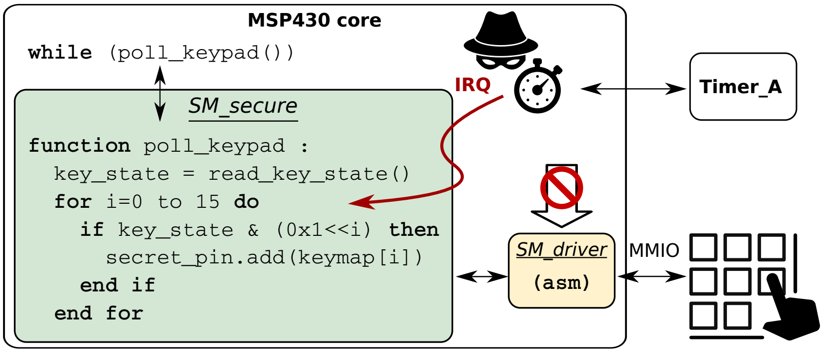
```
   cmp.b @r6+, r12
   jz    1f
   bis   #0x40, r11
   jmp   2f
1: nop nop nop nop 2: ...
```

Listing 5.1: (Un)balanced BSL password comparison.

However, the password comparison routine in some BSL versions is known to be vulnerable to an execution timing attack [76]. The left hand side of Listing 5.1 provides the original, actually used assembly code.[4] For clarity we only show the body of the password comparison loop, where the byte pointed to by `r6` is compared with the value in `r12`, and a bit in `r11` is set to invalidate access when the comparison fails. Observe that the code is unbalanced in that the two-cycle `bis` (bit-set) instruction is only executed for incorrect password bytes. Hence, an adversary can determine the correctness of individual bytes by observing the program's overall execution time. We close this vulnerability in the hardened version on the right by balancing the else branch with no-op compensation code, as previously suggested in literature [209, 43].

We show that, even when executing the balanced password comparison routine in a Sancus enclave, untrusted system software can still learn the correctness of individual password bytes by carefully timing interrupts. More specifically, an IRQ arriving in the first clock cycle after the conditional jump instruction, will

[4]Assembly code snippet from BSL v2.12, as published by [76].



```
while (poll_keypad())

function poll_keypad :
  key_state = read_key_state()
  for i=0 to 15 do
    if key_state & (0x1<<i) then
      secret_pin.add(keymap[i])
    end if
  end for
```

Figure 5.3: Secure keypad Sancus application scenario.

either interrupt the two-cycle `bis` instruction or the single-cycle `nop` instruction. Hence, depending on the secret password byte, an IRQ latency difference of one clock cycle will be observed. The hardened routine thus properly closes the timing side channel at the architectural assembly code level, but unknowingly introduces a new one at the microarchitectural level. As such, our elementary BSL case study serves as a clear demonstration of the additional attack surface induced by secure interrupts, where adversaries are no longer restricted to start-to-end timing measurements of the enclaved computation.

Secure keypad. Various authors [56, 191, 147, 192, 252] have suggested the use of small TEEs to securely interface embedded platforms with peripheral I/O devices. Our second Sancus application scenario leverages secure I/O to guarantee the secrecy of a 4-digit PIN code towards an untrusted embedded operating system.

Figure 5.3 summarizes the core idea, where the security-sensitive application logic is implemented in a protected SM_{sec} enclave that securely links to a dedicated SM_{drv} assembly enclave to gain exclusive access to the MMIO region of the keypad peripheral, as explained above. The untrusted OS can only interact with the keypad indirectly, through the public interface offered by SM_{sec}. A single entry point `poll_keypad` fetches the current key state from the driver enclave, and processes each bit sequentially. The 16-bit key state indicates which keys are down, and a static lookup table is used to translate key numbers to the corresponding characters. This is similar to a reference implementation for an unprotected MSP430 keypad application by Texas Instruments [178]. To increase readability, the pseudo code in Fig. 5.3 omits practical concerns such as detecting key releases and limiting the length of the PIN code. We refer the interested reader to Appendix C.2 for the full implementation, derived from a recently published open-source Sancus automotive application case study [252].

The keypad has to be polled regularly to detect key presses. For this, our application scenario relies on the untrusted operating system for availability of the CPU time resource. Since the OS is in control of scheduling decisions, it is allowed to interrupt SM_{sec} at all times.[5] Our attack exploits key state dependent control flow in the `poll_keypad` function. Appendix C.2 provides the full compiler-generated assembly code, but it suffices to say that the conditional code path consists of two single-cycle instructions followed by either a single-cycle `tst` or a two-cycle `cmp` instruction. If we succeed in timing an IRQ two cycles after the conditional jump, we will thus observe a difference in interrupt latency of one clock cycle, depending on whether the private key state bit was set or not. Reconfiguring the timer to repeat the attack in each loop iteration allows an untrusted ISR to unambiguously determine which keys were pressed in a single run of SM_{sec}.

5.3.2 Intel Software Guard Extensions

Recent Intel x86 processors include Software Guard Extensions (SGX) [176, 14] that enable isolated execution of security-critical code in hardware-enforced enclaves, embedded in the virtual address space of a conventional OS process. SGX reduces the TCB to the point where a remote software provider solely has to trust the implementation of her own enclave, plus the underlying processor. Enclave code is restricted to user space (ring 3), and has access to all its protected pages, as well as to the unprotected part of the host application's address space. Dedicated CPU instructions switch the processor in and out of *enclave mode*, where hardware-level access control logic verifies the output of the untrusted address translation process to safeguard enclaved pages from outside accesses. The `eenter` instruction transfers control from the unprotected application context to a predetermined location inside the enclave, and `eexit` leaves an enclave programmatically. Alternatively, in case of a fault or external interrupt, the processor executes an Asynchronous Enclave Exit (AEX) procedure that saves the execution context securely in a preallocated state save area inside the enclave, and replaces the CPU registers with a synthetic state to avoid direct information leakage to the untrusted ISR. The AEX procedure also takes care of pushing a predetermined Asynchronous Exit Pointer (AEP) on the unprotected call stack, so as to allow the OS interrupt handler to return transparently to

[5] Note that Sancus' secure IRQ logic stores execution state in the protected data section of the interrupted enclave. For MMIO driver enclaves without general purpose private data region, our hardware mechanism clears registers without saving them.

unprotected trampoline code outside the enclave. From this point, a previously interrupted enclave can be continued by means of the `eresume` instruction.

Intel SGX serves as our case study architecture for higher-end enclaved execution platforms. A modern SGX-enabled CPU implements the complex x86 instruction set architecture, and includes all advanced microarchitectural features found in modern processors.

Zigzagger branch obfuscation. Recent research on branch shadowing attacks [156] showed that enclaved control flow can be inferred by probing the CPU-internal Branch Target Buffer (BTB) after interrupting a victim enclave. Given the prevalence of conditional control flow in existing non-cryptographic applications, this work also includes a practical compile-time hardening scheme called Zigzagger. Figure 4.2 on page 111 shows how secret-dependent program branches are translated into an oblivious `cmov` instruction, followed by a tight trampoline sequence of unconditional jumps that ends with a single indirect branch instruction. The key idea behind Zigzagger is to prohibit probing the BTB for the current branch instruction by rapidly jumping back and forth between the instrumented code and the trampoline such that recognizing the current instruction pointer becomes difficult. It has since been shown, however, that Zigzagger-instrumented code can be reliably interrupted one instruction at a time [257], and concurrent research defeated Zigzagger in restricted circumstances through a segmentation-based side channel [91].

We will show that even the contained conditional control flow in Zigzagger-hardened code exhibits definite instruction timing differences that can be recognized to extract application secrets from IRQ latency traces. Particularly, to emphasize Nemesis's increased precision over state-of-the-art SGX attacks, we aligned the assembly code of Fig. 4.2 to fit entirely within one cache line, such that execution paths cannot be distinguished by their corresponding code cache or page access profiles [230]. Our Zigzagger attack scenario thus illustrates that Nemesis-type interrupt latency attacks leak microarchitectural timing information at the granularity of *individual* instructions, whereas previous controlled channels only expose enclaved memory accesses at a relatively coarse-grained 4 KiB page [277, 258] or 64-byte cache line [225, 92] granularity.

Binary search. Intel SGX technology has been explicitly put forward for securely offloading privacy-sensitive data analytics to an untrusted cloud environment [217]. Our second SGX application scenario considers enclaves that look up secret values in a known dataset, as it occurs for instance in privacy-friendly contact discovery [169] or DNA sequence processing [260, 29]. In case of the former, the enclave is provided with a known large list of users, plus an encrypted smaller list of secret contacts, and is requested to return only those contacts that occur in the known user list. In case of the latter, the enclave may lookup values in a public reference human genome dataset, based on an encrypted secret input tied to an individual. In both scenarios, adversaries may track control flow decisions made, for instance, by the widely used binary search algorithm to learn (parts of) the secret input. In this respect, binary search serves as a particularly relevant example for the difficulty of eliminating conditional control flow in general-purpose enclave programs. The obvious alternative at the application level, an exhaustive scan of the public data, would increase the time complexity from a logarithmic to a linear effort.

```
for (lim = nmemb; lim != 0; lim >>= 1) {
    p = base + (lim >> 1) * size;
    cmp = (*compare)(key, p);
    if (cmp == 0) return p;
    if (cmp > 0) {  /* key > p: move right */
        base = p + size; lim--;
    } /* else move left */
}
```

Listing 5.2: Binary search routine in Intel SGX Linux SDK.

Listing 5.2 shows the relevant part of the actual binary search routine provided by the official Intel SGX Linux SDK. We refer to Appendix C.3 for the complete unmodified source code, plus a disassembly of the compiled version. The implementation looks up a provided key in the sorted array between `base` and `lim` by repeatedly comparing it to the middle value. If the provided key was found, the function returns. Otherwise, the values of `base` and `limit` are adjusted according to whether the provided key was greater or smaller than the middle value. We will show that the assembly code paths corresponding to whether the algorithm took the left, right, or equal branch, manifest subtle yet distinct instruction latency patterns which are revealed in the extracted IRQ latency traces. As with the Zigzagger example above, the secret lookup key is learned even when the array fits entirely within a single cache line. For larger arrays, motivated adversaries can develop highly practical hybrid approaches that start with tracking array indices at a 4-KiB page-level granularity, over to a finer-grained 64-byte cache line granularity within a page, before finally leveraging Nemesis's instruction-granular interrupt timing differences to infer comparisons within a cache line.

5.4 Implementation aspects

We first describe the Sancus case, and then explain how distinctive IRQ latency traces can be extracted from SGX enclaves.

5.4.1 Implementation on Sancus platforms

Our Sancus case study attacks exploit timing differences as subtle as a single CPU cycle. In order to do so, the timer interrupt has to arrive at exactly the right time, within the first clock cycle of the enclaved instruction of interest. There is no room for deviation here, as a shift of a single cycle may miss the instruction we are aiming for and corrupt the latency timing difference.

Conveniently, the standard TI MSP430 architecture [243] comes with a Timer_A peripheral capable of generating cycle-accurate interrupts. The timer features an internal Timer_A Register (TAR) that is incremented every clock cycle, and can be configured to generate an IRQ upon reaching a certain value. After generation of the interrupt request, Timer_A immediately restarts counting from zero. Hence, interrupt latency on MSP430 microcontrollers can be measured trivially by reading TAR as the first instruction in the ISR. The key to a successful exploit thus comes down to determining the amount of clock cycles between configuring the timer, and execution of the instruction of interest in the enclave. Again, this is relatively straightforward on an MSP430 microcontroller where, in the absence of pipelining and caching, execution timing is *completely* deterministic. More specifically, instruction execution takes between one and six clock cycles, depending on the addressing modes of the source and destination operands. An MSP430 CPU [243] features seven different addressing modes, yielding a large variation in possible execution cycles. We refer to Appendix C.1 for a full instruction timing table.

Careful analysis of the compiled source code thus suffices to establish appropriate timer configurations for the Sancus application scenarios. To make our exploits more robust against changes in the application's source code, however, we opted for a different approach where the attacker first deploys a near-exact copy of the victim enclave, adjusted to copy the value of TAR in a global variable directly after execution of the conditional jump of interest. Our practical attack combines the execution timings retrieved from this "spy" enclave with predetermined constant parameters to dynamically configure the timer at runtime.

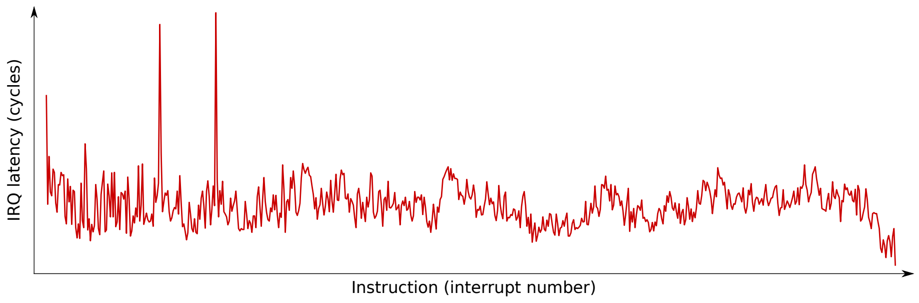


Figure 5.4: Sample interrupt latency trace revealing execution timings for individual SGX enclave instructions.

5.4.2 IRQ latency traces on SGX platforms

SGX enclave programs are explicitly left interrupt-unaware by design. While an x86 processor [114] in enclave mode ignores obvious hardware debug assistance features such as the single-step trap flag (`rflags.tf`) or hardware breakpoints, recent research on interrupt-driven SGX attacks [257, 92, 181, 156] has shown that untrusted OSs can accurately emulate this behavior by leveraging first-rate control over timer devices. So far, these attacks have focused on collecting side-channel information from frequent enclave preemptions via the page tables, CPU caches, or the branch prediction unit. We are the first to recognize, however, that the *act* of interrupting a victim enclave in itself leaks microarchitectural instruction timings.

We explain below how we extended and improved a state-of-the-art enclave single-stepping framework to collect precise interrupt latency measurements from SGX enclaves. The resulting IRQ latency traces, exemplified in Fig. 5.4, describe the execution time for *each* subsequent instruction in the enclaved computation, and can thus be thought of as an “x-ray” of the microarchitectural processor state and the code executing in the enclave.

Single-stepping enclaved execution. We based our implementation on the recently published open-source SGX-Step [257] framework that allows a privileged adversary to precisely “single-step” enclaves at most one instruction at a time. SGX-Step comes with a Linux kernel driver to establish convenient user space virtual memory mappings for enclave Page Table Entrys (PTEs) and the local Advanced Programmable Interrupt Controller (APIC) device. A very precise single-stepping technique is achieved by writing to the APIC timer register directly from user space, eliminating any jitter from kernel context

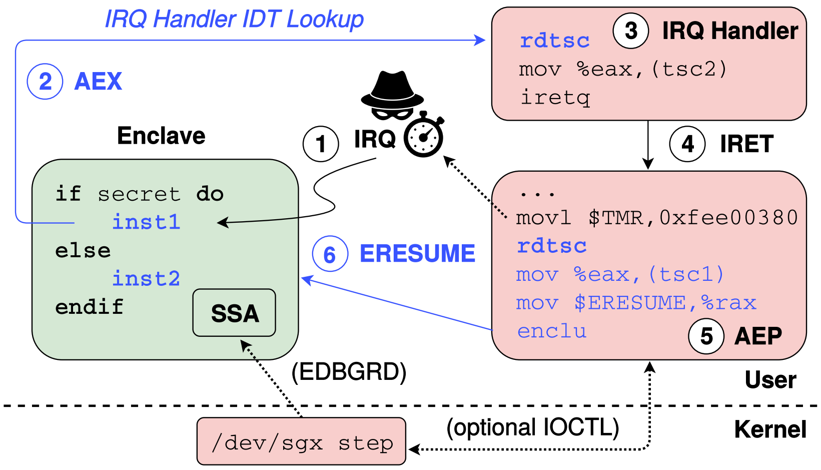


Figure 5.5: Enhanced SGX-Step framework for precise interrupt latency measurements (blue path) on Intel x86 platforms.

switches in the timer configuration path [92, 181, 156]. Carefully selecting a platform-specific timer interval ensures that interrupts reliably arrive with a very high probability ($> 97\%$) within the first enclaved instruction after `eresume` [257].

While SGX-Step allows APIC interrupts to be sent from a ring 3 user space process, the original framework still vectors to a conventional ring 0 kernel space interrupt handler. Execution will eventually return to the user space AEP stub where the single-stepping adversary collects side-channel information, and configures the local APIC timer for the next interrupt before resuming the enclave. This approach suffices to amplify conventional side channels, but subtle microarchitectural timing differences can be affected by noise from kernel space interrupt handling code, privilege level switches, and cache pollution [92, 181]. As such, precisely measuring interrupt latency on Intel x86 platforms presents a substantial challenge over state-of-the-art enclave execution control approaches. As an important contribution, we therefore extended SGX-Step to handle interrupts *completely* within user space, without ever having to vector to the kernel.

Figure 5.5 summarizes our improved approach to interrupt and resume enclaves. In an initial preparatory phase, the privileged adversary queries the `/dev/sgx-step` Linux kernel driver to establish user space virtual memory mappings for the local APIC MMIO range plus the IA-64 Interrupt Descriptor Table (IDT) [257, 114]. Custom user space ISRs can now be registered directly by writing to the relevant IDT entry, taking care to specify the handler address relative to the user code segment and with descriptor privilege level 3 [114].[6] When the local APIC timer interrupt ① arrives within an enclaved instruction, SGX's secure AEX microcode procedure stores and clears CPU registers inside the enclave. Next, the conventional interrupt logic takes over and ② vectors to the user space interrupt handler. At this point, ③ we immediately grab a timestamp as the very first ISR instruction before ④ returning to the aforementioned AEP stub. ⑤ Here, we log the extracted latency timing measurements, optionally annotating them for benchmark debug enclaves with the stored in-enclave program counter that can be retrieved via the privileged `edbgrd` instruction in the `/dev/sgx-step` driver. Thereafter, we configure the local APIC timer for the next interrupt by writing into the initial-count MMIO register, and grab another timestamp to mark the start of the interrupt latency measurement. We take care to ⑥ execute the `eresume` instruction immediately after storing the timestamp to memory. This ensures that the interrupt latency measurement path between the two timestamps (visualized in blue in Fig. 5.5) *only* includes *(i)* three unprotected instructions to store the first timestamp and resume the enclave, plus *(ii)* the enclaved instruction of interest, plus *(iii)* the AEX microcode procedure to vector to the untrusted interrupt handler.

Handling noise. In contrast to an embedded Sancus-enabled MSP430 CPU, microarchitectural optimizations found in modern x86 processors are known to cause non-constant instruction execution times [44, 43]. Conformant to our attacker model, and closely following previous SGX attacks [156, 258, 79, 29, 181, 92] our experimental setup attempts to reduce measurement noise to a minimum by leveraging some of the unique untrusted operating system adversary capabilities to increase execution time predictability: disable SMT and dynamic frequency scaling (C-states, SpeedStep, TurboBoost), and affinitize the enclave process to a dedicated CPU with Linux's `isolcpus` kernel parameter.

To compensate for the remaining measurement noise, we correlate IRQ latency observations from repeated enclaved executions over the same input, as is not uncommon practice in (SGX) side-channel research [181, 79, 29, 225, 156]. Specifically, we will show in Section 5.5 that the IRQ latency measurements extracted by our framework exhibit a normally distributed variance. As such, adversaries can rely on basic statistical analysis techniques (e.g., mean, median, standard deviation) to combine multiple IRQ latency observations into a representative overall trace of enclaved instruction timings. Our practical implementation uses a Python post-processing script to parse the raw measurements extracted by our framework for repeated enclaved executions.

[6] We register our user space handlers as an x86 trap gate, since otherwise the interrupt-enable flag (`rflags.if`) does not get restored upon interrupt return.

The resulting traces plot the median execution time (plus optionally a box plot describing the distribution) for each subsequent instruction in the enclaved execution.

Accurately aggregating IRQ latency measurements from repeated enclaved executions also presents another substantial challenge, however. That is, while SGX-Step guarantees that a victim enclave executes *at most* one instruction at a time, a relatively low fraction of the timer IRQs ($< 3\%$) still arrives within `eresume`—before an enclaved instruction is ever executed [257]. Such "zero-step" events are harmless in themselves, but should be filtered out in order to correctly associate repeated measurements for the same step (*i.e.*, instruction) in different enclave invocations. We therefore contribute a novel technique to deterministically recognize false zero-step interrupts by probing the "accessed" bit [114] in the unprotected page-table entry mapping the enclaved code page. Specifically, we experimentally verified that the CPU *only* sets the code PTE accessed bit when the enclave did indeed execute an instruction (*i.e.*, timer interrupt arrived *after* `eresume`). Merely clearing the PTE accessed bit for the relevant enclaved code page before sending the interrupt, and querying it afterwards thus suffices to filter out false zero-step observations and achieve noiseless single-stepping.

5.5 Evaluation

Our embedded scenarios were evaluated on a development version of Sancus, extended with the hardware-level secure interrupt mechanism described in Section 5.3.1. We interfaced the Sancus core with a Diligent PmodKYPD peripheral for the secure I/O application. All SGX experiments were conducted on an off-the-shelf Dell Inspiron 13 7359 laptop with a generic Linux v4.13.0 kernel on a Skylake dual-core Intel i7-6500U CPU running at 2.5 GHz. Custom BIOS and kernel parameters were described in the previous section.

5.5.1 Effectiveness on Sancus

To evaluate our attack against the MSP430 bootstrap loader software, we encapsulated the relevant password comparison routine `BSL430_unlock_BSL` in a protected Sancus enclave. Texas Instruments eliminated secret-dependent control flow entirely from BSL v3 onwards (with a bitwise `or` of the `xor` of each pair of bytes). To the best of our knowledge, vulnerable BSL versions are no longer distributed. We therefore based our implementation on the latest BSL v9, where we replaced the invulnerable, `xor`-based password comparison with

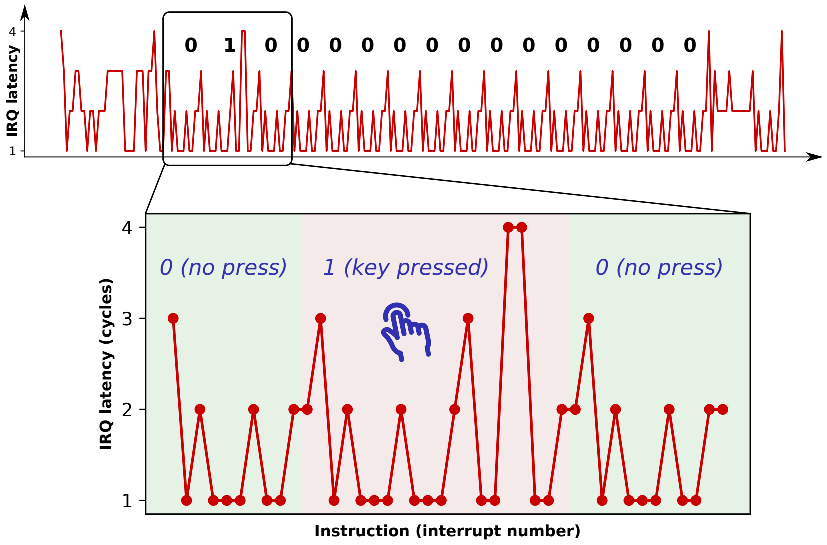


Figure 5.6: Interrupt latency trace revealing keystrokes in a Sancus enclave.

the hardened assembly code from Listing 5.1. The untrusted application context succeeds in recovering the full BSL password by iterating over all possible values for each input byte sequentially. A single interrupt per guess suffices to determine the correctness of the password byte under consideration. As such, our interrupt timing attack reduces an exhaustive search for the password from an exponential to a linear effort.

We provide the full source code of the `poll_keypad` function in Appendix C.2. The program was compiled with the Sancus C compiler based on LLVM/Clang v3.7.0. Our exploit recognizes all key presses without noise, in a single run of the victim enclave. This is an important property for I/O scenarios where, unlike cryptographic algorithms, a victim cannot be forced to execute the same code over the same secret data multiple times. Instead, key strokes should be recognized in real-time, while they are being entered by the human actor. Our online keypad attack only requires a single IRQ per loop iteration, totaling no more than 16 interrupts to recover the full key mask from a single enclaved execution.

Figure 5.6 visualizes the the interrupt latency signal and illustrates the maximal information leakage an adversary may collect when interrupting after every single instruction. This figure plots the execution timings of every individual instruction in one run of the Sancus keypad enclave. The resulting trace clearly reveals the full 16-bit key mask, where the adversary learns that key

"1" is currently pressed from a distinctly different interrupt latency behavior corresponding to the conditional control flow in Fig. 5.3.

5.5.2 SGX microbenchmarks

We first present microbenchmark experiments in order to quantify the effect of microarchitectural execution state and instruction type on the latency of individual x86 instructions. The microbenchmarks were obtained by single-stepping a benchmark SGX enclave that executes a slide of 10,000 identical assembly instructions. We refer to Chapter 4 for a thorough evaluation of SGX-Step's APIC timer-based single-stepping mechanism which guarantees that at most one enclaved instruction is executed per interrupt [257]. Additionally, we used the code PTE "accessed" bit technique described in Section 5.4.2 to deterministically filter out false zero-step observations, resulting in perfect single-stepping capabilities.

Differentiating instruction types. Figure 5.7 provides the IRQ latency distributions for selected processor instructions. The horizontal axis lists the observed latency timings in CPU cycles, whereas the number of corresponding interrupts in this latency class is depicted on the vertical axis. Note that the horizontal axis does not start from zero, as our interrupt latency measurement path (Fig. 5.5) includes the execution times of the `eresume` and AEX microcode.

As a first important result, we can decisively distinguish certain low-latency enclaved operations such as `nop` or `add` from higher-latency ones such as secure random number generation (`rdrand`) or certain floating point operations (`fscale`), solely by observing the latency they induce on interrupt. This confirms our hypothesis that IRQ latency on x86 platforms depends on the execution time of the interrupted instruction. Hence, these benchmarks can be considered clear evidence for the existence of a timing-based side channel in SGX's secure AEX procedure.

We can furthermore conclude that differentiating a `nop` instruction from an `add` with immediate and register operands is much less obvious, however. These instructions are indeed very similar at the microarchitectural level, both requiring only a single micro-op [66]. As an interesting special case, we investigated the IRQ latency behavior of the `lfence` instruction, which serializes all prior load-from-memory operations. This instruction has recently become particularly relevant, for Intel officially recommends [117] to insert `lfence` instructions after sensitive conditional branches to protect SGX enclaves against Spectre v1 speculative bounds check bypasses [146]. While the microarchitectural timing

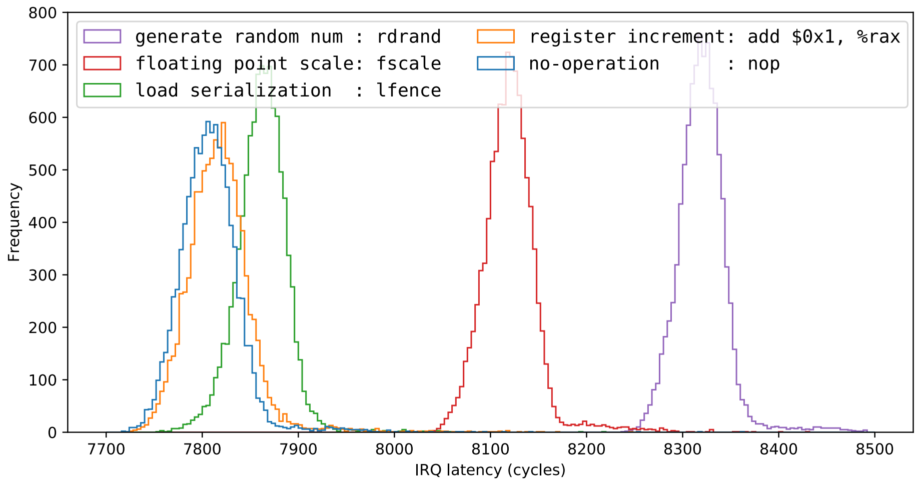


Figure 5.7: Interrupt latency distributions for selected x86 instructions, from left to right: `nop`, `add`, `lfence`, `fscale`, `rdrand`.

differences are more subtle, Fig. 5.7 still shows that one can on average plainly separate `lfence` from ordinary `nop` or `add` instructions.

Measuring data timing channels. Variable latency arithmetic instructions are known to be an exploitable side channel, even in code without secret-dependent control flow [43, 44, 15]. Previous research on microarchitectural data timing channels has established that the execution time of some commonly used x86 arithmetic instructions such as (floating point) multiplication or division depends on the operands they are being applied upon. Our second set of microbenchmark experiments therefore explore leakage of enclaved operand values through interrupt latency for the widely studied [43, 44, 15] unsigned integer division x86 `div` instruction.

Figure 5.8 shows the IRQ latency distributions for 10,000 enclaved executions of the `div` instruction applied on different 128-bit dividend operands and a fixed 64-bit divisor (`0xffffffffffffffff`). The average interrupt latency clearly increases as the dividend becomes larger, which confirms that "the throughput of `div/idiv` varies with the number of significant digits in the input `rdx:rax`" [113] As such, we conclude that IRQ latency leaks operand values for variable latency instructions. Importantly, in contrast to classical start-to-end timing measurements, Nemesis-style interrupt timing attacks leak this information at an *instruction-level* granularity, which allows to precisely

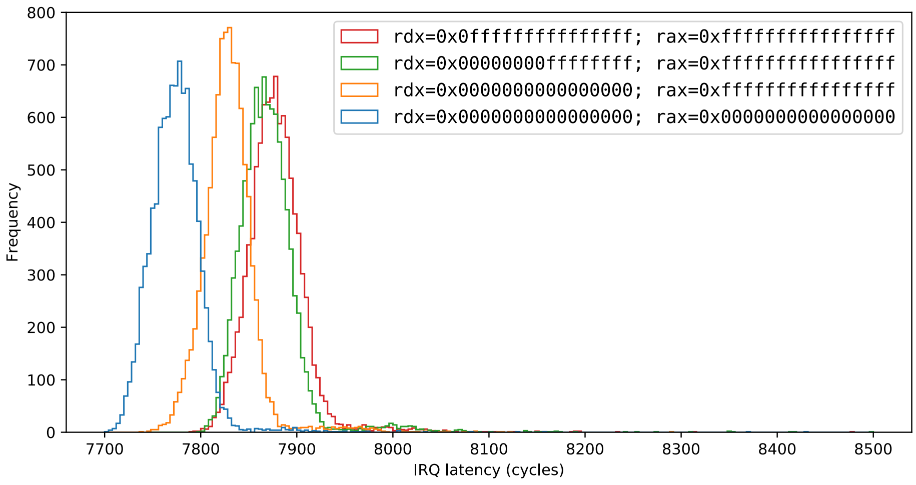


Figure 5.8: Data-dependent interrupt latencies for the x86 `div` instruction with from left to right an increasing amount of bits set in the dividend operand.

isolate (successive) data-dependent instruction timing measurements in a larger enclaved computation.

Influence of data caching. Figure 5.9 investigates the IRQ latency distributions for selected `mov` instructions to/from enclave memory. The store distribution is characterized by two prominent normally distributed peaks. Our hypothesis is that the right peak, representing measurements with a larger IRQ latency, is caused by a write miss in the data cache.[7] A write miss indeed forces the CPU to wait for completion of the memory transaction before finishing the instruction. It appears that in this particular experimental setup, the processor's cache replacement policy rather frequently evicts the data accessed by the benchmark enclave. To support this hypothesis, we examined IRQ latency behavior for the x86 `movnti` store operation with a non-temporal hint that forces the CPU to write the data directly into memory, without updating or fetching the corresponding cache line. `movnti` clearly manifests an increased latency that overlaps with the right peak of the store distribution.

[7]Intel SGX always uses a write-back caching policy for enclave memory [114]. This means that a write hit on enclave memory initially only updates the cache, unblocking the processor immediately, while writing to main memory is postponed until eviction of the dirty cache line. When the data was not yet in the cache (*i.e.*, write miss), however, any dirty line about to be replaced has to be written back, and the new line has to be fetched from main memory.

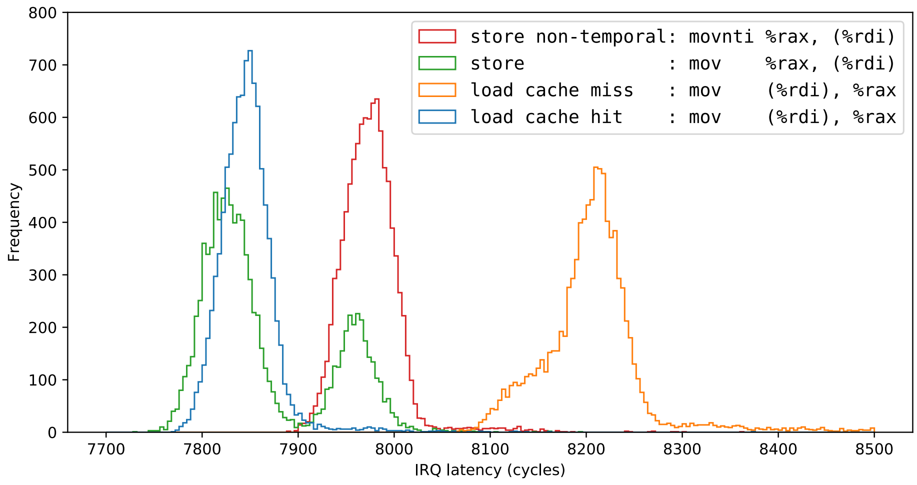


Figure 5.9: Increased interrupt latencies from enclaved data cache misses, from left to right: store, load hit, non-temporal store, load miss.

To investigate the impact of data cache misses on enclaved load operations, we instrumented the instruction slide in our benchmark enclave to explicitly invalidate the corresponding cache line by executing `clflush` before each `mov` instruction. Our noiseless single-stepping techniques allows to afterwards filter out latency measurements for the interleaved `clflush` instructions, such that the resulting IRQ latency distributions are solely characterized by the execution times of the `mov` instruction under consideration. Figure 5.9 shows a prominently increased latency for intra-enclave memory load operations that miss the data cache hierarchy. We suspect that the sparser distribution for load cache misses is caused by noise from the DRAM controller.

These experiments thus provide clear evidence for the fact that IRQ latency reveals cache misses. This finding may be particularly relevant for state-of-the-art fortified TEE designs like Sanctum [48] that include all known architectural countermeasures to prevent adversaries from gaining insight into enclave caching behavior.

Influence of address translation. SGX was explicitly designed to traverse untrusted page tables during enclaved execution, and verifies address translation metadata via an independent additional protection mechanism. Recent research on address translation side-channel attacks [258, 81], however, exploits the microarchitectural property that x86 page-table entries are cached as with

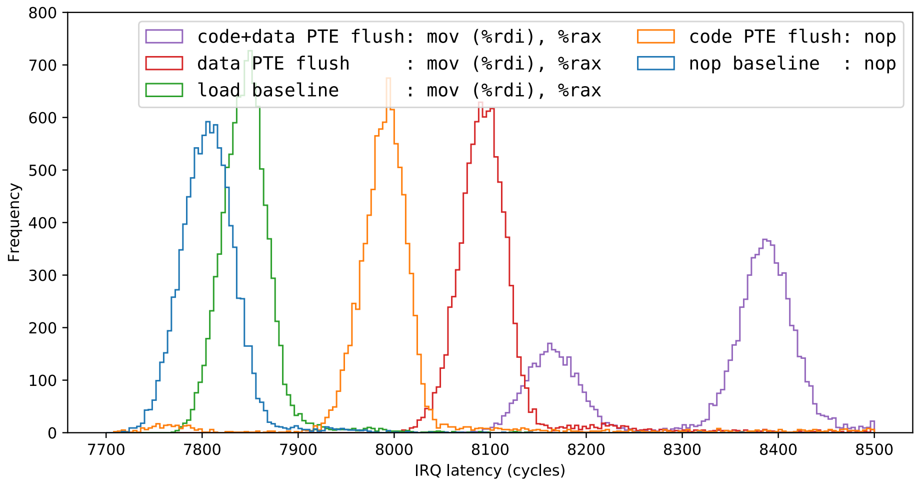


Figure 5.10: Increased interrupt latencies from cache misses in the page-miss handler, from left to right: `nop` baseline, `mov` baseline, `nop` code PTE miss, `mov` data PTE miss, `mov` code+data PTE miss.

normal data. By spying on unprotected cache lines, adversaries can gain insight into enclaved memory page accesses.

Our last set of microbenchmark experiments explores the impact of untrusted address translation data cache misses on the latency of the interrupted instruction. We used `clflush` before resuming the benchmark enclave to invalidate the cache line for the *unprotected* PTE entry that stores the physical address of the code page containing the microbenchmark instruction slide. Figure 5.10 demonstrates that we can distinctly increase the latency of even ordinary `nop` instructions in this way. Furthermore, for instructions with a memory operand, kernel-level adversaries can choose to flush the PTE entry for the data operand, and/or the enclaved code page to be executed. Figure 5.10 indeed shows a clear increase in IRQ latency for `mov` instructions that need an additional memory access to retrieve the physical address of the private data operand. Likewise, latency even further increases when also flushing the PTE entry of the enclaved code page containing the load instruction.

We conclude that IRQ latency reveals data cache misses in the page-table walk at instruction-level granularity. While SGX page tables reside in unprotected memory, this finding may once more impact fortified TEE designs [48, 58] that move page tables inside enclave memory, out of reach of the attacker, to protect against address translation side-channel attacks.

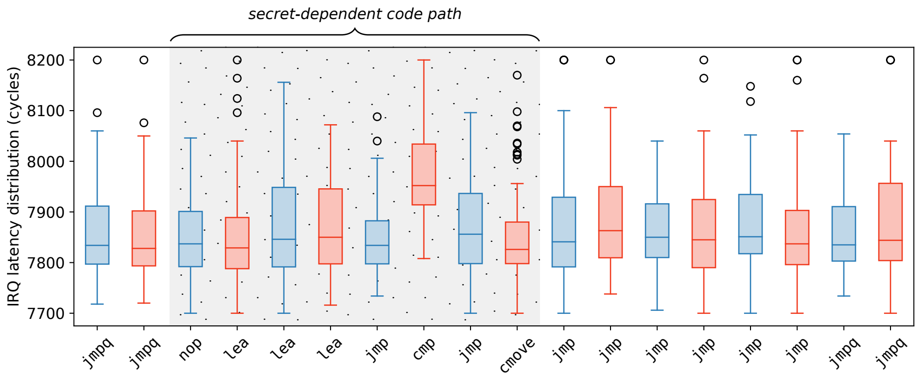


Figure 5.11: Interrupt latency distributions for 100 runs of Zigzagger branch taken (blue, left) vs. not-taken (red, right) execution paths.

5.5.3 SGX macrobenchmark attack scenarios

To demonstrate information leakage in larger enclave programs, we extracted full IRQ latency traces from the SGX case study applications introduced in Section 5.3.2. In contrast to the isolated microbenchmark experiments described above, our macrobenchmark results illustrate interrupt latency behavior in typical, compiler-generated mixed instruction streams.

Defeating Zigzagger. Since the Zigzagger compiler pass was not made publicly available, we copied the exemplary assembly code (Fig. 4.2 on page 111) from the corresponding paper [156] in an SGX enclave. As explained in Section 5.3.2, we made sure to manually align secret-dependent code to fit entirely within one 64-byte cache line. Figure 5.11 shows the IRQ latency distributions extracted by our framework for 100 repeated runs of a victim enclave that either takes the first Zigzagger-obfuscated branch (a=1; blue) or not (a=0; red). The leftmost two box plots visualize IRQ latency measurements for the indirect branch instruction `zz4` at the end of the Zigzagger trampoline, whereas the following two grouped box plots represent instruction latencies in the the conditional control flow path from either `b1` (blue, `nop`) or `b2` (red, `lea`) to the next secret-dependent jump at `zz4`. Note that both execution paths in the assembly code snippet of Fig. 4.2 already merge at `b2.j`, such that IRQ latency traces extracted from Zigzagger-hardened code only feature an extremely short secret-dependent sequence of 4 instructions, marked in Fig. 5.11.

A first important observation, in line with the microbenchmark results above, is that IRQ latency measurements are normally distributed such that we need

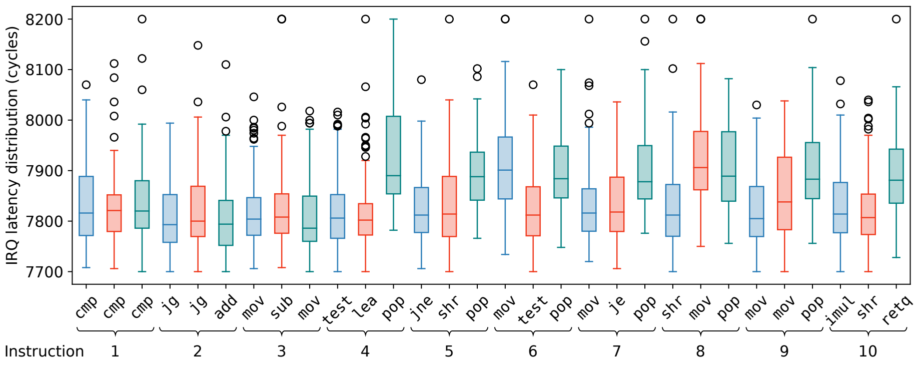


Figure 5.12: Interrupt latency distributions for 100 runs of `bsearch` left (blue) vs. right (red) vs. equal (green) execution paths.

to perform multiple observations before making decisive conclusions on the timing characteristics of the instruction under consideration. In this respect, the first two instructions in the secret-dependent execution paths exhibit similar and fairly indistinguishable IRQ latency distributions, which is indeed to be expected given that `nop` and `lea` instructions behave identical (micro-op count, latencies) at the microarchitectural level [66] The third secret-dependent instruction, either `jmp` or `cmp`, however, manifests a sharply visible (median) IRQ latency difference that can be exploited to unambiguously distinguish both branches. Specifically, by relying on the noiseless single-stepping technique from Section 5.4.2, adversaries can collect IRQ latencies from repeated enclaved executions, and afterwards categorize the samples for the third secret-dependent instruction as either a `jmp` or `cmp`. To compensate for outliers, we use the median IRQ latency instead of the mean. Note that Fig. 5.11 was generated from 100 repeated enclave invocations to yield a representative overall plot, but we found that in practice secret-dependent Zigzagger branches can already be reliably identified after as little as 10 enclave invocations. Finally, also note that there exists a subtle yet potentially exploitable IRQ latency distribution difference for the last secret-dependent instruction `jmp` vs. register `cmov`.

Inferring binary search indices. To evaluate our binary search attack, we constructed an enclave that calls the Intel SGX SDK `bsearch` trusted library function to look up a value in a fixed integer array. We carefully selected the exemplary lookup value to ensure that `bsearch` first looks left, then right, and finally returns the requested address. Our practical exploit faults on the code page containing the `bsearch` function to enter single-stepping mode and then starts collecting IRQ latency measurements. Figure 5.13 plots the

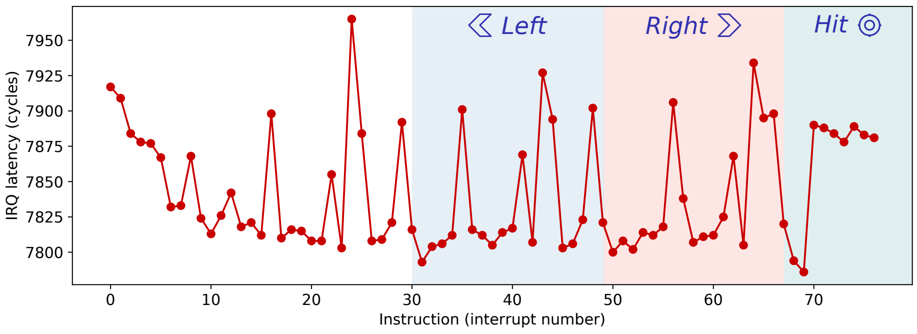


Figure 5.13: Median interrupt latencies over 100 `bsearch` invocations.

median IRQ latencies obtained from 100 enclaved `bsearch` executions, where each individual data point reveals the execution time of the corresponding assembly instruction in Appendix C.3. We annotated the trace to mark the three consecutive execution paths (left, right, equal) after comparing the value for that loop iteration. As with the Zigzagger benchmark, Fig. 5.12 furthermore compares relative IRQ latency distributions by means of box plots for each assembly instruction in the secret-dependent execution paths.

As a first important result, one can easily identify the relatively high-latency peaks from the 6 subsequent `pop` stack accessing instructions in the return path of the equal case (green instructions 4-10 in Fig. 5.12). Furthermore, while distinguishing the left (blue) and right (red) cases is more subtle, the source code in Listing 5.2 indicates that the right case has to perform slightly more work before continuing to the next loop iteration. This is indeed reflected at the assembly code level by two more low-latency register instructions (`sub` and `lea`) before the right branch continues along the common execution path. Again, we found this extremely subtle difference to be sufficient to distinguish both branches via the relative position of a higher-latency `mov` instruction at the start of the for loop. It is apparent from Fig. 5.13 that the IRQ latency patterns for the right branch are slightly shifted with respect to those of the left one. Particularly, the first high-latency peak in the left branch occurs 4 interrupts (instruction 6 in Fig. 5.12) after `cmp`, whereas for the right branch this peak only occurs after 6 interrupts (instruction 8 in Fig. 5.12). As with the Zigzagger benchmark, comparing median IRQ latency samples for specific instructions, identified by their single-stepping interrupt number, thus suffices to reliably infer control flow decisions in the binary search algorithm and establish the secret lookup key.

5.6 Discussion and mitigations

Interrupt timing leaks. While generally well-understood at the architectural level, asynchronous CPU events like faults and interrupts have not been studied extensively at the microarchitectural level. Recent developments on Meltdown-type "fault latency" attacks [162, 249] exposed fundamental flaws in the way modern out-of-order processors enforce software isolation, whereas Nemesis reveals a more intrinsic and subtle timing side channel in the CPU's interrupt mechanism. We showed that the act of interrupting enclaved execution leaks microarchitectural timing information at an instruction-level granularity, even on the most rudimentary of microcontrollers. In this, we have presented the first remotely exploitable controlled channel for embedded enclave processors, and we contribute to the understanding of SGX side-channel information leakage beyond the usual suspects.

Interrupt latency traces (e.g., Figs. 5.4, 5.6 and 5.13) can be regarded as an instruction-granular "x-ray" for enclaved execution. Our microbenchmark SGX experiments show that interrupt latency directly reveals certain high-latency enclaved operations, and can furthermore reliably quantify other microarchitectural properties that affect execution time on modern x86 processors [71], e.g., data-dependent instruction latencies, and data or page-table cache misses. In this respect, we expect that Nemesis's ability to extract fine-grained microarchitectural instruction timings from SGX enclaves will enable new and improved side channels such as MemJam-type [180] false dependency attacks. As a particularly relevant finding for fortified TEEs like Sanctum [48] that aim to eradicate known cache timing attacks, we identified what might well be one of the last remaining side channels that provide insight into enclave caching behavior. Specifically, since we have shown that interrupt latency reveals cache misses, we can see IRQ latency traces being leveraged in a trace-driven cache attack [1], for instance, to reduce the key space of cryptographic algorithms.

We have demonstrated that interrupt latency timing attacks pose a direct and serious threat to the protection model pursued by embedded TEEs such as Sancus, though further research is needed to investigate the bandwidth of practical Nemesis side-channel attacks on SGX platforms. A particularly promising future work avenue in this respect would be to supersede reverse engineering and statistical analysis efforts by applying automated machine learning techniques on IRQ latency traces extracted from multiple invocations of the victim enclave.

Why constant-time IRQ defenses are insufficient. We have shown how interrupt-capable adversaries can dissolve black box-style start-to-end protected execution times into (a sequence of) execution timing measurements for individual enclaved instructions. This chapter has focused on exploring "interrupt latency timing" channels on multi-cycle instruction set processors, but we want to stress that attack surface from secure interrupts is *not* limited to timing side channels only. Another potentially dangerous "interrupt counting" channel, for instance, would measure the total number of times the enclaved execution can be interrupted before it finally completes. For example, in the balanced BSL password comparison scenario of Listing 5.1, adversaries can interrupt the if branch twice (2 instructions), whereas the else branch featuring `nop` compensation code can be interrupted four times (4 instructions). As such, while the total enclaved execution time remains constant, interrupt-capable single-stepping adversaries will still notice a decrease in the total IRQ count for each correct password byte.

The above interrupt counting channel seems particularly interesting, for it only assumes a multi-cycle instruction set architecture, and thus continues to persist on processors with constant-time IRQ latency. We, for instance, considered a hardware patch for Sancus that always enforces the worst-case interrupt response time by inserting dummy execution cycles depending on the enclaved instruction being interrupted. Alternatively, ARM Cortex M0 processors [16] abandon multi-cycle instructions to handle any pending interrupt immediately. While such processors are immune to the IRQ latency timing attacks described in this chapter, they remain vulnerable to interrupt counting attacks and may additionally be exposed to advanced Nemesis-type interrupt timing attack variants.

We conclude that constant-time interrupt logic is a necessary but not sufficient condition to eradicate Nemesis-style interrupt attacks at the hardware level. In general processor-level solutions alone seem not to be able to completely prevent information leakage from secure interrupts in enclaved execution. This finding may have a consequential impact for fully abstract compilation schemes [203] and provably side-channel resistant processor designs [63, 64] that have so far not considered secure interrupt timing channels. We encourage further research and formal analysis to adequately address interrupt-based side channels via hardware-software co-design.

Application hardening. Considering that our attacks exploit secret-dependent control flow, an application-level solution should strive to eliminate conditional program branches and variable latency instructions completely. This can be realized by rewriting the enclave code manually (e.g., `xor`-based password comparison of Section 5.5.1), or by automated if-conversion in a compiler backend [44]. Such solutions remain compatible with existing TEE hardware, but also assume that sensitive information can be easily identified. Previous research [277, 29] in this area has shown that sensitive application data may be more ill-defined than the typical cryptographic keys of side-channel analysis. Moreover, if-conversion comes with a significant performance overhead [44] that somewhat invalidates the TEE promise of native code execution in a protected environment.

Alternatively, compilers could focus on detecting, rather than eliminating, IRQ timing attacks. Our interrupt extensions for Sancus indeed follow TEE designs [147, 28, 48] that explicitly call into an enclave to request resumption of internal execution. As such, Sancus enclaves are *interrupt-aware* and they could use excessive interrupt rates as an indicator to trigger some security policy that terminates the module and/or destroys secrets. Interrupts also occur in benign conditions, however, and a single interrupt already suffices to leak confidential information, as evident from our Sancus attack scenarios. Adversaries could thus adapt their attacks to the entry policy of a victim enclave.

Intel SGX on the other hand leaves enclave programs explicitly *interrupt-unaware* through the use of a dedicated `eresume` hardware instruction. However, a contemporary line of research [229, 42, 84] leverages hardware support for Transactional Synchronization Extensions (TSX) in recent x86 processors to detect interrupts or page faults in enclave mode. More specifically, these proposals rely on the property that code executing in a TSX transaction is aborted and automatically rolled back when an external interrupt request arrives. TSX furthermore modifies the stored in-enclave instruction pointer upon AEX, such that a preregistered transaction abort handler is called on the next `eresume` invocation. Whereas TSX-based defenses would likely recognize suspicious interrupt rates when single-stepping enclaved execution, advanced Nemesis adversaries could construct stealthy Sancus-like IRQ timing attacks that only interrupt the victim enclave minimally and stay under under the radar of the transaction abort handler's probabilistic security policy. Moreover, TSX-based defenses also suffer from some important limitations [257, 237], ranging from the absence of TSX features in some processors to severe runtime performance impact and the false positive/negative rates inherent to heuristic defenses. In conclusion, we do *not* regard current ad-hoc TSX approaches as a solution, even apart from compatibility and performance issues, since they cannot prevent the root information leakage cause. Our attacks against Sancus show that a *single* interrupt can deterministically leak sensitive information, and we expect further development of the attacks against SGX to increase stealthiness, as has been shown, for instance, for page-table based attacks [258, 263].

5.7 Related work

We have discussed TEE security architectures throughout this chapter. In this section we focus on relating our work to existing side-channel analysis research. There exists a vast body of work on microarchitectural timing channels [71], but side-channel attacks in a TEE context are only being explored very recently. To the best of our knowledge, we have presented the first remotely exploitable controlled channel for low-end embedded TEEs. Various authors [63, 158, 47] have explicitly expressed their concerns on software side-channel vulnerabilities in higher-end TEEs such as Intel SGX. This chapter argues, however, that current attack research efforts focus too narrowly on the "usual suspects" that are relatively well-known, and do not reveal anything really unexpected. Apart from our work, only page-table-based attacks [277, 230, 258, 263, 91] have to date been identified as a novel controlled channel. Compared to IRQ latency, the page fault channel has a coarser-grained granularity (instruction vs. page-level), but does not suffer from the noise inherent to microarchitectural channels.

Our attack vector is closely related to cache timing side channels in that IRQ latency traces reveal cache misses. A powerful class of access-driven cache attacks based on the PRIME+PROBE technique [202] first primes the cache by loading congruent addresses, and thereafter measures the time to reload these addresses so as to establish memory access patterns by the victim. Such PRIME+PROBE cache timing attacks have been successfully applied against SGX enclaves [79, 225, 29, 181, 92]. When memory is shared between the attacker and the victim, FLUSH+RELOAD [280] and FLUSH+FLUSH [87] techniques improve the efficiency of cache timing attacks. In the context of Intel SGX, these techniques have recently been leveraged to spy on unprotected page-table memory [258].

It has furthermore been shown that enclave-private control flow leaks via the CPU's branch prediction machinery [156, 59], which recently became particularly relevant for Spectre-type speculative execution attacks [146, 39]. Recent Intel microcode patches address Spectre attacks against SGX enclaves by clearing the BTB upon enclave entry/exit [39]. At the microarchitectural level, Nemesis-style interrupt latency timing attacks are more closely related to Meltdown [162] in that both abuse the property that asynchronous CPU events like faults and interrupts are only handled upon instruction retirement. While Intel SGX was initially considered to be resistant to Meltdown-type transient-execution vulnerabilities, recent work presented Foreshadow [249, 271], which allows for arbitrary in-enclave reads and completely collapses isolation and attestation guarantees in the SGX ecosystem. To allow for TCB recovery, Intel has revoked the compromised attestation keys, and released microcode patches to address Foreshadow at the hardware level.

5.8 Conclusion

The security aspects of asynchronous CPU events like interrupts and faults have not been amply studied from a microarchitectural perspective. We contributed Nemesis, a subtle timing channel in the CPU's rudimentary interrupt logic. Our work represents the first controlled-channel attack against embedded enclaved execution processors, and we demonstrated Nemesis's applicability on modern Intel SGX x86 platforms.

Chapter 6

Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution

This chapter was previously published as:

> J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx. "Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution". In: *27th USENIX Security Symposium*. Aug. 2018, pp. 991–1008

Preamble

This chapter presents Foreshadow, a novel transient-execution attack which for the first time decisively dismantled the security objectives of the Intel SGX ecosystem. At its core, Foreshadow exploits an incorrect transient forwarding effect similar to Meltdown, on top of which we develop an innovative exploitation methodology to reliably leak plaintext enclave secrets from the CPU cache. We demonstrate Foreshadow's disruptive impact by extracting full cryptographic keys from Intel's vetted architectural enclaves, allowing to launch rogue production enclaves and to forge arbitrary local and remote attestation responses. The extracted remote attestation keys affect millions of devices.

With Foreshadow, we were among the first to discover the security implications of transient execution in modern Intel processors. Specifically, this research was performed concurrently to Spectre and Meltdown, and we responsibly disclosed our findings to Intel on January 3, 2018, at which point we engaged in an extended embargo period until August 14, 2018. During the embargo period, and after our work had already been accepted at the USENIX Security conference, we were informed that another academic team developed a similar attack against SGX enclaves. This attack variant got later referred to by Intel as "enclave-to-enclave" (E2E) [107]. In the best interest of the scientific community, to avoid two concurrent papers on the same topic, we decided to invite the other researchers as co-authors and present our independent discoveries as the combined result of an international coalition. To raise awareness and disseminate our findings to the wider public, the website `https://foreshadowattack.eu/` was created.

Following our disclosure, Intel assigned CVE-2018-3615 to our findings and identified the root cause for Foreshadow to be an L1 Terminal Fault (L1TF) microarchitectural condition when accessing unmapped pages [107]. As it turned out, L1TF has much broader and more dire consequences than leaking enclave memory, for it essentially allows to dump the entire contents of the L1D cache, regardless of the owner of the data. In particular, Intel identified two closely related variants of our original Foreshadow attack, which we collectively refer to as "Foreshadow-NG"—Foreshadow Next Generation [271]. At a high level, Foreshadow-NG can be exploited by unprivileged applications for accessing physical memory out of the current process's address space (CVE-2018-3620), or by malicious guest virtual machines to access memory belonging to the hypervisor and other guest virtual machines (CVE-2018-3646). This chapter has been extended with a postscript in Section 6.9, which further elaborates on the microarchitectural root cause behind Foreshadow and reviews the mitigations that have been rolled out to protect SGX enclaves, operating system kernels, and hypervisors.

In the wider research landscape, Foreshadow led to important new insights and marked a paradigm shift in the way we should think about Meltdown-type threats. That is, in contrast to prior views, delayed exception handling is not a one-off bug to read kernel memory in Intel processors, but instead comprises an extensive and expanding class of Meltdown-type transient-execution attacks [162, 36, 223, 216, 35, 251]. Foreshadow directly contributed to the insight of differentiating Meltdown-type attacks in terms of the page-table bits used to trigger a page fault exception, which subsequently formed the basis for our more systematic transient-execution attack classification tree [36], summarized in

Appendix A. Furthermore, by transiently computing on unauthorized physical memory locations that are currently not mapped in the attacker's virtual address space, Foreshadow for the first time fully escaped the virtual memory sandbox. Merely unmapping secrets from an untrusted application's address space, as in the widely deployed kernel page-table isolation KAISER [85] mitigation to protect against the original Meltdown-US [162] attack, now becomes a necessary but not a sufficient condition.

6.1 Introduction

It becomes inherently difficult to place trust in modern, widely used operating systems and applications whose sizes can easily reach millions of lines of code, and where a single vulnerability can often lead to a complete collapse of all security guarantees. In response to these challenges, recent research [166, 189, 48] and industry efforts [10, 136, 176, 14] developed Trusted Execution Environments (TEEs) that feature an alternative, non-hierarchical protection model for isolated application compartments called *enclaves*. TEEs enforce the confidentiality and integrity of mutually distrusting enclaves with a minimal Trusted Computing Base (TCB) that includes only the processor package and microcode. Enclave-private CPU and memory state is exclusively accessible to the code running inside it, and remains explicitly out of reach of all other enclaves and software running at any privilege level, including a potentially malicious operating system and/or hypervisor. Besides strong memory isolation, TEEs typically offer an *attestation* primitive that allows local or remote stakeholders to cryptographically verify at runtime that a specific enclave has been loaded on a genuine (and hence presumed to be secure) TEE processor.

With the announcement of Intel's Software Guard Extensions (SGX) [176, 14, 114] in 2013, hardware-enforced TEE isolation and attestation guarantees are now available on off-the-shelf x86 processors. In light of the strong security guarantees promised by Intel SGX, industry actors are increasingly adopting this technology in a wide variety of applications featuring secure execution on adversary-controlled machines. Open Whisper Systems [169] relies on SGX for privacy-friendly contact discovery in its Signal network. Both Microsoft and IBM recently announced support for SGX in their cloud infrastructure. Various off-the-shelf Blu-ray players and initially also the 4K Netflix client furthermore use SGX to enforce Digital Rights Management (DRM) for high-resolution video streams. Emerging cryptocurrencies [179] and innovative blockchain technologies [111] rely even more critically on the correctness of Intel SGX.

Our contribution. This chapter shows that current SGX implementations cannot meet their security objectives. We present the Foreshadow attack, which leverages a speculative execution bug in recent Intel x86 processors to reliably leak plaintext enclave secrets from the CPU cache. At its core, Foreshadow abuses the same processor vulnerability as the recently announced Meltdown [162] attack, *i.e.*, a delicate race condition in the CPU's access control logic that allows an attacker to use the results of unauthorized memory accesses in transient out-of-order instructions before they are rolled back. Importantly, however, whereas Meltdown targets traditional hierarchical protection domains, Foreshadow considers a very different attacker model where the adversary's goal is not to read kernel memory from user space, but to compromise state-of-the-art *intra-address space* enclave protection domains that are not covered by recently deployed kernel page-table isolation defenses [85]. We explain how Foreshadow necessitates a novel exploitation methodology, and we show that our basic attack can be entirely mounted by an unprivileged adversary without root access to the victim machine. Given SGX's unique privileged attacker model, however, we additionally contribute a set of *optional* kernel-level optimization techniques to further reduce noise for root adversaries. Our findings have far-reaching consequences for the security model pursued by Intel SGX in that, in the absence of a microcode patch, current SGX processors cannot guarantee the confidentiality of enclaved data nor attest the integrity of enclaved execution, including for Intel's own architectural enclaves. Moreover, despite SGX's ambition to defend against strong kernel-level adversaries, present SGX processors cannot even safeguard enclave secrets in the presence of unprivileged user space attackers.

All previously known attacks against Intel SGX rely on application-specific information leakage from either side channels [277, 225, 258, 156, 257, 180, 118] or software vulnerabilities [266, 154]. It was generally believed that well-written enclaves could prevent information leakage by adhering to good coding practices, such as never branching on secrets, prompting Intel to state that "in general, these research papers do not demonstrate anything new or unexpected about the Intel SGX architecture. Preventing side channel attacks is a matter for the enclave developer" [133]. Foreshadow defeats this argument, however, as it relies solely on elementary Intel x86 CPU behavior and does *not* exploit any software vulnerability, or even require knowledge of the victim enclave's source code. We demonstrate this point by being the first to actually extract long-term platform launch and attestation keys from Intel's critical and thoroughly vetted architectural launch and quoting enclaves, decisively dismantling SGX's security objectives. In summary, our contributions are:

- We advance the understanding of Meltdown-type transient-execution CPU vulnerabilities by showing that they also apply to intra-address space isolation and SGX's non-terminating abort page semantics.

- We present novel exploitation methodologies that allow an unprivileged software-only attacker to reliably extract enclave secrets residing in either protected memory locations or CPU registers.
- We evaluate the effectiveness and bandwidth of the Foreshadow attack through controlled experiments.
- We extract full cryptographic keys from Intel's architectural enclaves, and demonstrate how to *(i)* bypass enclave launch control; and *(ii)* forge local and remote attestations to completely break confidentiality plus integrity guarantees for remote computations.

Current status. Following responsible disclosure practices, we notified Intel about our attacks in January 2018. Intel acknowledged the novelty and severity of Foreshadow-type "L1 Terminal Fault" attacks, and assigned CVE-2018-3615 to the results described in this chapter. We were further indicated that our attacks affect all SGX-enabled Core processors, while some Atom family processors with SGX support allegedly remain unaffected. At the time of this writing, Intel assigned CVSS severity ratings of "high" and "low" for respectively confidentiality and integrity. We note, however, that Foreshadow also affects the integrity of enclaved computations, since our attacks can arbitrarily modify sealed storage, and forge local and remote attestation responses.

Intel confirmed that microcode patches are underway and should be deployed concurrently to the public release of our results. As of this writing, however, we have not been provided with substantial technical information about these mitigations. We discuss defense strategies in Section 6.6, and provide further guidelines on the impact of our findings at `https://foreshadowattack.eu/`.

Disclosure. Foreshadow was independently and concurrently discovered by two teams. The KU Leuven authors discovered the vulnerability, independently developed the attack, and first notified Intel on January 3, 2018. Their work was done independently from and concurrently to other recent x86 speculative execution vulnerabilities, notably Meltdown and Spectre [162, 146]. The authors from Technion, University of Michigan, and the University of Adelaide independently discovered and reported the vulnerability to Intel during the embargo period on January 23, 2018.

6.2 Background

We first overview Intel SGX [176, 14, 114, 47] and refine the attacker model. Thereafter, we introduce the relevant parts of the x86 microarchitecture, and discuss previous research results on speculative execution vulnerabilities.

6.2.1 Intel SGX

Intel SGX extends the x86 instruction set architecture to allow for the secure execution of isolated *enclaves* in an untrusted environment. SGX relies on two important hardware-level building blocks: memory isolation and attestation.

Memory isolation. SGX enclaves live in the virtual address space of a conventional user mode process, but their physical memory isolation is strictly enforced in hardware. This separation of responsibilities ensures that enclave-private memory can never be accessed from outside, while untrusted system software remains in charge of enclave memory management (*i.e.*, allocation, eviction, and mapping of pages). An SGX-enabled CPU furthermore verifies the untrusted address translation process, and may signal a page fault when traversing the untrusted page tables, or when encountering rogue enclave memory mappings. Subsequent address translations are cached in the processor's Translation Lookaside Buffer (TLB), which is flushed whenever the enclave is entered/exited. Any attempt to directly access private pages from outside the enclave, on the other hand, results in abort page semantics: reads return the value -1 and writes are ignored.

SGX furthermore protects enclaves against motivated adversaries that exploit Rowhammer DRAM bugs, or resort to physical cold boot attacks. A hardware-level Memory Encryption Engine (MEE) [89] transparently safeguards the integrity, confidentiality, and freshness of enclaved code and data while residing outside of the processor package. That is, any access to main memory is first authenticated and decrypted before being brought as plaintext into the CPU cache.

Enclaves can only be entered through a few predefined entry points. The `eenter` and `eexit` instructions transfer control between the untrusted host application and an enclave. In case of a fault or external interrupt, the processor executes the Asynchronous Enclave Exit (AEX) procedure, which securely stores CPU register contents in a preallocated State Save Area (SSA) at an established location inside the interrupted enclave. AEX furthermore takes care of clearing CPU registers before transferring control to the untrusted operating system. A dedicated `eresume` instruction allows the unprotected application to re-enter a previously interrupted enclave, and restore the previously saved processor state from the SSA frame.

Enclave measurement. While an enclave is being built by untrusted system software, the processor composes a secure hash of the enclave's initial code and data. Besides this content-based identity (MRENCLAVE), each enclave also features an alternative, author-based identity (MRSIGNER) which includes a hash of the enclave developer's public key and version information. Upon enclave initialization, and before it can be entered, the processor verifies the enclave's signature and stores both MRENCLAVE and MRSIGNER measurements at a secure location, inaccessible to software—even from within the enclave. This ensures that an enclave's initial measurement is unforgeable, and can be attested to other parties, or used to access sealed secrets.

Each SGX-enabled processor is shipped with a platform master secret stored deep within the processor and exclusively accessible to key derivation hardware. To allow for TCB upgrades, and to protect against key wear-out, each key derivation request always takes into account the current CPU security version number and a random KEYID. Enclaves can make use of the key derivation facility by means of two SGX instructions: `ereport` and `egetkey`. The former creates a tagged local attestation report (including MRENCLAVE/MRSIGNER plus application-specific data) destined for another enclave. The target enclave, residing on the same platform, can use the `egetkey` instruction to derive a "report key" that can be used to verify the local attestation report. Successful verification effectively binds the application data to the reporting enclave, with a specified identity, which is executing untampered on the same platform. A secure, mutually authenticated cryptographic channel can be established by means of an application-level protocol that leverages the above local attestation hardware primitives.

Likewise, enclaves can invoke `egetkey` to generate "sealing keys" based on either the calling enclave's content-based or developer-based identity. Such sealing keys can be used to securely store persistent data outside the enclave, for later use by either the exact same enclave (MRENCLAVE) or the same developer (MRSIGNER).

Architectural enclaves. As certain policies are too complex to realize in hardware, some key SGX aspects are themselves implemented as Intel-signed enclaves. Specifically, Intel provides *(i)* a *launch enclave* that gets to decide which other enclaves can be run on the platform, *(ii)* a *provisioning enclave* to initially supply the long-term platform attestation key, and *(iii)* a *quoting enclave* that uses the asymmetric platform attestation key to sign local attestation reports for a remote stakeholder.

To regulate enclave development, Intel SGX distinguishes debug and production enclaves at creation time. The internal state of the former can be arbitrarily inspected and altered by means of dedicated debug instructions, such that only production enclaves boast SGX's full confidentiality and integrity commitment.

6.2.2 Attack model and objectives

Adversary capabilities. Whereas most existing SGX attacks require the full potential of a kernel-level attacker, we show that the basic Foreshadow attack can be entirely mounted from user space. Our attack essentially implies that current SGX implementations cannot even protect enclave secrets from *unprivileged* adversaries, for instance co-residing cloud tenants. Additionally, to further improve the success rate of our attack for *root* adversaries, we contribute various optional noise-reduction techniques that exploit full control over the untrusted operating system, in line with SGX's privileged attacker model.

Crucially, in contrast to all previously published SGX side-channel attacks [277, 225, 79, 180, 258, 156, 257] and existing Spectre-style speculative execution attacks [39, 197] against SGX enclaves, Foreshadow does *not* require any side-channel vulnerabilities, code gadgets, or even knowledge of the victim enclave's code. In particular, our attack is immune to all currently proposed side-channel mitigations for SGX [229, 42, 84, 237, 227, 40], as well as countermeasures for speculative execution attacks [128, 126]. In fact, as long as secrets reside in the enclave's address space, our attack does not even require the victim enclave's execution.

Breaking SGX confidentiality. The Intel SGX documentation unequivocally states that "enclave memory cannot be read or written from outside the enclave regardless of current privilege level and CPU mode (ring3/user-mode, ring0/kernel- mode, SMM, VMM, or another enclave)" [120]. As Foreshadow compromises confidentiality of production enclave memory, this security objective of Intel SGX is clearly broken.

Our basic attack requires enclave secrets to be residing in the L1 data cache. We show how unprivileged adversaries can preemptively or concurrently extract secrets as they are brought into the L1 data cache when executing the victim enclave. For root adversaries, we furthermore contribute an innovative technique that leverages SGX's paging instructions to prefetch arbitrary enclave memory into the L1 data cache without even requiring the victim enclave's cooperation.

When combined with a state-of-the-art enclave execution control framework, such as SGX-Step [257], our root attack can essentially dump the entire memory and register contents of a victim enclave at any point in its execution.

Breaking SGX sealing and attestation. The SGX design allows enclaves to "request a secure assertion from the platform of the enclave's identity [and] bind enclave ephemeral data to the assertion" [14]. While we cannot break integrity of enclaved data directly, we do leverage Foreshadow to extract enclave sealing and report keys. The former compromises the confidentiality and integrity of sealed secrets directly, whereas the latter can be used to forge false local attestation reports. Our attack on Intel's trusted quoting enclave for remote attestation furthermore completely collapses confidentiality plus integrity guarantees for remote computations and secret provisioning.

6.2.3 Microarchitectural x86 organization

Instruction pipeline. For a complex instruction set, such as Intel x86 [114, 47], individual instructions are first split into smaller micro-ops during the decode stage. Micro-operation decoding simplifies processor design: only actual micro-ops need to be implemented in hardware, not the entire rich instruction set. In addition it enables hardware vendors to patch processors when a flaw is found. In case of Intel SGX, this may lead to an increased CPU security version number.

Micro-operations furthermore enable superscalar processor optimization techniques stemming from a reduced instruction set philosophy. An execution pipeline improves throughput by parallelizing three main stages. First, a *fetch-decode* unit loads an instruction from main memory and translates it into the corresponding micro-op series. To minimize pipeline stalls from program branches, the processor's branch predictor will try to predict the outcome of conditional jumps when fetching the next instruction in the program stream. Secondly, individual micro-ops are scheduled to available *execution units*, which may be duplicated to further increase parallelism. To maximize the use of available execution units, simultaneous multithreading (Intel HyperThreading) technology can furthermore interleave the execution of multiple independent instruction streams from different logical processors executing on the same physical CPU core. Finally, during the instruction *retirement* stage, micro-op results are committed to the architecturally visible machine state (*i.e.*, register and memory contents).

Out-of-order and speculative execution. As an important optimization technique, the processor may choose to not execute sequential micro-operations as provided by the in-order instruction stream. Instead, micro-ops are executed *out-of-order*, as soon as the required execution unit plus any source operands become available. Following Tomasulo's algorithm [244], micro-ops are dynamically scheduled, e.g., using reservation stations, and await the availability of their input operands before they are executed. After completing micro-op execution, intermediate results are buffered, e.g., in a Reorder Buffer (ROB), and committed to architectural state only upon instruction retirement.

To yield correct architectural behavior, however, the processor should ensure that micro-ops are retired according to the intended in-order instruction stream. Out-of-order execution therefore necessitates a roll-back mechanism that flushes the pipeline and ROB to discard uncommitted micro-op results. Generally, such *speculatively* executed micro-ops are to be dropped by the CPU in two different scenarios. First, after realizing an execution path has been mispredicted by the branch predictor, the processor flushes micro-op results from the incorrect path and starts executing the correct execution path. Second, hardware exceptions and interrupts are guaranteed to be "always taken in the 'in-order' instruction stream" [114], which implies that all transient micro-op results originating from out-of-order instructions following the faulting instruction should be rolled-back as well.

CPU cache organization. To speed up repeated code and data memory accesses, modern Intel processors [114] feature a dedicated L1 and L2 cache per physical CPU (shared among logical SMT cores), plus a single last-level L3 cache shared among all physical cores. The unit of cache organization is called a *cache line* and measures 64 bytes. In multi-way, set-associative caches, a cache line is located by first using the lower bits of the (physical) memory address to locate the corresponding *cache set*, and thereafter using a tag to uniquely identify the desired cache line within that set.

Since CPU caches introduce a measurable timing difference for DRAM memory accesses, they have been studied extensively in side-channel analysis research [71].

6.2.4 Transient-execution attacks

The aforementioned in-order instruction retirement ensures functional correctness: the CPU's architectural state (memory and register file contents) shall be consistent with the intended program behavior. Nevertheless, the CPU's

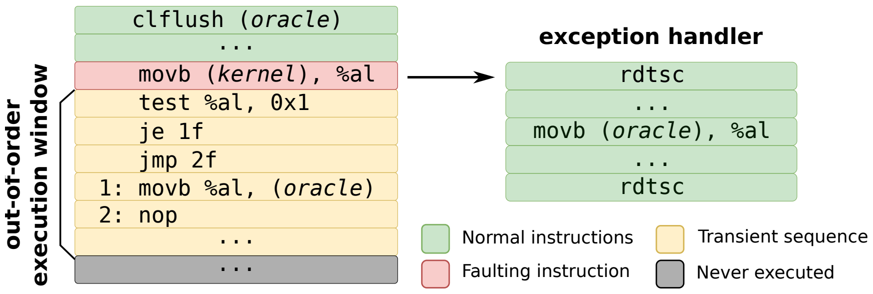


Figure 6.1: Rogue data cache loads can be leveraged to leak sensitive data from more privileged security layers.

microarchitectural state (e.g., internal caches) can still be affected by micro-ops that were speculatively executed and afterwards discarded. Recent concurrent research [146, 162, 101, 168, 67] on *transient-execution attacks* shows how an adversary can abuse such subtle microarchitectural side effects to breach memory isolation barriers.

A first type of Spectre [146] attacks exploit the CPU's branch prediction machinery to trick a victim protection domain into speculatively executing instructions out of its intended execution path. By "poisoning" the shared branch predictor resource, an attacker is able to steer the victim program's execution into transient instruction sequences that dereference memory locations the victim is authorized to access but the attacker not. A second type of attacks, including Meltdown [162] and Foreshadow, exploit a more crucial flaw in modern Intel processors. Namely, that there exists a small time window in which the results of unauthorized memory accesses are available to the out-of-order execution, before the processor issues a fault and rolls back any speculatively executed micro-ops. As such, Meltdown represents a critical race condition inside the CPU, which enables an attacker to transiently execute instructions that access unauthorized memory locations.

Essentially, transient execution allows an attacker to perform secret-dependent computations whose direct architectural effects are later discarded. In order to actually extract secrets, a "covert channel" should therefore be established to bring information into the architectural state. That is, the transient instructions have to deliberately alter the shared microarchitectural state so as to transfer/leak secret values. The CPU cache constitutes one such reliable covert channel; Meltdown-type vulnerabilities have therefore also been dubbed "rogue data cache loads" [101].

Figure 6.1 illustrates a toy example scenario where an attacker extracts one bit of information across privilege levels. In the first step, an attacker attempts to read data from a more privileged protection layer, eventually causing a fault to be issued and the execution of an exception handler. But, a small attack window exists where attackers can execute instructions based on the actual data read, and encode secrets in the CPU cache. The example uses a reliable FLUSH+RELOAD [280] covert channel, where the transient instruction sequence loads a predetermined "oracle" memory location into the cache, dependent on the least significant bit of the kernel data just read. When the processor catches up and eventually issues the fault, a previously registered user-level exception handler is called. This marks the beginning of the second step, where the adversary receives the secret bit by carefully measuring the amount of time it takes to reload the oracle memory slot.

6.3 The Foreshadow attack

In contrast to Meltdown [162], Foreshadow targets enclaves operating within an untrusted context. As such, adversaries have many more possibilities to execute the attack. However, as explained below and further explored in Appendix D, targeting enclaved execution also presents substantial challenges, for SGX's modified memory access and non-terminating fault semantics reflect extensive microarchitectural changes that affect transient execution.

We first present our basic approach for reading cached enclave secrets from the unprivileged host process, and thereafter elaborate on various optimization techniques to increase the bandwidth and success rate of our attack for unprivileged as well as root adversaries. Next, we explain how to reliably bring secrets in the L1 cache by executing the victim enclave. Particularly, we explain how to precisely interrupt enclaves and extract CPU register contents, and we introduce a stealthy Foreshadow attack variant that gathers secrets in real-time—without interrupting the victim enclave. We finally contribute an innovative kernel-level attack technique that brings secrets in the L1 cache without even executing the victim.

6.3.1 The basic Foreshadow attack

The basic Foreshadow attack extracts a single byte from an SGX enclave in three distinct phases, visualized in Fig. 6.2. As part of the attack preparation, the untrusted enclave host application should first allocate an "oracle buffer" ① of 256 slots, each measuring 4 KiB in size (in order to avoid false positives from unintentionally activating the processor's cache line prefetcher [113, 162]). In

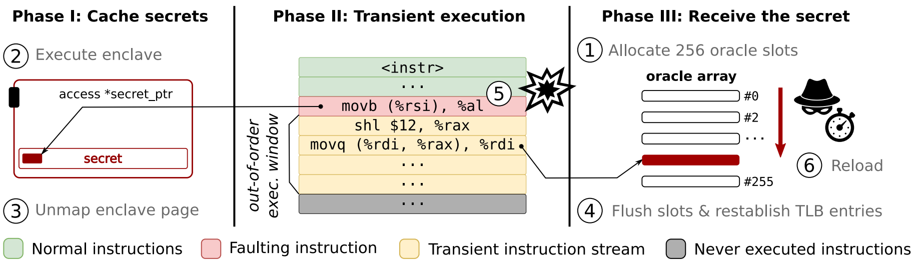


Figure 6.2: Basic overview of the Foreshadow attack to extract a single byte from an SGX enclave.

Phase I of the attack, plaintext enclave data is brought into the CPU cache. Next, Phase II dereferences the enclave secret and speculatively executes the transient instruction sequence, which loads a secret-dependent oracle buffer entry into the cache. Finally, Phase III acts as the receiving end of the FLUSH+RELOAD covert channel and reloads the oracle buffer slots to establish the secret byte.

Phase I: Caching enclave secrets. In contrast to previous research [162, 101, 67] on exploiting Meltdown-type vulnerabilities to read kernel memory, we found consistently that enclave secrets never reach the transient out-of-order execution stage in Phase II when they are not already residing in the L1 cache. A prerequisite for any successful transient extraction therefore is to bring enclave secrets into the L1 cache. As we noticed that the untrusted application cannot simply `prefetch` [86] enclave memory directly, the first phase of the basic Foreshadow attack executes the victim enclave ② in order to cache plaintext secrets. For now, we assume the secret we wish to extract resides in the L1 cache after the enclaved execution. We elaborate on this assumption in Sections 6.3.3 and 6.3.4 for interrupt-driven and SMT-based attacks respectively. Section 6.3.5 thereafter explains how root adversaries can bring secrets in the L1 cache without even executing the victim enclave.

Note that, while Meltdown has reportedly been successfully applied to read uncached kernel data directly from DRAM, Intel's official analysis report clarifies that "on some implementations such a speculative operation will only pass data on to subsequent operations if the data is resident in the lowest level data cache (L1)" [115]. We suspect that SGX's modified memory access semantics bring about fundamental differences at the microarchitectural level, such that the CPU's access control logic does not pass the results of unauthorized enclave memory loads unless they can be served from the L1 cache. Intel confirmed this hypothesis, officially referring to Foreshadow as an "L1 Terminal Fault" attack. We furthermore provide experimental evidence in Appendix D, showing that

Foreshadow can indeed transiently compute on kernel data in the L2 cache, but decisively *not* on enclave secrets residing in the L2 cache.

Regarding Intel SGX's hardware-level memory encryption [89], it should be noted that the MEE security perimeter encompasses the processor package, including the entire CPU cache hierarchy. That is, enclave secrets always reside as plaintext inside the caches and are only encrypted/decrypted as they move to/from DRAM. Practically, this means that transient instructions can in principle compute on plaintext enclave secrets as long as they are cached. As such, the MEE hardware unit does not impose any fundamental limitations on the Foreshadow attack, and is assuredly not the cause for the observation that we cannot read enclave secrets residing in the L2 cache.

Phase II: Transient execution. In the second phase, we dereference `secret_ptr` and execute the transient instruction sequence. In contrast to previous transient-execution attacks [162, 101, 67, 115] that result in a page fault after accessing kernel space, however, dereferencing unauthorized enclave memory does *not* produce a page fault. Instead, abort page semantics [120] apply and the data read is silently replaced with the dummy value -1. As such, in the absence of an exception, the race condition does not apply and any (transient) instructions following the rogue data fetch will never see the actual enclave secret, but rather the abort page value.

Foreshadow overcomes this challenge by taking advantage of previous research results on page-table-based enclaved execution attacks [277, 258]. Intel SGX implements an additional layer of hardware-enforced isolation on top of the legacy page-table-based virtual memory protection mechanism. That is, abort page semantics apply only *after* the legacy page-table permission check succeeded without issuing a page fault.[1] This property effectively enables the unprivileged host process to impose strictly more restrictive permissions on enclave memory, as illustrated in Fig. 6.3. In our running example, we proceed by revoking ③ all access permissions to the enclave page we wish to read:

```
mprotect( secret_ptr & ~0xfff, 0x1000, PROT_NONE );
```

We verified that the above `mprotect` system call simply clears the "present" bit in the corresponding page-table entry, such that any access to this page now

[1] Alternatively, as a result of SGX's additional EPCM checks [114], rogue virtual-to-physical mappings also result in page fault behavior *after* passing the address translation process. We experimentally verified that such faults can be successfully exploited by an attacker enclave that transiently dereferences a victim enclave's pages via a malicious memory mapping. Future mitigations (Section 6.6) should therefore decisively also take this microarchitectural exploitation path into account.

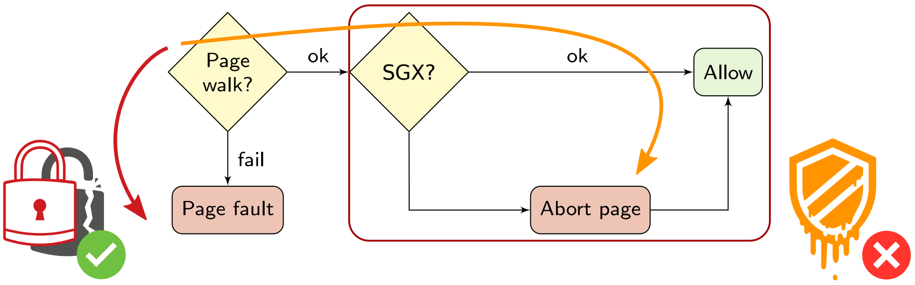


Figure 6.3: Naively applying Meltdown to transiently dereference SGX memory results in abort page semantics (right), whereas Foreshadow provokes a non-present fault in the untrusted page-table walk (left), before abort page semantics apply.

(eventually) leads to a fault. This observation yields an important side result, in that previous Meltdown attacks [162, 101, 67, 115] focused exclusively on reading kernel memory pages. Intel's analysis of speculative execution vulnerabilities hence explicitly mentions that rogue data cache loads only apply "to regions of memory designated supervisor-only by the page tables; not memory designated as not present" [115]. This is not in agreement with our findings.

As explained above, any enclave entry/exit event flushes the entire TLB on that logical processor. In our running example, this means that accessing the oracle slots in the transient execution will result in an expensive page-table walk. As this takes considerable time, the size of the attack window will be exceeded and no secrets can be communicated. Foreshadow overcomes this limitation by explicitly (re-)establishing ④ TLB entries for each oracle slot. In addition we need to ensure that none of the oracle slot entries are already present in the processor's cache. We achieve both requirements simultaneously by issuing a `clflush` instruction for all 256 oracle slots.

Finally, we execute ⑤ the transient instruction sequence displayed in Listing 6.1. When called with a pointer to the oracle buffer and `secret_ptr`, the secret value is read at line 5. As we made sure to mark the enclave page as not present, SGX's abort page semantics no longer apply and a fault will eventually be issued. However, the transient instructions at lines 6–7 will still be executed and compute the secret-dependent location of a slot v in the oracle buffer before fetching it from memory.

Phase III: Receiving the secret. Finally when the processor determines that it should not have speculatively executed the transient instructions, uncommitted register changes are discarded and a page fault is issued. After the fault is caught by the operating system, the attacker's user-level exception handler is

```
1 foreshadow:
2   # %rdi: oracle
3   # %rsi: secret_ptr
4
5   movb (%rsi), %al
6   shl $12, %rax
7   movq (%rdi, %rax), %rdi
8   retq
```

```
1 void foreshadow(
2    uint8_t *oracle,
3    uint8_t *secret_ptr)
4 {
5   uint8_t v = *secret_ptr;
6   v = v * 0x1000;
7   uint64_t o = oracle[v];
8 }
```

Listing 6.1: Transient instruction sequence to encode enclave secrets with Foreshadow: x86 assembly (left) and equivalent C code (right).

called. Here, she carefully measures ⑥ the timings to reload each oracle slot to establish the secret enclave byte. If the transient instruction sequence reached the execution at line 7, the oracle slot at the secret index now resides in the CPU cache and will experience a significantly shorter access time.

6.3.2 Reading full cache lines

The basic Foreshadow attack of the previous section leaks sensitive information while only leveraging the capabilities of a conventional user space attacker. But as SGX also aims to defend against kernel-level attackers, this section presents various optimization techniques, some of which assume root access (when indicated). In Section 6.4 we will show that these optimizations increase the bandwidth plus reliability of our attack, enabling us to extract complete cache lines from a single enclaved execution.

All of our optimization techniques share a common goal. Namely, increasing the likelihood that we do not destroy secrets as part of the measurement process. That is, an adversary executing Phases II and III of the basic Foreshadow attack should avoid inadvertently evicting enclave secrets that were originally brought into the L1 CPU cache during the enclaved execution in Phase I. We particularly found that repeated context switches and kernel code execution may unintentionally evict enclave secrets from the L1 cache. When this happens, the transient execution invariably loses the Meltdown race condition—effectively closing the attack window before the oracle slot is cached. Evicting enclave cache lines in this manner not only destroys the current measurement, but also eradicates the possibility to extract additional bytes belonging to the same cache line without executing the enclave again (Phase I). We therefore argue that minimizing cache pollution is crucial to successfully extract larger secrets from a single enclaved execution.

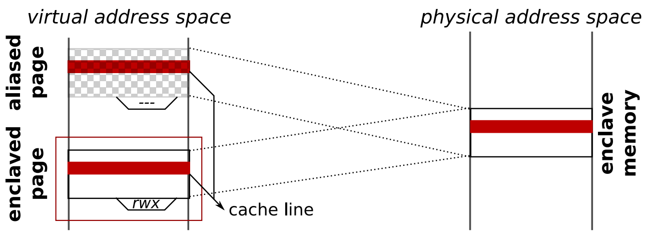


Figure 6.4: The physical enclave secret is mapped to an inaccessible virtual address for transient dereference.

Page aliasing (root). When untrusted code accesses enclave memory, abort page semantics apply and secrets do not reach the transient execution. The basic Foreshadow attack avoids this behavior by revoking all access rights from the enclave page through the `mprotect` interface. However, as enclaved execution also abides by page-table-based access restrictions [277, 258], these privileges can only be revoked *after* the enclave call returned. Unfortunately, we found that the `mprotect` system call exerts pressure on the processor's cache and may cause the enclave secret to be evicted from the L1 cache.

We propose an inventive "page aliasing" technique to avoid `mprotect` cache pollution for root adversaries. Figure 6.4 shows how our malicious kernel driver establishes an additional virtual-to-physical mapping for the physical enclave location holding the secret. As caches on modern Intel CPUs are physically tagged [114], memory accesses via the original or alias pages end up in the exact same cache lines. That is, the aliased page behaves similarly to the original enclaved page; only an additional page-table walk is required for address translation. We evade abort page semantics for the alias page in the same way as in the basic Foreshadow attack, by calling `mprotect` to clear the present bit in the page table. Importantly, however, we can now issue `mprotect` once in Phase I of the attack, *before* entering the enclave. For the aliased memory mapping is never referenced by the enclave itself.

Fault suppression. A second substantial source of cache pollution comes from the exception handling mechanism. Specifically, after executing the transient instruction sequence in Phase II of the attack, the processor delivers a page fault to the operating system kernel. Eventually the kernel transfers execution to our user-level exception handler, which receives the secret (Phase III). At this point, however, enclave secrets and/or oracle slots may have already been unintentionally evicted.

We leverage the Transactional Synchronization Extensions (TSX) included in modern Intel processors [114] to silently handle exceptions *within* the unprivileged attacker process. Previous research [229, 42, 237, 162] has exploited an interesting feature of Intel TSX. Namely, a page fault during transactional execution immediately calls the user-level transaction abort handler, without first signalling the fault to the operating system. We abuse this property to avoid unnecessary kernel context switches between Phases II and III of the Foreshadow attack by wrapping the entire transient instruction sequence of Listing 6.1 in a TSX transaction. While the transaction's write set is discarded, we did not notice any difference in the read set. That is, accessed oracle slots remain in the L1 cache.

Note that, while readily available on many processors, TSX is by no means the only fault suppression mechanism that attackers could leverage. Alternatively, as previously suggested [162, 101], the instruction dereferencing the secret could also be speculatively executed itself, behind a high-latency mispredicted branch. As a true hybrid between Spectre [146] and Meltdown [162], such a technique would deliberately mistrain the CPU's branch predictor to ensure that none of the instructions in Listing 6.1 are committed to the architecture, and hence no faults are raised.

Keeping secrets warm (root). Context switches to kernel space are not the only sources of cache pollution. In Phase III of the attack the access time to each oracle slot is carefully measured. As each slot is loaded into the cache, enclave secrets might get evicted from the L1 cache. To make matters worse, oracle slots are placed 4 KiB apart to avoid false positives from the cache line prefetcher [113]. All 256 oracle slots thus share the same L1 cache index and map to the same cache set.

We present two novel techniques to decrease pressure on cache sets containing enclave secrets. First, root adversaries can execute the privileged `wbinvd` instruction to flush the entire CPU cache hierarchy *before* executing the enclave (Phase I). This has the effect of making room in the cache, such that non-enclave accesses to the cache set holding a secret can be more likely accommodated in one of the vacant ways. Second, for unprivileged adversaries, instead of calling the transient-execution Phase II once, we execute it in a tight loop as part of the measurement process (Phase III). That is, by transiently accessing the enclave secret each time before we reload an oracle slot, we ensure the cache line holding the secret data remains "warm" and is less likely to be evicted by the CPU's least recently used cache replacement policy. Importantly, as both techniques are entirely implemented in the untrusted application runtime, we do *not* need to make additional calls to the enclave (Phase I).

Isolating cores (root). We found overall system load to be another significant source of cache pollution. Intel architectures typically feature an inclusive cache hierarchy: data residing in the L1 cache shall also be present in the L2 and L3 caches [114]. Unfortunately, maintaining this invariant may lead to unexpected cache evictions. When an enclaved cache line is evicted from the shared last-level L3 cache by another resource-intensive process, for instance, the processor is forced to also evict the enclave secret from the L1 cache. Likewise, since L1 and L2 caches are shared among logical processors, cache activity on one core might unintentionally evict enclave secrets on its sibling core.

In order to limit such effects, root adversaries can pin the victim enclave process to a specific core, and offload interrupts as much as possible to another physical core.

Dealing with zero bias. Consistent with concurrent work on Meltdown-type vulnerabilities [162, 101, 67, 168], we found that the processor zeroes out the result of unauthorized memory reads upon encountering an exception. When this nulling happens before the transient out-of-order instructions in Phase II can operate on the real secret, the attacker loses the internal race condition from the CPU's access control logic. This will show up as reading an all-zeroes value in Phase III. To counteract this zero bias, Foreshadow retries the transient-execution Phase II multiple times when receiving `0x00` in Phase III, before decisively concluding the secret byte was indeed zero.

Since Foreshadow's transient-execution phase critically relies on the enclave data being in the L1 cache, we consistently receive `0x00` bytes from the moment a secret cache line was evicted from the L1 cache. As such, the processor's nulling mechanism also enables us to reliably detect whether the targeted enclave data still lives in the L1 cache. That is, whether it still makes sense to proceed with Foreshadow cache line extraction or not.

6.3.3 Preemptively extracting secrets

As explained above, Foreshadow's transient-extraction Phase II critically relies on secrets brought into the L1 cache during the enclaved execution (Phase I). In the basic attack description, we assumed secrets are available after programmatically exiting the enclave, but this is often not the case in more realistic scenarios. Secrets might be explicitly overwritten, or evicted from the L1 cache by bringing in other data from other cache levels.

To improve Foreshadow's temporal resolution, we therefore asynchronously exit the enclave after a secret in memory was brought into the L1 cache, and before it is later overwritten/evicted. We first explain how root adversaries can combine Foreshadow with the state-of-the-art SGX-Step [257] enclave execution control framework to achieve a maximal temporal resolution: memory operands leak after every single instruction. Next, we re-iterate that even unprivileged adversaries can pause enclaves at a coarser-grained 4 KiB page fault granularity [277, 266] through the `mprotect` system call interface. Using this capability, we contribute a novel technique that allows unprivileged Foreshadow attackers to reliably inspect private CPU register contents of a preempted victim enclave.

Single-stepping enclaved execution (root). SGX prohibits obvious interference with production enclaves. Specifically, the processor ignores advanced x86 debug features, such as hardware breakpoints or the single-step trap flag (RFLAGS.TF) [114]. We therefore rely on the recently published open-source SGX-Step [257] framework to interrupt the victim enclave instruction per instruction.

SGX-Step comes with a Linux kernel driver to establish convenient user space virtual memory mappings for the local Advanced Programmable Interrupt Controller (APIC) device. A very precise single-stepping technique is achieved by configuring the APIC timer directly from user space, eliminating any noise from kernel context switches. Carefully selecting a platform-specific APIC timer interval ensures that the interrupt reliably arrives within the first instruction after `eresume`.

Dumping enclaved CPU registers. Section 6.2.1 explained how SGX securely stores the interrupted enclave's register contents in a preallocated SSA frame as part of the AEX microcode procedure. By targeting SSA enclave memory, a Foreshadow attacker can thus extract private CPU register contents. For this to work, however, the SSA frame data of interest should reside in the processor's L1 cache. The entire SSA frame measures multiple cache lines, with the general purpose register area alone already occupying 144 bytes (2.25 cache lines). These SSA cache lines could be unintentionally evicted as part of the kernel context switches needed to handle interrupts, or during Foreshadow's transient extraction Phases II and III.

We contribute an inventive way to reliably extract complete SSA frames. By revoking execute permissions on the victim enclave's code pages, the unprivileged application context can provoke a page fault on the first instruction after completing `eresume`. No enclaved instruction is actually executed, and register contents thus remain unmodified, but the entire SSA frame is re-filled and brought into the L1 cache as a side effect of the AEX procedure triggered by the page fault. We abuse such *zero-stepping* as an unlimited prefetch mechanism for bringing SSA data into the L1 cache. Before restoring execute permissions, a Foreshadow attacker reads the full SSA frame byte-per-byte, forcing the enclave to zero-step whenever an SSA cache line was evicted (*i.e.*, read all zero).

Together with a precise interrupt-driven or page fault-driven enclave execution control framework, our SSA prefetching technique allows for an accurate dump of the complete CPU register file as it changes over the course of the enclaved execution.

6.3.4 Concurrently extracting secrets

In modern Intel processors with SMT technology, the L1 cache is shared among multiple logical processors [114]. This property has recently been abused to mount stealthy SGX Prime+Probe L1 cache side-channel attacks entirely from a co-resident logical processor, without interrupting the victim enclave [79, 225, 29].

We explored such a stealthy Foreshadow attack mode by pinning a dedicated spy thread on the sibling logical core before entering the victim enclave. The spy thread repeatedly executes Foreshadow in a tight loop to try and read the secret of interest. As long as the secret is not brought into the L1 cache by the concurrently running enclave, the spy loses the CPU-internal race condition. This shows up as consistently reading a zero value. We use this observation to synchronize the spy thread. As long as a zero value is being read, the spy continues to transiently access the first byte of the secret. When the enclave finally touches the secret, it is at once extracted by the concurrent spy thread.

This approach has considerable disadvantages as compared to the above interrupt-driven attack variants. Specifically, we found that the bandwidth for concurrently extracting secrets is severely restricted, since each Foreshadow round needs 256 time-consuming Flush+Reload measurements in order to transfer one byte from the microarchitectural state (Phase II) to the architectural state (Phase III). As the enclave now continues to execute during the measurement process, secrets are more likely to be overwritten or evicted before being read by the attacker. Nonetheless, this stealthy Foreshadow attack variant should decidedly be taken into account when considering possible defense strategies in Section 6.6.

6.3.5 Reading uncached secrets

All attack techniques described thus far explicitly assume that the secret we wish to extract resides in the L1 cache after executing the victim enclave in Phase I of the attack. We now describe an innovative method to remove this assumption, allowing root adversaries to read any data located inside the victim's virtual memory range, including data that is *never* accessed by the victim enclave.

Managing the enclave page cache. The SGX design [176, 114] explicitly relies on untrusted system software for oversubscribing the limited protected physical memory Enclave Page Cache (EPC) resource. For this, untrusted operating systems can make use of the privileged `ewb` and `eldu` SGX instructions that respectively copy encrypted and integrity-protected 4 KiB enclave pages out of, and back into EPC.

We observed that, when decrypting and verifying an encrypted enclave page, the `eldu` instruction loads the entire page as plaintext into the CPU's L1 cache. Crucially, we experimentally verified that the `eldu` microcode implementation never evicts the page from the L1 cache, leaving the page's contents explicitly cached after the instruction terminates.

Dumping the entire enclave contents (root). We proceed as follows to extract the entire victim memory space. Going over all enclave pages (e.g., by inspecting `/proc/pid/maps`), our malicious kernel driver first uses `ewb` to evict the page from the EPC, only to immediately load it back using the `eldu` instruction. As `eldu` loads the page into the L1 cache and does not evict it afterwards, the basic Foreshadow attack described in Section 6.3.1 can reliably extract its content. Finally, the attack process is repeated for the next page of the victim enclave.

The above `eldu` technique dumps the entire address space of a victim enclave without requiring its cooperation. Since the initial memory contents is known to the adversary at enclave creation time, however, secrets are typically generated or brought in at runtime (e.g., through sealing or remote secret provisioning). As such, in practice, the victim should still be executed at least once, and the attacker could rely on a single-stepping primitive, such as SGX-Step [257], to precisely pause the enclave when it contains secrets, and before they are erased again.

Crucially, however, our `eldu` technique allows to extract secrets that are never brought into the L1 cache by the enclave code itself. As further discussed in Section 6.6, this attacker capability effectively rules out software-only mitigation strategies that force data to be directly stored in memory while deliberately evading the CPU cache hierarchy. For instance, by relying on explicit non-temporal write `movnti` instructions [114, 27].

6.4 Microbenchmark evaluation

We first present controlled microbenchmark experiments that assess the effectiveness of the basic Foreshadow attack and the various optimizations discussed earlier.

All experiments were conducted on publicly available, off-the-shelf Intel x86 hardware. We used a commodity Dell Optiplex 7040 desktop featuring a Skylake quad-core Intel i7-6700 CPU with a 32 KiB, 8-way L1 data cache.

Experimental setup. For benchmarks, we consider the capabilities of both root and unprivileged attackers, conformant to our threat model in Section 6.2.2. The *root adversary* has full access to the targeted system. She for example aims to attack DRM technology enforced by an enclave running on her own device. This enables her to use all the attack optimization techniques described in Section 6.3.2. In addition, she may reduce cache pollution by pinning the victim thread to as specific logical core and offloading peripheral device interrupts to another core.

The *unprivileged adversary*, on the other hand, is much more constrained and represents an attacker targeting a remote server. She gained code execution on the device, and targets an enclave running in the same address space, but did not manage to gain kernel-level privileges. Some attack optimizations, such as page aliasing or isolating workloads, can therefore not be applied.

We assess the effectiveness of Foreshadow by attacking a specially crafted benchmark enclave containing a 4 KiB memory page filled with randomized data. A dedicated entry point first loads 64 bytes of the secret page (*i.e.*, one full cache line) into the L1 cache. Upon `eexit`, we then extract all 64 bytes with Foreshadow, and finally verify their correctness. This process is repeated for all 64 cache lines within the 4 KiB page. To ensure representative measurements, we randomize both the targeted data locations and the enclave's load address. For this, we *(i)* randomly select 5 pages from a preallocated pool of 1024 enclaved pages per benchmark run, and *(ii)* combine the outputs of 200 runs of the benchmark process. In total 4,000 KiB of enclaved data was extracted for each attack scenario.

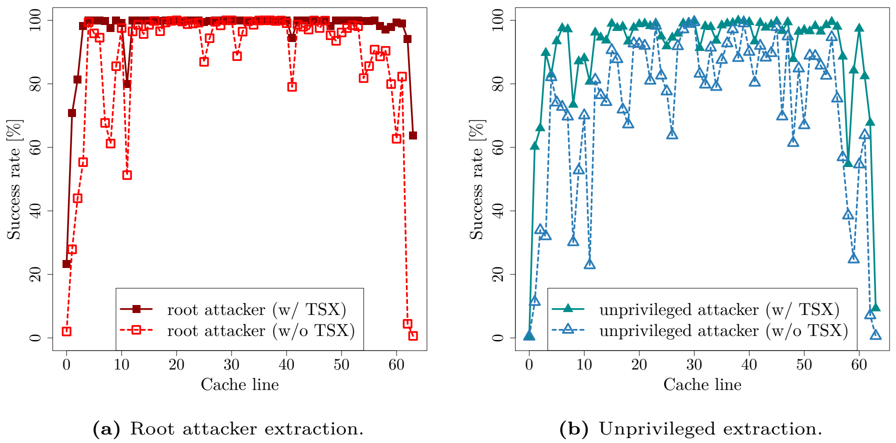


Figure 6.5: Success rates of the Foreshadow attack per cache line.

Success rates. Figure 6.5a displays the success rate for each cache line in the root attacker model. Overall, we reached an outstanding median success rate of 99.92% (with TSX). As not every SGX-capable machine supports TSX, we executed the same benchmark without relying on TSX features. This resulted in a moderate median success rate drop of 2.59 percentage points (97.32%).

Interestingly, the cache lines storing data at the beginning/end of the targeted page (*i.e.*, cache lines `#0` and `#63`) manifest a distinctly lower average success rate: respectively 23.25/2.03% and 63.78/0.63% with and without TSX. We attribute this effect to unintended L1 cache line evictions from *(i)* the remaining enclaved execution after loading the secret into the cache (e.g., `eexit`); and *(ii)* our own attack measurement code (e.g., probing of the oracle buffer in Phase III). Specifically, upon closer inspection, we found that recent interrupt-driven SGX cache attacks [181, 92] explicitly report similar lowered success rates for the first and last cache lines, attributed to asynchronous enclave exit and kernel context switches. Note that we consider the increased cache pressure on the first and last cache lines only a nonessential limitation of our current attack framework, however, and decisively *not* an avenue to defend against improved Foreshadow attacks.

Figure 6.5b displays the result of the same benchmark for an unprivileged attacker. As expected, the median success rate drops reasonably to 96.82% and 81.14% with and without TSX respectively. While these rates are somewhat lower, they distinctly show that even much more restrained user-level adversaries can successfully attack SGX enclaves with an impressive success rate.

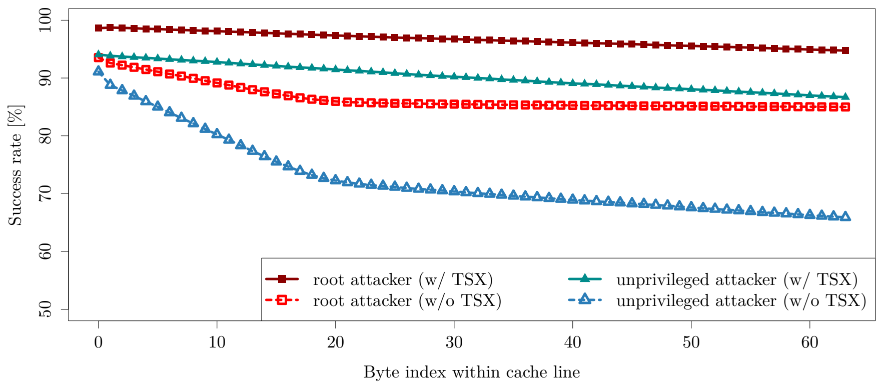


Figure 6.6: Success rate of the Foreshadow attack per byte within a cache line.

It is crucial for the Foreshadow attack to succeed that the cache line holding the secret remains in the L1 cache. We found that the likelihood of inadvertently evicting secrets from the L1 cache increases with each byte extracted within a cache line. Figure 6.6 quantifies this *intra-cache line degradation* behavior. For the root adversary, the probability of successfully extracting the first byte within a cache line is 98.61%. By the time the last last byte of the cache line is extracted, however, the success rate has degraded to 94.75%. Especially the use of TSX shows to play a large role here. An unprivileged TSX attacker can limit intra-cache line degradation from 94.05% to 86.68%. This outperforms even all other optimization mechanisms for the root adversary without TSX (93.53% - 84.99%).

6.5 Attacking Intel architectural enclaves

While SGX is largely realized in hardware and microcode, Intel implemented certain critical functionality in software through dedicated "architectural enclaves". These enclaves are part of the TCB, and were written by experts with detailed knowledge of the security architecture. No obvious security flaws [266, 154] have ever been found, and Intel's architectural enclaves additionally implement various defense in-depth mechanisms. For example, even though private memory should never leak from enclaves, sensitive data gets overwritten as soon as possible.

To the best of our knowledge, we are the first to present full key extraction attacks against Intel's vetted architectural enclaves. To date only one subtle side-channel vulnerability [50] has been identified in Intel's quoting enclave, which only affects secondary privacy concerns and assuredly does not invalidate remote attestation guarantees. This shows that Foreshadow is substantially more powerful than previous enclaved execution attacks that rely on either side channels or memory-safety bugs.

Note that, for maximum reliability, both our attacks against Intel's architectural launch and quoting enclaves assume the root adversary model, and apply all of the optimization techniques described in Section 6.3.2. Since our final exploits do *not* need to resort to the single-stepping or `eldu` prefetching root-only techniques of Sections 6.3.3 and 6.3.5, however, we expect they could be further improved to run entirely with user space privileges.

6.5.1 Attacking the Intel launch enclave

Background. SGX enclaves are created in a multi-stage process performed by untrusted system software. Before the enclave can be initialized through the `einit` instruction, a valid EINITTOKEN needs to be retrieved from the Intel Launch Enclave (LE). Essentially, such a token contains the target enclave's content-based (MRENCLAVE) and author-based (MRSIGNER) identities, requested features and attributes, plus a random KEYID. A Message Authentication Code (MAC) over the token data furthermore safeguards integrity, such that EINITTOKENs can be freely passed around by untrusted software.

As with local attestation (Section 6.2.1), the security of this scheme ultimately relies on a processor-level secret accessible to both LE and `einit`. We refer to this secret as the platform *launch key.* The `einit` instruction derives the 128-bit launch key to verify the correctness of the provided EINITTOKEN, and takes care to only initialize enclaves whose identities and attributes match the ones in the token. In order to bootstrap initialization for the LE itself, Intel's MRSIGNER value is hard-coded in the processor and used by `einit` to skip the EINITTOKEN check and grant access to the launch key. This ensures that only an Intel-signed LE can invoke `egetkey` to derive the launch key needed to compute valid MACs.

Intel uses the above enclave launch control scheme to impose a strict, software-defined enclave attribute control policy. More specifically, current LE implementations enforce that *(i)* either the enclave debug attribute is set or `mrsigner` is white-listed by Intel; and *(ii)* the enclave does not feature privileged, Intel-only attributes, such as access to the long-term platform provisioning key.

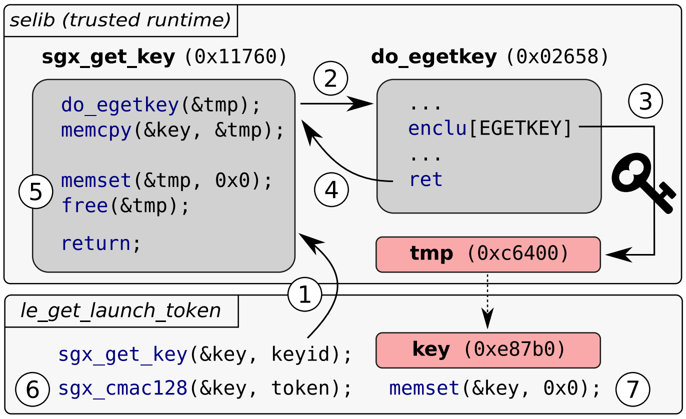


Figure 6.7: Key derivation in the SGX Launch Enclave.

Attack and exploitation. Our goal is to extract a full 128-bit launch key from a single LE execution. This is necessary, for each `egetkey` derivation (Section 6.2.1) includes a random 256-bit KEYID, which is securely generated inside the enclave, such that each LE invocation uses a different launch key. We can therefore *not* correlate partial key recoveries from repeated launch enclave executions to extract a full key, as is common practice in side-channel research [181, 79, 29, 225, 156].

Intel's official LE image[2] features an entry point to create a tagged EINITTOKEN based on the provided target enclave measurements and attributes. This process is illustrated in Fig. 6.7. LE first generates a random KEYID and calls ① the `sgx_get_key` function to obtain the launch key. For this, the trusted in-enclave runtime allocates a temporary buffer, before calling ② a small `do_egetkey` assembly stub that executes the `egetkey` instruction to derive ③ the actual launch key. Next, the temporary buffer is copied ④ into a caller-provided buffer; and ⑤ overwritten plus deallocated before returning. LE now uses the launch key to compute ⑥ the required MAC, and immediately afterwards zeroes out ⑦ the key buffer.

An attacker can get hold of the launch key by targeting either the short-lived `tmp` buffer, or the longer-lived `key` buffer. Our exploit targets the more challenging `tmp` buffer to demonstrate Foreshadow's strength in combination with state-of-the-art enclave execution control frameworks [257, 277]. In the exploratory (offline) phase of the attack, we single-step LE and dump register

[2] `libsgx_le.signed.so` from Intel SGX Linux SDK v2.0 with product ID `0x20` and security version number `0x01`.

content (see Section 6.3.3) so as to easily establish the deterministic `tmp` address, plus any code locations of interest.[3] Next, in the online phase of the attack, we interrupt the victim enclave between steps ③ and ④ above, and instruct Foreshadow to extract the cache line containing the 128-bit key. We rely on page fault sequences [277] here to avoid any noise from timing-based interrupts, and to minimize the number of AEXs induced by our exploit. Specifically, we constructed a small finite state machine that alternately revokes access to either the `sgx_get_key` or `do_egetkey` code page. Merely counting page faults now suffices to deterministically locate the return instruction ④ in `do_egetkey`. At this point, the launch key resides in the L1 cache and can thus be reliably extracted by Foreshadow. We observed a 100% success rate in practice; that is, our final (online) exploit extracts the full 128-bit key without noise, from a single LE run with only 13 page faults in total—without resorting to the single-stepping or `eldu` prefetching techniques of Sections 6.3.3 and 6.3.5.

To validate the correctness of the extracted keys, we integrated a rogue launch token provider service into the untrusted runtime of the SGX SDK. The rogue launch token provider transparently creates tagged EINITTOKENs using a previously extracted key, and includes the corresponding (non-secret) KEYID, such that `einit` derives an identical launch key from the platform master secret. Obtaining a single LE key thus suffices to launch arbitrarily many rogue production enclaves on the same platform.

Impact. Bypassing Intel's controversial [47] launch control policy allows one to create arbitrary production enclaves without going through a license agreement process. Removing control over which enclaves can be run is a clear breach of Intel's licensing interests, but by itself has limited impact on SGX's security objectives. We are *not* able to fabricate enclaves. Any properly implemented key derivation in an enclave will depend on either the MRENCLAVE or MRSIGNER values (Section 6.2.1). Neither can be forged as they rely on cryptographic properties of SHA-256 and the signer's private key respectively. The ability to create rogue production enclaves could be abused for hiding malware [225], but does not provide an enclave writer with any substantial advantage.

There is one notable exception, related to CPU tracking privacy concerns [47]. Specifically, an attacker can now create enclaves with the ability to derive a "provisioning key" that remains constant as a processor changes owners. LE should make sure that only Intel-signed enclaves can derive such keys, needed for securing long-term remote attestation keys (Section 6.5.2). All other `egetkey` derivations include an internal OWNEREPOCH register, which can be

[3] Some reverse engineering is required for all symbols were stripped from the signed LE image.

re-randomized when a user sells her platform. This ensures that any remaining secrets are approvedly destroyed when a computer changes owners [14]. Note that provisioning key derivations do include MRSIGNER, however, such that we cannot derive Intel's provisioning key without access to Intel's private enclave signing key.

6.5.2 Attacking the Intel quoting enclave

Background. Section 6.2.1 introduced local, intra-platform attestation through the `ereport` instruction. Such tagged local attestation reports are useless to a remote stakeholder, however, as they can only be verified by a target enclave executing on the *same* platform. The Intel SGX design therefore includes a trusted Quoting Enclave (QE) [14, 47] to validate local attestation reports, and sign them with an asymmetric private key. The resulting signed attestation report, or *quote*, can now be verified by a remote party via the corresponding public key.

Intel imposes itself as a trusted third party in the attestation process. To address privacy concerns, QE implements Intel's Enhanced Privacy Identifier (EPID) [134] group signature scheme. An EPID group covers millions of CPUs of the same type (e.g., core i3, i5, i7) and security version number. In fully anonymous mode, the cryptosystem ensures that remote parties can verify quotes from genuine SGX-enabled platforms, without being able to track individual CPUs within a group or recognize previously verified platforms. In pseudonymous mode, on the other hand, remote verifiers can link different quotes from the same platform.

Figure 6.8 outlines the complete SGX remote attestation procedure. In an initial platform configuration phase Ⓐ, Intel deploys a dedicated Provisioning Enclave (PE) to request an EPID private key, from here on referred to as the platform *attestation key*, from the remote Intel Provisioning Service. Upon receiving the attestation key, PE derives an author-based *provisioning seal key* in order to securely store Ⓑ the long-term attestation key on untrusted storage. For a successful enclave attestation, the remote verifier issues ① a challenge, and the enclave executes ② the `ereport` instruction to bind the challenge to its identity. The untrusted application context now forwards ③ the local attestation report to QE, which derives ④ its report key to validate the report's integrity. Next, QE derives the provisioning seal key to decrypt ⑤ the platform attestation key received from system software. QE signs ⑥ the local attestation report to convert it into a quote. Upon receiving the attestation response, the remote verifier finally submits ⑦ the quote to Intel's Attestation Service for verification using the EPID group public key.

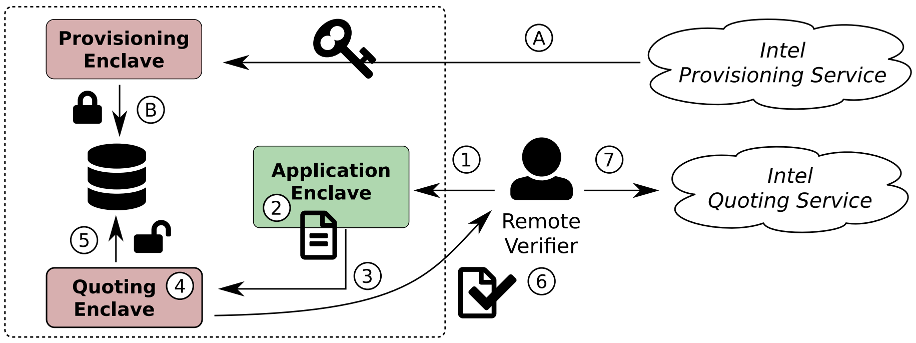


Figure 6.8: SGX Quoting Enclave for remote attestation.

Attack and exploitation. Remote attestation, as implemented by the SGX Quoting Enclave[4], relies on two pillars. First, QE relies on the infallibility of SGX's local attestation mechanism. An attacker getting hold of QE's report key can make QE sign arbitrary enclave measurements, effectively turning QE into a signing oracle. Second, QE relies on SGX's sealing mechanism to securely store the asymmetric attestation key. Should the platform provisioning seal key leak, an attacker can get hold of the long-term attestation key and directly sign rogue enclave reports herself. We exploited both options to show how Foreshadow can adaptively dismantle different SGX primitives.

As with the LE attack, illustrated in Fig. 6.7, both our QE key extraction exploits target the `sgx_get_key` trusted runtime function. We again constructed a carefully crafted page fault state machine to deterministically preempt the QE execution between the `egetkey` invocation and the key buffer being overwritten. As with the LE exploit, our final attack does *not* rely on advanced single-stepping or `eldu` prefetching techniques, and achieves a 100% success rate in practice. That is, our exploit reliably extracts the full 128-bit report and provisioning seal keys from a single QE run suffering 14 page faults in total.

We validated the correctness of the extracted keys by fabricating bogus local attestation reports, using a previously extracted QE report key, and successfully ordering the genuine Intel QE to sign them. Alternatively, we created a rogue quoting service that uses the leaked platform provisioning seal key to get hold of the long-term attestation key for signing. This allows an attacker to fabricate arbitrary remote attestation responses directly, without even executing QE on the victim platform.

[4] `libsgx_qe.signed.so` from Intel SGX Linux SDK v2.0 with product ID `0x01` and security version number `0x05`.

Impact. The ability to spoof remote attestation responses has profound consequences. Attestation is typically the first step to establish a secure communication channel, e.g., via an authenticated Diffie-Hellman key exchange [14]. Using our rogue quoting service, a network-level adversary (e.g., the untrusted host application) can trivially establish a man-in-the-middle position to read plus modify all traffic between a victim enclave and a remote party. All remotely provisioned secrets can now be intercepted, without even executing the victim enclave or requiring detailed knowledge of its internals—effectively rendering SGX-based DRM or privacy-preserving analytics [169, 179] applications useless. Apart from such confidentiality concerns, adversaries can furthermore fabricate arbitrary remote SGX computation results. This observation rules out transparent, integrity-only enclaved execution paradigms [245], and directly threatens an emerging ecosystem of untrusted cloud environments [20] and innovative blockchain technologies [111].

Intel's EPID [134] group signature scheme implemented by QE makes matters even worse. That is, in fully anonymous mode, obtaining a single EPID private key suffices to forge signatures for the *entire* group containing millions of SGX-capable Intel CPUs. Alarmingly, this allows us to use the platform attestation key extracted from our lab machine to forge anonymous attestations for enclaves running on remote platforms we don't even have code execution on. This does fortunately not hold for the officially recommended [134] pseudonymous mode, however, as remote stakeholders would recognize our fabricated quotes as coming from a different platform.

6.6 Discussion and mitigations

Impact of our findings. Concurrent research on transient-execution attacks [146, 162, 101, 168, 67] revealed fundamental flaws in the way current CPUs implement speculative out-of-order execution. So far, the focus of these attacks has been on breaching traditional kernel-level memory isolation barriers from an unprivileged user space process. Our work shows, however, that Meltdown-type CPU vulnerabilities also apply to non-hierarchical intra-address space isolation, as provided by modern Intel x86 SGX technology. This finding has profound consequences for the development of adequate defenses. The widely-deployed software-only KAISER [85] defense falls short of protecting enclave programs against Foreshadow adversaries. Indeed, page-table isolation mitigations are ruled out, for SGX explicitly distrusts the operating system kernel, and enclaves live *within* the address space of an untrusted host process.

We want to emphasize that Foreshadow exploits a microarchitectural *implementation* bug, and does not in any way undermine the architectural *design* of Intel SGX and TEEs in general. We strongly believe that the non-hierarchical protection model supported by these architectures is still as valuable as it was before. An important lesson from the recent wave of transient-execution attacks including Spectre, Meltdown, and Foreshadow, however, is that current processors exceed our levels of understanding [19, 185]. We therefore want to urge the research community to develop alternative hardware-software codesigns [48, 62], as well as inspectable open-source [189, 185] TEEs in the hopes of making future vulnerabilities easier to identify, mitigate, and recover from.

Mitigation strategies. State-of-the-art enclave side-channel hardening techniques [229, 42, 84, 237, 227, 40] offer little protection only and cannot address the root causes of the Foreshadow attack. These defenses commonly rely on hardware transactional memory (TSX) support to detect suspicious page fault and interrupt rates in enclave mode, which only marginally increases the bar for Foreshadow attackers. First, not all SGX-capable processors are also shipped with TSX extensions, ruling out TSX-based hardening techniques for Intel's critical Launch and Quoting Enclaves. Second, since the `egetkey` instruction is *not* allowed within a TSX transaction [114], adversaries can always interrupt a victim enclave unnoticed after key derivation to leak secrets (similar to Fig. 6.7). Furthermore, while the high interrupt rates generated by SGX-Step would be easily recognized, stealthy exploits can limit the number of enclave preemptions, or SMT-based Foreshadow variants can be executed concurrently from another logical core. Finally, we showed how to abuse SGX's `eldu` instruction to extract enclaved memory secrets without even executing the victim enclave, effectively rendering any software-only defense strategy inherently insufficient.

Only Intel is placed in a unique position to patch hardware-level CPU vulnerabilities. They recently announced "silicon-based changes to future products that will directly address the Spectre and Meltdown threats in hardware [...] later this year." [149] Likewise, we expect Foreshadow to be directly addressed with silicon-based changes in future Intel processors. The SGX design [14] includes a notion of TCB recovery by including the CPU security version number in all measurements (Section 6.2.1). As such, future microcode updates could in principle mitigate Foreshadow on existing SGX-capable processors. In this respect, beta microcode updates [128] have recently been distributed to mitigate Spectre, but, at the time of this writing, no microcode patches have been released addressing Meltdown nor Foreshadow. Given the fundamental nature of out-of-order CPU pipeline optimizations, we expect it may not be feasible to directly address the Foreshadow/Meltdown access control race condition in microcode. Alternatively, based on our findings (see Appendix D) that Foreshadow requires enclave data to reside in the L1 cache, we envisage a hardware-software co-design mitigation strategy. Foreshadow-resistant enclaves should be guaranteed that *(i)* both logical cores in an SMT setting execute within the same enclave [84, 237, 40], and *(ii)* the L1 cache is flushed upon each enclave exiting event [48]. We provide a comprehensive analysis of the mitigations deployed by Intel in response to Foreshadow-SGX in Section 6.9.4.

6.7 Related work

Several recent studies investigate attack surface for SGX enclaves. Existing attacks either exploit low-level memory safety vulnerabilities [154, 266], or abuse application-specific information leakage from side channels. Importantly, in contrast to Foreshadow, all known attacks explicitly fall out-of-scope of Intel SGX's threat model [120, 133], and can be effectively avoided by rewriting the victim enclave's code to exclude such vulnerabilities.

Conventional microarchitectural side channels [71] are, however, considerably amplified in the context of SGX's strengthened attacker model. This point has been repeatedly demonstrated in the form of a steady stream of high-resolution PRIME+PROBE CPU cache [225, 29, 79, 181, 92, 50] and branch prediction [156, 59] attacks against SGX enclaves. The additional capabilities of a root-level attacker have furthermore been leveraged to construct instruction-granular enclave interrupt primitives [257], and to exploit side-channel leakage from x86 memory paging [277, 258] and segmentation [91]. Unexpected side channels can also arise at the application level. We for example recently reported [118] a side-channel vulnerability in auto-generated `edger8r` code of the official Intel SGX SDK.

Concurrent research [39, 197] has demonstrated proof-of-concept Spectre-type speculation attacks against specially crafted SGX enclaves. Both attacks rely on executing vulnerable code within the victim enclave. Our attack, in contrast, does not require any specific code in the victim enclave, and can even extract memory contents without ever executing the victim enclave. While existing work shows vulnerable gadgets exist in the SGX SDK [39], such Spectre-type attack surface can be mitigated by patching the SDK. Recent Intel microcode updates furthermore address Spectre-type attacks against SGX enclaves directly at the hardware level, by cleansing the CPU's branch target buffer on every enclave transition [39].

6.8 Conclusion

We presented Foreshadow, an efficient transient-execution attack that completely compromises the confidentiality guarantees pursued by contemporary Intel SGX technology. We furthermore contributed practical attacks against Intel's trusted architectural enclaves, essentially dismantling SGX's local and remote attestation guarantees as well.

While, in the absence of a microcode patch, current SGX versions cannot maintain their hardware-level security guarantees, Foreshadow does assuredly not undermine the non-hierarchical protection model pursued by trusted execution environments, such as Intel SGX.

6.9 The microarchitecture behind Foreshadow

In the absence of a white-box view on Intel CPU internals, our original Foreshadow publication could only provide limited insights into the microarchitectural root cause behind the attack. After responsibly disclosing our findings, however, subsequent internal analysis by Intel [107] revealed that the same underlying processor vulnerability can also be abused to break conventional process or even virtual machine isolation. This postscript, based on our successive technical report [271], analyzes the microarchitectural root cause behind Foreshadow-type attacks and reviews mitigations that have been rolled out across the system stack.

6.9.1 L1 terminal fault

Modern Intel processors [114, 47] feature a carefully crafted virtually-indexed, physically-tagged L1 cache design. As illustrated in Fig. 6.9, this allows the first step of the address translation to proceed in parallel with the L1 cache set lookup, as the latter is completely determined by the virtual address specified by the load operation. After locating the correct L1 cache set, however, the processor should still determine whether any of the non-vacant ways within that set contain the required data. For this, the CPU matches the physical page number resulting from the address translation process against an internal metadata tag stored along each of the individual ways. Only when one of the non-vacant ways contains the exact right physical address tag, *i.e.*, upon an L1 hit event, is the data returned to the processor's execution units

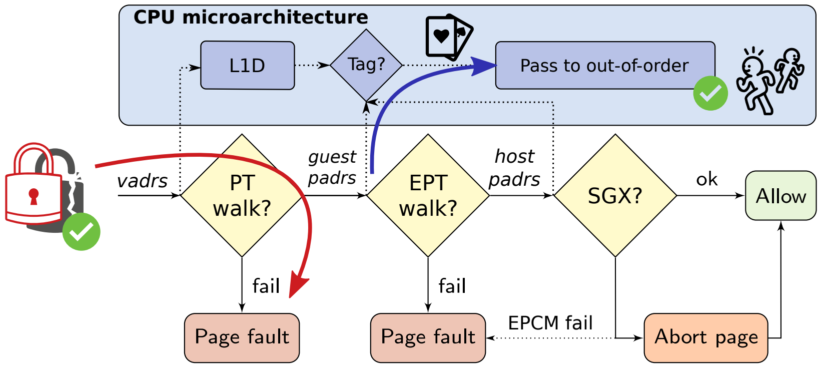


Figure 6.9: CPU address translation and L1 terminal fault behavior: documented architectural view (bottom) and undocumented microarchitectural transient-execution interactions (top).

The original Meltdown-US [162] attack showed that affected Intel processors do not transiently respect the “supervisor” page-table attribute, allowing to read cached kernel memory from L1D. For Foreshadow, on the other hand, Intel explains [107] that a special type of L1 Terminal Fault (L1TF) microarchitectural condition occurs when accessing a page-table entry with either the “present” bit cleared or a “reserved” bit set. In such cases, the CPU immediately aborts address translation, before applying any additional sanitizations enforced by the hypervisor or SGX in later stages of the page-table walk. Crucially, however, on vulnerable processors an illegitimate physical address is still derived from the faulting page-table entry and passed on to the L1D cache, before the exception is eventually raised at the architectural level. Any data present in L1D and tagged with that rogue physical address will now be illegally forwarded to the transient instructions following the faulting load, regardless of access permissions.

As illustrated in the top half of Fig. 6.9, data residing in the L1D cache is immediately forwarded to dependent operations upon successful tag comparison. While desirable from a performance perspective in the common case where no fault occurs, this behavior also implies that transient instructions may compute on unauthorized data when the address translation process is prematurely aborted due to a terminal fault. Hence, as with plain Meltdown, the non-existent memory request is properly blocked at the architectural level, by raising a terminal fault, but adversaries can still encode the results of secret-dependent computations at the microarchitectural level. Especially dangerous in this respect, is that the physical tag address has not been sanitized by subsequent stages in the page-table walk. We will explain below how early outing the address translation process due to a terminal fault allows Foreshadow adversaries to bypass all three successive memory protection phases, illustrated in Fig. 6.9, enforced by the operating system, hypervisor, and SGX

6.9.2 Foreshadow-OS: Escaping process isolation

Whenever the operating system kernel decides to swap a virtual memory page from DRAM to persistent storage, it is required to clear the present bit in the corresponding page-table entry. According to the processor's architectural specification [114], however, the kernel can freely use the remaining bits in a non-present page-table entry for bookkeeping purposes. The operating system may, for instance, decide to leave these bits unchanged, zero them out, or use them to store metadata that assists in bringing back the page from disk. Thus, while unprivileged user-space applications have no direct control over page-table entries, the metadata left by the OS may still be transiently interpreted as a valid physical page number that could point to sensitive data out of the process's architecturally accessible address space. Making things worse, when the operating system supports page sizes larger than 4 KiB, a user-space attacker can use the inadvertent mapping to access an unauthorized contiguous physical memory range of up to 2 MiB or 1 GiB.

In response to Foreshadow-OS, the kernel has to sanitize the physical address field of unmapped page-table entries so as to maintain process isolation on vulnerable processors. The "PTE inversion" technique adopted by the Linux kernel [46], for instance, simply inverts the physical page number bits in a page-table entry when it is marked as being not present. This serves as an elegant, zero-overhead mitigation that always ensures that unmapped pages point into non-existent physical memory, beyond the physical-address width supported by the processor [107].

6.9.3 Foreshadow-VMM: Escaping virtual machine isolation

While the above Foreshadow-OS variant allows unprivileged adversaries to transiently compute on unauthorized physical memory locations, they have no direct control over *which* exact physical addresses are being accessed. Foreshadow-type attacks therefore become even more devastating when considering untrusted virtual machines that directly control the guest-physical address input to the L1D tag comparison in Fig. 6.9. Such Foreshadow-VMM attacks allow an untrusted virtual machine to extract the host machine's entire L1D cache, including data belonging to the hypervisor or other virtual machines. The underlying problem is that a terminal fault in the guest page tables early outs the address translation process, such that guest-physical addresses are erroneously passed to the L1D data cache, without first being translated into a proper host-physical address.

In response to Foreshadow-VMM, Intel released microcode updates that provide a new `ia32_flush_cmd` MSR interface which allows the hypervisor to flush the entire L1D cache before handing over control to an untrusted virtual machine [107]. Additionally, to thwart SMT-based attacks across logical cores, the hypervisor must ensure to not schedule threads belonging to different virtual machines on the same physical CPU and to shoot down any guest executing on the sibling logical core before accessing secrets in hypervisor mode.

6.9.4 Foreshadow-SGX: Escaping enclave isolation

This chapter first demonstrated how adversaries can abuse the untrusted operating system's control over page tables to provoke terminal faults and read cached data from SGX enclaves, including full sealing and attestation keys from Intel's own architectural enclaves. This work directly led to Intel's subsequent discovery of the underlying L1TF microarchitectural condition and the wider category of Foreshadow-NG attack variants. Analogous to Foreshadow-VMM's extended page-table bypass above, Foreshadow-SGX essentially early outs the address translation, passing any cached enclave secrets to the transient out-of-order execution before the SGX machinery is allowed to replace them with abort page behavior.

Note that Section 6.3.1 furthermore proposed an alternative way to provoke terminal fault behavior via EPCM sanity checks (cf. Fig. 3.1 on page 73) when dereferencing a rogue virtual memory mapping from a customized attacker enclave. This Foreshadow-SGX subvariant has been dubbed "enclave-to-enclave" (E2E) in Intel's analysis [107].

In response to Foreshadow-SGX, Intel issued microcode patches for existing processors and developed silicon mitigations that are now included in newer processors enumerating `RDCL_NO`. The microcode mitigations [107] protect SGX enclaves in two ways: *(i)* by ensuring that no secrets are left in the L1D cache when the enclave is not executing; and *(ii)* by including SMT status during any key derivation and attestation. The former requirement is met by transparently flushing the L1D cache on every enclave entry and exit event. As we demonstrated in Section 6.3.5 that root adversaries can also abuse SGX's secure page swapping mechanism to bring secrets into the L1D cache without even executing the victim enclave, the `eldu` microcode has furthermore been modified to also perform an L1D flush before returning.

To address the second requirement, enabling remote parties to re-establish trust, Intel initiated TCB recovery [122] by upgrading SGX's security version number and extending remote attestation responses to reflect SMT status. Particularly, with the new microcode, different keys are derived depending on whether SMT is turned on or off by the BIOS. Since SMT cannot be dynamically re-enabled after booting, enclave data sealed when SMT was turned off, cannot be unsealed while SMT is active. Unfortunately, however, Intel [124] leaves it up to the remote stakeholder to decide whether or not to trust attestation responses from SMT-enabled systems. In light of the findings in Section 6.3.4 and also regarding more recent transient-execution attack developments [223, 216, 251], we consider SMT to be fundamentally broken and we decidedly recommend to *not* trust current SMT-enabled SGX platforms. Chapter 8 outlines a possible hardware mechanism that may aid to re-establish trust in SMT-enabled SGX systems, as long as it can be guaranteed that both sibling logical cores enter and exit the enclave in lock step [84, 40].

The above series of microcode countermeasures are indeed sufficient to block the data leakage aspect of Foreshadow, as described in this chapter. However, the next chapter introduces our insights on LVI [251], which shows that, despite extensive microcode mitigations, the legacy of Foreshadow is still haunting enclave security on vulnerable processors.

Chapter 7

LVI: Hijacking transient execution through microarchitectural load value injection

This chapter was previously published as:

> J. Van Bulck, D. Moghimi, M. Schwarz, M. Lipp, M. Minkin, D. Genkin, Y. Yuval, B. Sunar, D. Gruss, and F. Piessens. "LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection". In: *41st IEEE Symposium on Security and Privacy (S&P)*. May 2020, pp. 54–72

Preamble

This chapter presents Load Value Injection (LVI), an innovative technique to reversely exploit Meltdown-type microarchitectural data leakage. LVI abuses that faulting or assisted loads, executed by a legitimate victim program, may transiently use dummy values or poisoned data from various microarchitectural buffers, before eventually being re-issued by the processor. We for the first time combine Spectre-style code gadgets with Meltdown-type illegal data flows to bypass existing defenses and inject attacker-controlled data into a victim enclave's transient execution. State-of-the-art Meltdown and Spectre defenses, including widespread silicon-level and microcode mitigations, are orthogonal to our novel LVI techniques. Instead, fully mitigating our attacks requires serializing the processor pipeline with `lfence` speculation barriers after possibly *every* memory load. Depending on the application and optimization strategy, we observe extensive overheads of factor 2 to 19 for prototype implementations of the full mitigation.

The key idea of turning microarchitectural data sampling into microarchitectural data injection developed when working on the Fallout and Zombieload transient-execution attacks. We first reported this novel exploitation technique, including proof-of-concepts for all of the variants described in this chapter, to Intel on April 4, 2019, at which point we engaged in an extended embargo period until March 10, 2020. Near the end of the embargo period, in February 2020, the LVI-LFB subvariant was independently rediscovered and reported by researchers at Bitdefender, who provided Intel with a proof-of-concept for LVI-LFB in a synthetic cross-process SMT scenario. To raise awareness and disseminate our findings to the wider public, the website `https://lviattack.eu/` was created.

From an attack perspective, LVI represents an advanced exploitation technique that ultimately combines many of the insights developed in the previous chapters: from untrusted pointers in Chapter 2, via page-table manipulations and SGX-Step in Chapters 3 and 4, to incorrect transient forwarding in Chapter 6. Given these considerable exploitation challenges, we believe that LVI is principally relevant as an attack by an untrusted operating system targeting Intel SGX enclaves. While this chapter also shows that none of the ingredients for LVI are strictly unique to Intel SGX, and LVI may in principle even apply to traditional cross-process, user-to-kernel, and sandboxed environments, we presently consider such unprivileged non-enclave attack scenarios to be of mainly academic interest. That is, LVI presents an important attack vector in the privileged Intel SGX adversary model, but further research should tell whether practically exploitable LVI gadgets and reliable ways to provoke exceptions also exist for non-enclave victim programs.

Current status of mitigations. From a defensive perspective, LVI marks the end of transparently patching Meltdown-type processor vulnerabilities in CPU microcode. Our research closed the gap between Spectre-type misspeculation and Meltdown-type data extraction attacks, prompting Intel to refine their terminology for software guidance and adopt the term “transient execution” to more accurately describe the common underlying effect [125]. In response to our findings, extensive `lfence` software mitigations landed in the GNU assembler, LLVM, and Microsoft’s MSVC to allow at least compilation of SGX enclaves to remain secure on LVI-vulnerable systems. In line with the evaluation performed in this chapter, several independent researchers have since confirmed the grave performance impact of these mitigations. Larabel [151] evaluates LVI mitigation options provided by the GNU assembler in a variety of applications and measures overheads of factor 4 to 15 for the full `-mlfence-after-load` mitigation. To somewhat alleviate this impact, Intel developed an LLVM-based compiler pass which optimally inserts `lfence` instructions [106]. The evaluation in this chapter shows that Intel's optimized LLVM mitigation indeed clearly outperforms naive assembler-level solutions. Further, in line with our findings for partial LVI mitigations, Larabel [150] reports moderate overheads in the order of 9 % for LLVM's new `-mlvi-cfi` option, which, however, only serializes indirect branches and does not protect against data-only LVI attacks.

Finally, an alternative mitigation approach was proposed by Google engineers in the form of "Speculative Execution Side Effect Suppression" (SESES) [30], an experimental LLVM compiler pass that aims to provide a more generic, last-resort measure for high-security SGX enclaves. Instead of blocking the illegal transient data flow by inserting fences after vulnerable load instructions, SESES attempts to more generally prevent side-channel leakage from the transient-execution domain by inserting fences before every memory operation or control flow redirection. Expectedly, however, in a performance evaluation on the BoringSSL cryptographic library, this mitigation caused extensive slow downs of factor 6 to 23 [30]. Moreover, by attempting to prevent known side-channel leakage sources, SESES might not protect against alternative covert channels that do not rely on redirecting transient control flow or data accesses.

7.1 Introduction

Recent research on transient-execution attacks has been characterized by a sharp split between on the one hand Spectre-type misspeculation attacks, and on the other hand, Meltdown-type data extraction attacks. The first category, Spectre-type attacks [146, 148, 167, 23, 102], trick a victim into transiently diverting from its intended execution path. Particularly, by poisoning the processor's branch predictor machinery, Spectre adversaries steer the victim's transient execution to gadget code snippets, which inadvertently expose secrets through the shared microarchitectural state. Importantly, Spectre gadgets execute entirely *within* the victim domain and can hence only leak architecturally accessible data.

The second category consists of Meltdown-type attacks [162, 249, 271, 234, 223, 216, 35], which target architecturally inaccessible data by exploiting illegal data flow from faulting or assisted instructions. Particularly, on vulnerable processors, the results of unauthorized loads are still forwarded to subsequent transient operations, which may encode the data before an exception is eventually raised. Over the past year, delayed exception handling and microcode assists have been shown to transiently expose data from various microarchitectural elements (*i.e.*, L1D cache [162, 249], FPU register file [234], line-fill buffer [162, 223, 216], store buffer [35], and load ports [216, 108]). Unlike Spectre-type attacks, a Meltdown attacker in one security domain can directly exfiltrate architecturally inaccessible data belonging to another domain (e.g., kernel memory). Consequently, existing Meltdown mitigations focus on restricting the attacker's point of view, e.g., placing victim data out of reach [85], flushing buffers after victim execution [107, 108], or zeroing unauthorized data flow directly at the silicon level [128].

Given the widespread deployment of Meltdown countermeasures, including changes in operating systems and CPUs, we ask the following fundamental questions in this chapter:

Can Meltdown-type effects only be used for leakage or also for injection? Would current hardware and software defenses suffice to fully eradicate Meltdown-type threats based on illegal data flow from faulting or assisted instructions?

7.1.1 Our results and contributions

In this chapter, we introduce an innovative class of Load Value Injection (LVI) attack techniques. Our key contribution is to recognize that, under certain adversarial conditions, unintended microarchitectural leakage can also be inverted to *inject* incorrect data into the victim's transient execution. Being essentially a "reverse Meltdown"-type attack, LVI abuses that a faulting or assisted load instruction executed within a victim domain does not always yield the expected result, but may instead transiently forward dummy values or (attacker-controlled) data from various microarchitectural buffers. We consider attackers that can either directly or indirectly induce page faults or microcode assists during victim execution. LVI provides such attackers with a primitive to force a *legitimate* victim execution to transiently compute on "poisoned" data (e.g., pointers, array indices) before the CPU eventually detects the fault condition and discards the pending architectural state changes. Much like in Spectre attacks, LVI relies on "confused deputy" code gadgets surrounding the faulting or assisted load in the victim to hijack transient control flow and disclose information. We are the first to combine Meltdown-style microarchitectural data leakage with Spectre-style code gadget abuse to compose a novel type of transient load value injection attacks.

Table 7.1 summarizes how Spectre [146] first applied an injection-based methodology to invert prior branch prediction side-channel attacks, whereas LVI

Table 7.1: Characterization of known side-channel and transient-execution attacks in terms of targeted microarchitectural predictor or data buffer (vertical axis) vs. leakage- or injection-based methodology (horizontal axis). The LVI attack plane, first explored in this chapter, is indicated on the lower right and applies an injection-based methodology known from Spectre attacks (upper right) to reversely exploit Meltdown-type data leakage (lower left).

Methodology Buffer		**Leakage**	**Injection**
Prediction history	PHT	BranchScope [59], Bluethunder [105]	Spectre-PHT [146]
	BTB	SBPA [2], BranchShadow [156]	Spectre-BTB [146]
	RSB	Hyper-Channel [33]	Spectre-RSB [148, 167]
	STL	–	Spectre-STL [102]
Program data	L1D	Meltdown [162]	LVI-NULL
	L1D	Foreshadow [249]	LVI-L1D
	FPU	LazyFP [234]	LVI-FPU
	SB	Fallout [35]	LVI-SB
	LFB/LP	ZombieLoad [223], RIDL [216]	LVI-LFB/LP

similarly shows that recent Meltdown-type microarchitectural data leakage can be reversely exploited. Looking at Table 7.1, it becomes apparent that Spectre-style injection attacks have so far only been applied to auxiliary history-based branch prediction and dependency prediction buffers that accumulate program metadata to steer the victim's transient execution indirectly. Our techniques, on the other hand, intervene much more directly in the victim's transient data stream by injecting erroneous load values straight from the CPU's memory hierarchy, *i.e.*, intermediate load and store buffers and caches.

These fundamentally different microarchitectural behaviors (*i.e.*, misprediction vs. illegal data flow) also entail that LVI requires defenses that are orthogonal and complementary to existing Spectre mitigations. Indeed, we show that some of our exploits can transiently redirect conditional branches, even *after* the CPU's speculation machinery correctly predicted the architectural branch outcome. Furthermore, since LVI attacks proceed entirely *within* the victim domain, they remain intrinsically immune to widely deployed software and microcode Meltdown mitigations that flush microarchitectural resources after victim execution [107, 108]. Disturbingly, our analysis reveals that even state-of-the-art hardened Intel CPUs [128], with silicon changes that zero out illegal data flow from faulting or assisted instructions, do not fully eradicate LVI-based threats.

Our findings challenge prior views that, unlike Spectre, Meltdown-type threats could be eradicated straightforwardly at the operating system or hardware levels [36, 278, 98, 170, 82]. Instead, we conclude that potentially every illegal data flow in the microarchitecture can be inverted as an injection source to purposefully disrupt the victim's transient behavior. This observation has profound consequences for reasoning about secure code. We argue that depending on the attacker's capabilities, ultimately, *every* load operation in the victim may potentially serve as an exploitable LVI gadget. This is in sharp contrast to prior Spectre-type effects that are contained around clear-cut (branch) misprediction locations.

Successfully exploiting LVI requires the ability to induce page faults or microcode assists during victim execution. We show that this requirement can be most easily met in Intel SGX environments, where we develop several proof-of-concept attacks that abuse dangerous real-world gadgets to arbitrarily divert transient control flow in the enclave. We furthermore mount a novel transient fault attack on AES-NI to extract full cryptographic keys from a victim enclave. While LVI attacks in non-SGX environments are generally much harder to mount, we consider none of the adversarial conditions for LVI to be unique to Intel SGX. We explore consequences for traditional process isolation by showing that, given a suitable LVI gadget and a faulting or assisted load in the kernel, arbitrary supervisor memory may leak to user space. We also show that the same vector could be exploited in a cross-process LVI attack.

Underlining the impact and the practical challenges arising from our findings, Intel plans to mitigate LVI by extensive revisions at the compiler and assembler levels to allow at least compilation of SGX enclaves to remain secure on LVI-vulnerable systems. Particularly, fully mitigating LVI requires introducing `lfence` instructions to serialize the processor pipeline after possibly *every* memory load operation. Additionally, certain instructions featuring implicit loads, including the pervasive x86 `ret` instruction, should be blacklisted and emulated with equivalent serialized instruction sequences. We observe extensive performance overheads of factor 2 to 19 for our evaluation of prototype compiler mitigations, depending on the application and whether `lfence`s were inserted by an optimized compiler pass or through a naive post-compilation assembler approach.

In summary, our main contributions are as follows:

- We show that Meltdown-type data leakage can be inverted into a Spectre-like Load Value Injection (LVI) primitive. LVI transiently hijacks data flow, and thus control flow.
- We present an extensible taxonomy of LVI-based attacks.
- We show the insufficiency of silicon changes in the latest generation of acclaimed Meltdown-resistant Intel CPUs

- We develop practical proof-of-concept exploits against Intel SGX enclaves, and we discuss implications for traditional kernel and process isolation in the presence of suitable LVI gadgets and faulting or assisted loads.
- We evaluate compiler mitigations and show that a full mitigation incurs a runtime overhead of factor 2 to 19.

7.1.2 Responsible disclosure and impact

We responsibly disclosed LVI to Intel on April 4, 2019. We also described the non-Intel-specific parts to ARM and IBM. To develop and deploy appropriate countermeasures, Intel insisted on a long embargo period for LVI, namely, until March 10, 2020 (CVE-2020-0551, Intel-SA-00334). Intel considers LVI particularly severe for SGX and provides a compiler and assembler-based full mitigation for enclave programs, described and evaluated in Section 7.9. Intel furthermore acknowledged that LVI may in principle be exploited in non-SGX user-to-kernel or process-to-process environments and suggested addressing by manually patching any such exploitable gadgets upon discovery.

We also contacted Microsoft, who acknowledged the relevance when paging out kernel memory and continues to investigate the applicability of LVI to the Windows kernel. Microsoft likewise suggested addressing non-SGX scenarios by manually patching any exploitable gadgets upon discovery.

7.2 Background

In this section, we provide background on CPU microarchitecture, Intel SGX, and transient-execution attacks.

7.2.1 CPU microarchitecture

In a complex Instruction Set Architecture (ISA) such as Intel x86 [113] instructions are decoded into RISC-like *micro-ops*. The CPU executes micro-ops from the reorder buffer out of order when their operands become available but retires micro-ops in order. Modern CPUs perform history-based speculation to predict branches and data dependencies ahead of time. While the CPU implements the most common fast-path logic directly in hardware, certain corner cases are handled by issuing a *microcode assist* [47, 74]. In such a corner case, the CPU flags the corresponding micro-op to be re-issued later as a microcode

P	**RW**	**US**	WT	*UC*	*A*	*D*	S	G	**Physical Page Number**	**Rsvd.**	**XD**

Figure 7.1: Overview of an x86 page-table entry and attributes that may trigger architectural page fault exceptions (red, bold) or microcode assists (green, italic). Attributes that are periodically cleared by some OS kernels are underlined; all other fields can only be modified by privileged attackers.

routine. When encountering exceptions, misspeculations, or microcode assists, the CPU pipeline is flushed, and any outstanding micro-op results are discarded from the reorder buffer. This rollback ensures that the results of unintended *transient instructions*, which were wrongly executed ahead of time, are never visible at the architectural level.

Address translation Modern CPUs use virtual addresses to isolate concurrently running tasks. A multi-level page-table hierarchy is set up by the operating system (OS) or hypervisor to translate virtual to physical addresses. The lower 12 address bits are the index into a 4 KiB page, while higher address bits index a series of page-table entries (PTEs) that ultimately yield the corresponding physical page number (PPN). Figure 7.1 overviews the layout of an Intel x86 PTE [114, 47]. Apart from the physical page number, PTEs also specify permission bits to indicate whether the page is present, accessible to user space, writable, or executable.

The translation lookaside buffer (TLB) caches recent address translations. Upon a TLB miss, the CPU's page-miss handler performs a page-table walk and updates the TLB. The CPU's TLB miss handler circuitry is optimized for the fast path, and delegates more complex operations, e.g., setting of "accessed" and "dirty" PTE bits, using microcode assists [74]. Depending on the permission bits, a page fault (`#PF`) may be raised to abort the memory operation and redirect control to the OS.

Memory hierarchy Superscalar CPUs consist of multiple physical cores connected through a bus interconnect to the memory controller. As the main memory is relatively slow, the CPU uses a complex memory subsystem (cf. Fig. 7.2), including various caches and buffers. On Intel CPUs, the L1 cache is the fastest and smallest, closest to the CPU, and split into a separate unit for data (L1D) and instructions (L1I). L1D is usually a 32 KiB 8-way set-associative cache. It is virtually-indexed and physically-tagged, such that lookups can proceed in parallel to address translation. A cache line is 64 bytes, which also defines the granularity of memory transactions (load and store) through the cache hierarchy. To handle various sized memory operations, L1D is connected

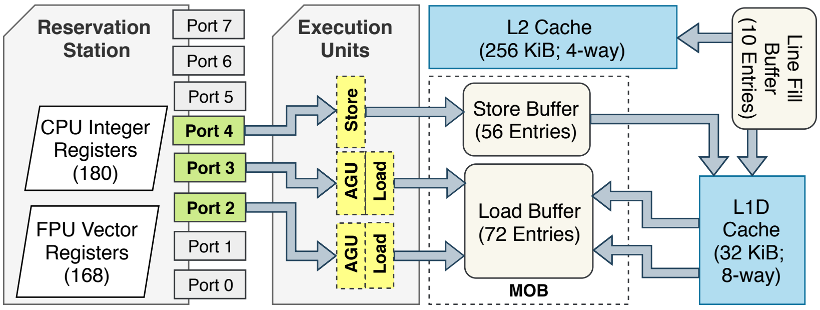

Reservation Station
CPU Integer Registers (180)
FPU Vector Registers (168)
Port 7
Port 6
Port 5
Port 4
Port 3
Port 2
Port 1
Port 0
Execution Units
Store
AGU
Load
AGU
Load
L2 Cache (256 KiB; 4-way)
Line Fill Buffer (10 Entries)
Store Buffer (56 Entries)
Load Buffer (72 Entries)
MOB
L1D Cache (32 KiB; 8-way)

Figure 7.2: Overview of the memory hierarchy in modern x86 microarchitectures.

to a memory-order buffer (MOB), which is interfaced with the CPU's register files and execution units through dedicated load ports (LPs).

The MOB includes a store buffer (SB) and load buffer (LB), plus various dependency prediction and resolution circuits to safeguard correct ordering of memory operations. The SB keeps track of outstanding store data and addresses to commit stores in order, without stalling the pipeline. When a load entry in LB is predicted to not depend on any prior store, it is executed out of order. If a store-to-load (STL) dependency is detected, the SB forwards the stored data to the dependent load. However, if the dependency of a load and preceding stores is not predicted correctly, these optimizations may lead to situations where the load consumes either stale data from the cache or wrong data from the SB while the CPU reissues the load to obtain the correct data. These optimizations within the MOB can undermine security [130, 102, 35].

Upon on L1D cache miss, data is fetched from higher levels in the memory hierarchy via the line-fill buffer (LFB), which keeps track of outstanding load and store requests without blocking the L1D cache. The LFB retrieves data from the next cache levels or main memory and afterward updates the corresponding cache line in L1D. An "LFB hit" occurs if the CPU has a cache miss for data in a cache line that is in the LFB. Furthermore, uncacheable memory and non-temporal stores bypass the cache hierarchy using the LFB.

7.2.2 Intel SGX

Intel Software Guard Extensions (SGX) [47] provides processor-level isolation and attestation for secure "enclaves" in the presence of an untrusted OS. Enclaves are contained in the virtual address space of a conventional user-space process, and virtual-to-physical address mappings are left under explicit control of untrusted system software. To protect against active address remapping attackers [47], SGX maintains a shadow entry for every valid enclave page in the enclave page-cache map (EPCM) containing amongst others the expected virtual address. Valid address mappings are cached in the TLB, which is flushed upon enclave entry, and a special EPCM page fault is generated when encountering an illegal virtual-to-physical mapping (cf. Fig. 3.1 on page 73).

However, previous work showed that Intel SGX root attackers can mount high-resolution, low-noise side-channel attacks through the cache [181, 225, 29], branch predictors [156, 59, 105], page-table accesses [277, 258, 257], or interrupt timing [256]. In response to recent transient-execution attacks [249, 223, 216, 39], which can extract enclave secrets from side-channel resistant software, Intel released microcode updates which flush microarchitectural buffers on every enclave entry and exit [107, 108].

7.2.3 Transient-execution attacks

Modern processors safeguard architectural consistency by discarding the results of any outstanding transient instructions when flushing the pipeline. However, recent research on transient-execution attacks [146, 162, 249] revealed that these unintended transient computations may leave secret-dependent traces in the CPU's microarchitectural state, which can be subsequently recovered through side-channel analysis. Following a recent classification [36], we refer to attacks exploiting misprediction [146, 141, 148, 167, 102] as Spectre-type, and attacks exploiting transient execution after a fault or microcode assist [162, 249, 234, 35, 223, 216] as Meltdown-type.

Meltdown-type attacks extract unauthorized program data across architectural isolation boundaries. Over the past years, faulting loads with different exception types and microcode assists have been demonstrated to leak secrets from intermediate microarchitectural buffers in the memory hierarchy: the L1 data cache [162, 249, 271], the line-fill buffer and load ports [223, 216], the FPU register file [234], and the store buffer [35, 220].

A perpendicular line of Spectre-type attacks, on the other hand, aims to steer transient execution in the victim domain by poisoning various microarchitectural predictors. Spectre attacks are limited by the depth of the transient-execution window, which is ultimately bounded by the size of the reorder buffer [261]. Most Spectre variants [146, 148, 167] hijack the victim's transient control flow by mistraining shared branch prediction history buffers prior to entering the victim domain. Yet, not all Spectre attacks depend on branch history, e.g., in Spectre-STL [102] the processor's memory disambiguation predictor incorrectly speculates that a load does not depend on a prior store, allowing the load to transiently execute with a stale outdated value. Spectre-STL has, for instance, been abused to hijack the victim's transient control flow in case the stale value is a function pointer or indirect branch target controlled by a previous attacker input [261].

7.3 Load value injection

Table 7.1 summarizes the existing transient-execution attack landscape. The Spectre family of attacks (upper right) contributed an injection-based methodology to invert prior prediction history side channels (upper left) by abusing confused-deputy code gadgets within the victim domain. At the same time, Meltdown-type attacks (lower left) demonstrated cross-domain data leakage. The LVI attack plane (lower right) remains unexplored until now. In this chapter, we adopt an injection-based methodology known from Spectre attacks to reversely exploit Meltdown-type microarchitectural data leakage. LVI brings a significant change in the threat model, similar to switching from branch history side channels to Spectre-type attacks. Crucially, LVI has the potential to replace the outcome of *any* victim load, including implicit load micro-ops like in the x86 `ret` instruction, with attacker-controlled data. This is in sharp contrast to Spectre-type attacks, which can only replace the outcomes of branches and store-to-load dependencies by poisoning execution metadata accumulated in various microarchitectural predictors.

7.3.1 Attack overview

We now outline how LVI can hijack the result of a trusted memory load operation, under the assumption that attackers can provoke page faults or microcode assists for (arbitrary) load operations in the victim domain. The attacker's goal is to force a victim to transiently compute on unintended data, other than the expected value in trusted memory. Injecting such unexpected load values forces a victim to transiently execute gadget code immediately following the faulting or assisted load instruction with unintended operands.

Figure 7.3 overviews how LVI exploitation can be abstractly broken down into four phases.

1. In the first phase, the microarchitecture is optionally prepared in the desired state by filling a hidden buffer with an (attacker-controlled) value A.

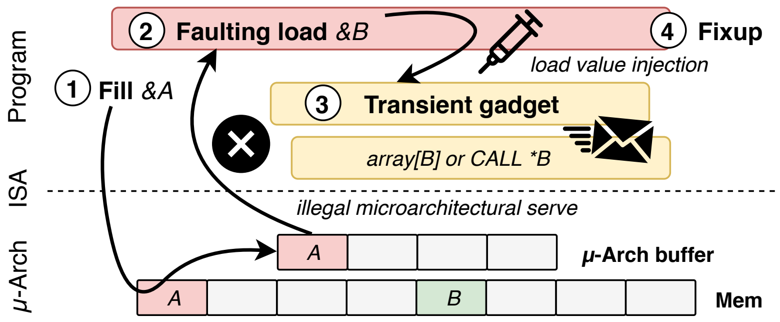


Figure 7.3: Phases in a Load Value Injection (LVI) attack: (1) a microarchitectural buffer is filled with value A; (2) the victim executes a faulting or assisted load to retrieve value B which is incorrectly served from the microarchitectural buffer; (3) the injected value A is forwarded to transient instructions following the faulting or assisted load, which may now perform unintended operations depending on the available gadgets; (4) the CPU flushes the faulting or assisted load together with all other transient instructions.

2. The victim then executes a load micro-op to fetch a trusted value B. However, in case this instruction suffers a page fault or microcode assist, the CPU may erroneously serve the load request from the microarchitectural buffer. This results in incorrect forwarding of value A to dependent transient micro-ops following the faulting or assisted load. At this point, the attacker has succeeded in tricking the victim into transiently computing on the injected value A instead of the trusted value B.

3. These unintended transient computations may subsequently expose victim secrets through microarchitectural state changes. Depending on the specific “gadget” code surrounding the original load operation, LVI may either encode secrets directly or serve as a transient control or data flow redirection primitive to facilitate second-stage gadget abuse, e.g., when B is a trusted code or data pointer.

4. The architectural results of gadget computations are eventually discarded at the retirement of the faulting or assisted load instruction. However, secret-dependent traces may have been left in the CPU’s microarchitectural state, which can be subsequently recovered through side-channel analysis.

```
1 void call_victim(size_t untrusted_arg) {
2     *arg_copy = untrusted_arg;
3     array[**trusted_ptr * 4096];
4 }
```

Listing 7.1: An LVI toy gadget for leaking arbitrary data from a victim domain.

7.3.2 A toy example

Listing 7.1 provides a toy LVI gadget to illustrate how faulting loads in a victim domain may trigger incorrect transient forwarding. Our example gadget bears a high resemblance to known Spectre gadgets but notably does *not* rely on branch misprediction or memory disambiguation. Furthermore, our gadget executes entirely within the victim domain and is hence not affected by widely deployed microcode mitigations that flush microarchitectural buffers on context switch. Regardless of the prevalence of this specific toy gadget, it serves as an initial example which is easy to understand and illustrates the power of LVI as a generic attack primitive.

Following the general outline of Fig. 7.3, the gadget code in Listing 7.1 first copies a 64-bit value `untrusted_arg` provided by the attacker into trusted memory (e.g., onto the stack) at line 2. In the example, the argument copy is further not used, and this store operation merely serves to bring some attacker-controlled value into some microarchitectural buffer. Subsequently, in the second phase of the attack, a pointer-to-pointer `trusted_ptr` (e.g., a pointer in a dynamically allocated struct) is dereferenced at line 3. We assume that, upon the first-level pointer dereference, the victim suffers a page fault or microcode assist. The faulting load causes the processor to incorrectly forward the attacker's value `untrusted_arg` that was previously brought into the store buffer by the completely unrelated store at line 2, like in a Meltdown-type attack [35]. At this point, the attacker has succeeded in replacing the architecturally intended value at address `*trusted_ptr` with her own chosen value. In the third phase of the attack, the gadget code transiently uses `untrusted_arg` as the base address for a second-level pointer dereference and uses the result as an index in a lookup table. Similar to a Spectre gadget [146], the lookup in `array` serves as the sending end of a cache-based side channel, allowing to encode arbitrary memory locations within the victim's address space.

Figure 7.4 illustrates how in the final phase of the attack, after the fault has been handled and the load has been re-issued allowing the victim to complete, adversaries can abuse access timings to the probing array to reconstruct secrets from the victim's transient execution. Notably, the timing diagram showcases two clear drops: one dip corresponds to the architecturally intended value that

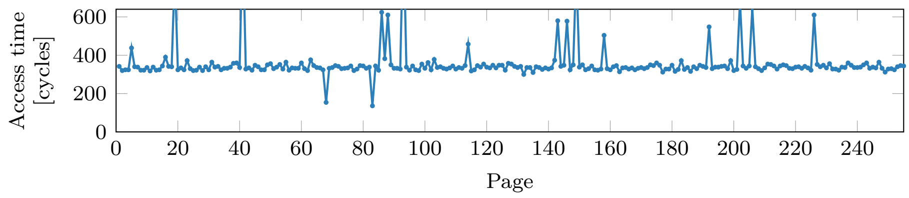


Figure 7.4: Access times to the probing array after the execution of Listing 7.1. The dip at 68 (`'D'`) is the transmission specified by the victim's architectural program semantics. The dip at 83 (`'S'`) is the victim secret at the address `untrusted_arg` injected by the attacker.

was processed after the faulting load got successfully re-issued, while the second dip corresponds to the victim secret at the address chosen by the attacker. This toy example hence serves as a clear illustration of the danger of incorrect transient forwarding following a faulting load in a victim domain. We elaborate further on attacker assumptions and gadget requirements for different LVI variants in Sections 7.4 and 7.6 respectively.

7.3.3 Difference with Spectre-type attacks

While LVI adopts a gadget-based exploitation methodology known from Spectre-type attacks, both attack families exploit fundamentally different microarchitectural behaviors (*i.e.*, incorrect transient forwarding vs. misprediction). We explain below how LVI is different from and requires orthogonal mitigations to known Spectre variants.

LVI vs. branch prediction. Most Spectre variants [146, 148, 167, 36] transiently hijack branch outcomes in a victim process by poisoning various microarchitectural branch prediction history buffers. On recent and updated systems, these buffers are typically not simultaneously shared anymore and flushed on context switch. Furthermore, to foil mistraining strategies within a victim domain, hardened compilers insert explicit `lfence` barriers after potentially mispredicted branches.

In contrast, LVI allows to hijack the result of *any* victim load micro-op, not just branch targets. By directly injecting incorrect values from the memory hierarchy, LVI allows data-only attacks as well as control-flow redirection in the transient domain. Essentially, LVI and Spectre exploit different subsequent phases of the victim's transient execution: while Spectre hijacks control flow *before* the architectural branch outcome is known, LVI-based control-flow redirection manifests only *after* the victim attempts to fetch the branch-target address from application memory. LVI does not rely on mistraining of any (branch) predictor, and hence, applies even to CPUs without exploitable prediction elements, and to systems protected with up-to-date microcode and compiler mitigations.

LVI vs. speculative store bypass. Spectre-STL [102] exploits the memory disambiguation predictor, which may speculatively issue a load even before all prior store addresses are known. That is, in case a load is mispredicted to not depend on a prior store, the store is incorrectly not forwarded and the load transiently executes with a stale outdated value.

Crucially, while Spectre-STL is strictly limited to injecting stale values for loads that closely follow a store to the exact same address, LVI has the potential to replace the result of *any* victim load with unrelated and possibly attacker-controlled data. LVI therefore drastically widens the spectrum of incorrect transient paths. As an example, the code in Listing 7.1 is not in any way exposed to Spectre-STL since the store and load operations are to different addresses, but this gadget can still be exploited with LVI in case the load suffers a page fault or microcode assist. Consequently, LVI is also not affected by Spectre-STL mitigations, which disable the memory disambiguation predictor in microcode or hardware.

LVI vs. value prediction. While value prediction has already been proposed more than two decades ago [160, 262], commercial CPUs do not implement it yet due to complexity concerns [201]. As long as no commercial CPU supports value speculation, Spectre-type value misprediction attacks are purely theoretical. In LVI, there is no mistraining of any (value) predictor, and hence, it applies to today's CPUs already.

7.4 Attacker model and assumptions

We focus on software adversaries who want to disclose secrets from an isolated victim domain, e.g., the OS kernel, another process, or an SGX enclave. For SGX, we assume an attacker with root privileges, *i.e.*, the OS is under control of the attacker [47]. Successful LVI attacks require carefully crafted adversarial conditions. In particular, we identify the following three requirements for LVI exploitability:

Incorrect transient forwarding. As with any fault injection attack, LVI requires some form of exploitable incorrect behavior. We exploit that faulting or assisted loads do not always yield the expected architectural result, but may transiently serve dummy values or poisoned data from various microarchitectural buffers. There are many instances of incorrect transient forwarding in modern CPUs [162, 249, 234, 36, 223, 216, 35]. In this work, we show that such incorrect transient forwarding is *not* limited to cross-domain data leakage. We are the first to show cross-domain data injection *and* identify dummy `0x00` values as an exploitable incorrect transient forwarding source, thereby widening the scope of LVI even to microarchitectures that were previously considered Meltdown-resistant.

Faulting or assisted loads. LVI requires firstly the ability to (directly or indirectly) provoke architectural exceptions or microcode assists for legitimate loads executed by the victim. This includes *implicit* load micro-ops as part of larger ISA instructions, e.g., popping the return address from the stack in the x86 `ret` instruction. Privileged SGX attackers can straightforwardly provoke page faults for enclave memory loads by modifying untrusted page tables, as demonstrated by prior research [277, 258]. Even unprivileged attackers can induce demand paging non-present faults by abusing the OS interface to unmap targeted victim pages through legacy interfaces or contention of the shared page cache [83]. Finally, more recent works showed that Meltdown-type effects are *not* limited to architectural exceptions, but also exist for assisted loads [223, 216, 35]. In case a microcode assist is required, the load micro-op does not architecturally commit, but may still transiently forward incorrect values before being re-issued as a microcode routine. Microcode assists occur in a wide variety of conditions, including subnormal floating point numbers and setting of "accessed" and "dirty" PTE bits [47, 108].

Code gadgets. A final yet crucial requirement for LVI is the presence of a suitable code gadget that allows to hijack the victim's transient execution and encode unintended secrets in the microarchitectural state. In practice, this requirement comes down to identifying a load operation in the victim code that can be faulting or assisted, followed by an instruction sequence that redirects control or data flow based on the loaded value (e.g., a pointer, or array index). We find that there are many different types of gadgets which mostly consist of only a few ubiquitously used instructions. We provide practical instances of such exploitable gadgets in Section 7.6.

7.5 Building blocks of the attack

We compose transient fault-injection attacks using the three building blocks described in the previous section and Fig. 7.3.

7.5.1 Phase $\mathcal{P}1$: Microarchitectural poisoning

The main challenge in the first phase is to prepare the CPU's microarchitectural state such that a (controlled) incorrect transient forwarding happens for the faulting load in the second stage. We later classify LVI variants based on the microarchitectural buffer that forwards the incorrect data. Depending on the variant, it suffices in this phase to fill a particular buffer (cf. Section 7.2.1: L1D, LFB, SB, LP) with a chosen value at a chosen location. This is not always a requirement, as we also consider a special LVI-NULL variant that abuses incorrect forwarding of `0x00` dummy values which are often returned when faulting loads miss the cache, or on Meltdown-resistant microarchitectures [128]. Such null values are "hard wired" in the CPU, and the poisoning phase can hence be entirely omitted for LVI-NULL attacks.

In a straightforward scenario, the shared microarchitectural buffer can be poisoned directly from within the attacker context. This scenario assumes, however, that said buffer is *not* explicitly overwritten or flushed when switching from the attacker to the victim domain, which is often not anymore the case with recent software and microcode mitigations [107, 108]. Alternatively, for buffers competitively shared among logical CPUs, LVI attackers can resort to concurrent poisoning from a co-resident SMT core running in parallel to the victim [249, 223, 216].

Finally, in the most versatile LVI scenario, the attack runs *entirely* within the victim domain without placing any assumptions on prior attacker execution or co-residence. We abuse appropriate "fill gadgets" preceding the faulting load within the victim execution. As explored in Section 7.6, LVI variants may impose more or fewer restrictions on suitable fill gadget candidates. The most generically exploitable fill gadget loads or stores attacker-controlled data from or to an attacker-chosen location, without introducing any architectural security problem. This is a common case if attacker and victim share an address space (enclave, user-kernel boundary, sandbox) and exchange arguments or return values via pointer passing.

7.5.2 Phase $\mathcal{P}2$: Provoking faulting or assisted loads

In the second and principal LVI phase, the victim executes a faulting or assisted load micro-op triggering incorrect transient forwarding. The crucial challenge here is to provoke a fault or assist for a *legitimate* and trusted load executed by the victim.

Intel SGX. When targeting Intel SGX enclaves, privileged adversaries can straightforwardly manipulate PTEs in the untrusted OS to provoke page-fault exceptions [277] or microcode assists [223, 35]. Even user-space SGX attackers can indirectly revoke permissions for enclave code and data pages through the unprivileged `mprotect` system call [249]. Alternatively, if the targeted LVI gadget requires a more precise temporal granularity, privileged SGX attackers can leverage a single-stepping interrupt attack framework like SGX-Step [257] to manipulate PTEs and revoke enclave-page permissions precisely at instruction-level granularity.

Generalization to other environments. In the more general case of unprivileged cross-process, cross-VM, or sandboxed attackers, we investigated exploitation via memory contention. Depending on the underlying OS or hypervisor implementation and configuration, an attacker can forcefully evict selected virtual memory pages belonging to the victim via legacy interfaces or by increasing physical memory utilization [83]. The "present" bit of the associated PTE is cleared (cf. Fig. 7.1), and the next victim access faults. On Windows, this can even affect the kernel heap due to demand paging [213].

Furthermore, prior research has shown that the page-replacement algorithm on Windows periodically clears "accessed" and "dirty" PTE bits [223]. Hence, unprivileged attackers can simply wait until the OS clears the accessed bit on the victim PTE. Upon the next access to that page, the CPU's page-miss handler circuitry prematurely aborts the victim's load micro-op to issue a microcode assist for re-setting the accessed bit on the victim PTE [47, 223]. Finally, even without any OS intervention, a victim program may expose certain load gadget instructions that always require a microcode assist (e.g., split-cache line accesses which have been abused to leak data from load ports [216, 215]).

7.5.3 Phase $\mathcal{P}3$: Gadget-based secret transmission

The key challenge in the third LVI phase is to identify an exploitable code "gadget" exhibiting incorrect transient behavior over poisoned data forwarded from a faulting load micro-op in the previous phase. In contrast to all prior Meltdown-type attacks, LVI attackers do *not* control the instructions surrounding the faulting load as the load runs entirely in the victim domain. We, therefore, propose a gadget-oriented exploitation methodology closely mirroring the classification from the Spectre world [36, 146].

Disclosure gadget. A first type of gadget, akin Spectre-PHT-style information disclosure, encodes victim secrets in the instructions immediately following the faulting load (cf. Listing 7.1). The gadget encodes secrets in conditional control flow or data accesses. Importantly, however, this gadget does *not* need to be secret-dependent. Hence, LVI can even target side-channel resistant constant-time code [71]. That is, at the architectural level, the victim code only dereferences known, non-confidential values when evaluating branch conditions or array indices. At the microarchitectural level, however, the faulting load in the second LVI phase causes the known value to be transiently replaced. As a result of this "transient remapping" primitive, the gadget instructions may now inadvertently leak secret values that were brought into the targeted microarchitectural buffer during prior victim execution.

Control-flow hijack gadget. A second and more powerful type of LVI gadgets, mirroring Spectre-BTB-style branch-target injection, exploits indirect branches in the victim code. In this case, the attacker's goal is not to disclose forwarded values, but instead to abuse them as a transient control-flow hijacking primitive. That is, when dereferencing a function pointer (`call`, `jmp`) or loading a return address from the stack (`ret`), the faulting load micro-op in the victim code may incorrectly pick up attacker-controlled values from the poisoned microarchitectural buffer. This essentially enables the attacker to arbitrarily redirect the victim's transient control flow to selected second-stage code gadgets found in the victim address space. Adopting established techniques from jump-oriented [25] and return-oriented programming (ROP) [228], second-stage gadgets can further be chained together to compose arbitrary transient instruction sequences. Akin traditional memory-safety exploits, attackers may also leverage "stack pivoting" techniques to transiently point the victim stack to an attacker-controlled memory region.

Although they share similar goals and exploitation methodologies, LVI-based control-flow hijacking should be regarded as a *complementary* threat compared to Spectre-style branch-target injection. Indeed, LVI only manifests *after* the victim attempts to fetch the architectural branch target, whereas Spectre abuses speculative execution *before* the actual branch outcome is determined. Hence, the CPU may first (correctly or incorrectly) predict transient control flow based on the history accumulated in the BTB and RSB, until the victim execution later attempts to verify the speculation by comparing the actual branch-target address loaded from application memory. At this point, LVI kicks in since the faulting load micro-op yields an incorrect attacker-controlled value and erroneously redirects the transient instruction stream to a poisoned branch-target address.

LVI-based control-flow hijack gadgets can be as little as a single x86 `ret` instruction, making this case extremely dangerous. As explained in Section 7.9, fully mitigating LVI requires blacklisting all indirect branch instructions and emulating them with equivalent serialized instruction sequences.

Widening the transient window. A final challenge is that, unlike traditional fault-injection attacks that cause persistent bit flips at the architectural level [140, 241, 188], LVI attackers can only disturb victim computations for a limited time interval before the CPU eventually catches up, detects the fault, and aborts transient execution. This implies that there is only a limited "transient window" in which the victim inadvertently computes on the poisoned load values, and all required gadget instructions need to complete within this window to transmit secrets. The transient window is ultimately bounded by the size of the processor's reorder buffer [261].

Naturally, widening the transient window is a requirement common to all transient-execution attacks. Therefore, we can leverage techniques known from prior Spectre attacks [39, 148, 167]. Common techniques include, e.g., flushing selected victim addresses or PTEs from the CPU cache.

Summary. To summarize, we construct LVI attacks with the three phases $\mathcal{P}1$ (poisoning), $\mathcal{P}2$ (provoking injection), $\mathcal{P}3$ (transmission). For each of the phases, we have different instantiations, based on the specific environment, hardware, and attacker capabilities. We now discuss gadgets in Section 7.6 and, subsequently, practical LVI attacks on SGX in Section 7.7.

7.6 LVI Taxonomy and gadget exploitation

We want to emphasize that LVI represents an entirely new *class* of attack techniques. Building on the (extended) transient-execution attack taxonomy by Canella et al. [36], we propose an unambiguous naming scheme and multi-level classification tree to reason about and distinguish LVI variants in Appendix E.1.

In the following, we overview the leaves of our classification tree by introducing the main LVI variants exploiting different microarchitectural injection sources (cf. Table 7.1). Given the particular relevance of LVI to Intel SGX, we especially focus on enclave adversaries but also include a discussion on gadget requirements and potential applicability to other environments.

7.6.1 LVI-L1D: L1 data cache injection

In this section, we contribute an innovative "reverse Foreshadow" injection-based exploitation methodology for SGX attackers. Essentially, LVI-L1D can best be regarded as a *transient page-remapping* primitive allowing to arbitrarily replace the outcome of *any* legitimate enclave load value (e.g., a return address on the stack) with any data currently residing in the L1D cache and sharing the same virtual page offset.

Microarchitectural poisoning. An "L1 terminal fault" (L1TF) occurs when the CPU prematurely early-outs address translation when a PTE has the present bit cleared or a reserved bit set [249, 271]. A special type of L1TF may also occur for SGX EPCM page faults if the untrusted PTE contains a rogue physical page number [249, 107]. In our LVI-L1D attack, the root attacker replaces the PPN field in the targeted untrusted PTE, before entering or resuming the victim enclave. If the enclave dereferences the targeted location, SGX raises an EPCM page fault. However, before the fault is architecturally raised, the poisoned PPN is sent to the L1D cache. If a cache hit occurs at the rogue physical address (composed of the poisoned PPN and the page offset specified by the load operation), illegal values are "injected" into the victim's transient data stream.

Gadget requirements. LVI-L1D works on processors vulnerable to Foreshadow, but with patched microcode, *i.e.*, not on more recent silicon-resistant CPUs [107]. The $\mathcal{P}1$ gadget, a load or store, brings secrets or attacker-controlled data into the L1D cache. The $\mathcal{P}2$ gadget is a faulting or assisted memory load. The $\mathcal{P}3$ gadget creates a side channel from the transient domain, or it redirects control flow based on the injected data (e.g., x86 `call` or `ret`), ultimately also leading to the execution of an attacker-chosen $\mathcal{P}3$ gadget. The addresses in both memory operations must have the same page offset (*i.e.*, lowest 12 virtual address bits). This is not a limiting factor since L1D can hold 32 KiB of data, allowing the three gadgets ($\mathcal{P}1$, $\mathcal{P}2$, $\mathcal{P}3$) to be far apart in the enclaved execution. Similar to architectural memory-safety SGX attacks [254], we found that high degrees

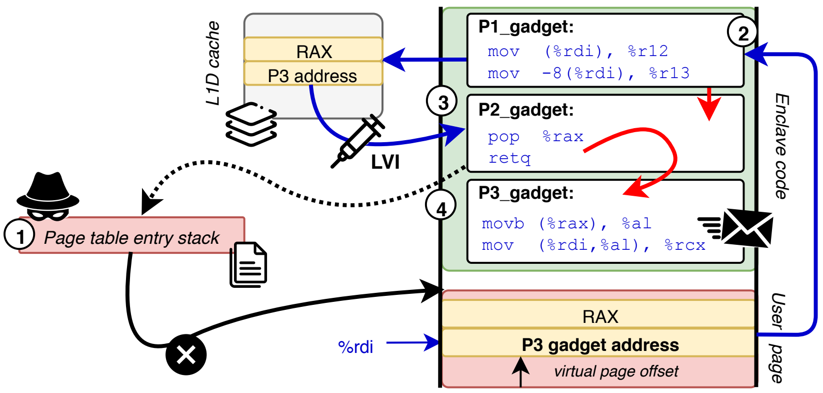


Figure 7.5: Transient control-flow hijacking using LVI-L1D: (1) the enclave's stack PTE is remapped to a user page outside the enclave; (2) a $\mathcal{P}1$ gadget inside the enclave loads attacker-controlled data into L1D; (3) a $\mathcal{P}2$ gadget pops trusted data (return address) from the enclave stack, leading to faulting loads which are transiently served with poisoned data from L1D; (4) the enclave's transient execution continues at an attacker-chosen $\mathcal{P}3$ gadget encoding arbitrary secrets in the microarchitectural CPU state.

of attacker control are often provided by enclave entry and exit code gadgets copying user data to or from chosen addresses outside the enclave.

Current microcode flushes L1D on enclave entry and exit, and SMT is recommended to be disabled [107]. We empirically confirmed that if SMT is enabled, no $\mathcal{P}1$ gadget is required and that on outdated microcode, L1D can trivially be poisoned before enclave entry.

Gadget exploitation. Figure 7.5 illustrates LVI-L1D hijacking return control flow in a minimal enclave. First, the attacker uses a page fault controlled-channel [277] or SGX-Step [257] to precisely advance the enclaved execution to right before the desired $\mathcal{P}1$ gadget. Next, the attacker sets up the malicious memory mapping ① by changing the PPN of the enclave stack page to a user-controlled page. The enclave then executes a $\mathcal{P}1$ gadget ② accessing the user page and loading attacker-controlled data into the L1D cache (e.g., when invoking `memcpy` to copy parameters into the enclave). Next, the enclave executes the $\mathcal{P}2$ gadget ③ which pops some data plus a return address from the enclave stack. For address resolution, the CPU first walks the untrusted page tables leading to the rogue PPN to be forwarded to L1D. Since the prior $\mathcal{P}1$ gadget ensured that data is indeed present in L1D at the required address, a cache hit occurs, and the poisoned data (including the return address) is served to the dependent transient micro-ops. Now, execution transiently continues at the attacker-chosen $\mathcal{P}3$ gadget ④ residing at an arbitrary location inside the enclave. The $\mathcal{P}3$ gadget encodes arbitrary secrets into the microarchitectural state before the CPU resolves the EPCM memory accesses, unrolls transient execution, and raises a page fault.

Note that for clarity, we focused on hijacking `ret` control flow in the above example, but we also demonstrated successful LVI attacks for `jmp` and `call` indirect control-flow instructions. We observe that large or repeated $\mathcal{P}1$ loads enable attackers to setup a fake "transient stack" in L1D to repeatedly inject illegal values for consecutive enclave stack loads (`pop-ret` sequences). Much like in architectural ROP code re-use attacks [228], we experimentally confirmed that attackers may chain together multiple $\mathcal{P}3$ gadgets to compose arbitrary transient computations. LVI attackers are only limited by the size of the transient window (cf. Section 7.5.3).

Applicability to non-SGX environments. We carefully considered whether cross-process or virtual machine Foreshadow variants [271] may also be reversely exploited through an injection-based LVI methodology. However, we concluded that these variants are already properly prevented by the recommended PTE inversion [46] countermeasure, which has been widely deployed in all major OSs (cf. Appendix E.1).

7.6.2 LVI-SB, LVI-LFB, and LVI-LP: Buffer and port injection

LVI-SB applies an injection-based methodology to reversely exploit store buffer leakage. The recent Fallout [35] attack revealed how faulting or assisted loads can pick up SB data if the page offset of the load (least-significant 12 virtual address bits) matches with that of a recent outstanding store. Similarly, LVI-LFB and LVI-LP inject from the line-fill buffer and load ports, respectively, which were exploited for data leakage in the recent RIDL [216] and ZombieLoad [223] attacks.

Gadget requirements. In response to Fallout, RIDL, and ZombieLoad, recent Intel microcode updates now overwrite SB, LFB, and LP entries on every enclave and process context switch [108]. Hence, to reversely exploit SB, LFP, or LP leakage, we first require a $\mathcal{P}1$ gadget to bring interesting data (e.g., secrets or attacker-controlled addresses) into the appropriate buffer. Next, we need a $\mathcal{P}2$ gadget consisting of a trusted load operation which can be faulted or assisted, followed by a $\mathcal{P}3$ gadget creating a side channel for data transmission

```
1 ; %rbx: user-controlled argument ptr (outside enclave)
2 sgx_my_sum_bridge:
3  ...
4  call my_sum          ; compute 0x10(%rbx) + 0x8(%rbx)
5  mov %rax,(%rbx)      ; P1: store sum to user address
6  xor %eax,%eax
7  pop %rbx
8  ret                  ; P2: load from trusted stack
```

Listing 7.2: Intel `edger8r`-generated code snippet with LVI-SB gadget.

or control flow redirection. For LVI-SB, we further require that the store and load addresses in $\mathcal{P}1$ and $\mathcal{P}2$ share the same page offset and are sufficiently close, such that the injected data in $\mathcal{P}1$ has not yet been drained from the store buffer. Alternatively, for LVI-LFB and LVI-LP, attackers may resort to injecting poisoned data from a sibling logical core, as LFB and LP are competitively shared between SMT cores [223, 108].

Gadget exploitation. We found that LVI-SB can be a particularly powerful primitive, given the prevalence of store operations closely followed by a return or indirect call. We illustrate this point in Listing 7.2 with trusted proxy bridge code that is automatically generated by Intel's `edger8r` tool of the official SGX SDK [116]. The `edger8r`-generated bridge code is responsible for transparently verifying and copying user arguments to and from enclave memory. The omitted code verifies that the untrusted argument pointer, which is also used to pass the result, lies outside the enclave [254].

An attacker can interrupt the enclave after line 4, clear the supervisor or accessed bit for the enclave stack, and resume the enclave. As the `edger8r` bridge code solely verifies that the attacker-provided argument pointer lies outside the enclave, it provides the attacker with full control over the lower 12 bits of the store address ($\mathcal{P}1$). When the enclave code returns at line 8, the control flow is redirected to the attacker-injected location, as the faulting or assisted `ret` ($\mathcal{P}2$) incorrectly picks up the value from the SB (which in this case is the sum of two attacker-provided arguments). Similar to LVI-L1D (Fig. 7.5), an attacker can encode arbitrary enclave secrets by chaining together one or more $\mathcal{P}3$ gadgets in the victim enclave code.

Finally, note that LVI is *not* limited to control flow redirection as secrets may also be encoded directly in the data flow through a combined $\mathcal{P}2$-$\mathcal{P}3$ gadget (e.g., by means of a double-pointer dereference as illustrated in the toy example of Listing 7.1).

Applicability to non-SGX environments. Importantly, in contrast to LVI-L1D above, SB, LFB, and LP leakage does *not* necessarily require adversarial manipulation of PTEs, or rely on microarchitectural conditions that are specific to Intel SGX. Hence, given a suitable fault or assist primitive plus the required victim code gadgets, LVI-SB, LVI-LFB, and LVI-LP may be relevant for other contexts as well (cf. Section 7.8).

7.6.3 LVI-NULL: 0x00 dummy injection

A highly interesting special case is LVI-NULL, which is based on the observation that known Meltdown-type attacks [162, 249] commonly report a strong bias to the value zero for faulting loads. We experimentally confirmed that the latest generation of acclaimed Meltdown-resistant Intel CPUs (`RDCL_NO` [128] from Whiskey Lake onwards) merely zero-out the results of faulting load micro-ops while still passing a dummy `0x00` value to dependent transient instructions. While this nulling strategy indeed suffices to prevent Meltdown-type data leakage, we show that the ability to inject zero values in the victim's transient data stream can be dangerously exploitable. Hence, LVI-NULL reveals a fundamental shortcoming in current silicon-level mitigations, and ultimately requires more extensive changes in the way the CPU pipeline is organized.

Gadget requirements. Unlike the other LVI variants, LVI-NULL does not rely on any microarchitectural buffer to inject poisoned data, but instead directly abuses dummy `0x00` values injected from the CPU's silicon circuitry in the $\mathcal{P}1$ phase. The $\mathcal{P}2$ gadget consists of a trusted load operation that can be faulted or assisted, followed by a $\mathcal{P}3$ gadget which, when operating on the unexpected value null, creates a side channel for secret transmission or control-flow redirection.

In some scenarios, transiently replacing a trusted load micro-op with the unexpected value zero may directly lead to information disclosure, as explored in the AES-NI case study of Section 7.7.2. Moreover, LVI-NULL is especially dangerous in the common case of indirect pointer dereferences.

Gadget exploitation. While transiently computing on zero values might at first seem rather innocent, we make the key insight that zero can be regarded as a *valid* virtual address and that SGX root attackers can trivially map an arbitrary memory page at virtual address null. Using this technique, we contribute an innovative *transient null-pointer dereference* primitive that allows to hijack the result of *any* indirect pointer dereference in the victim enclave's transient domain.

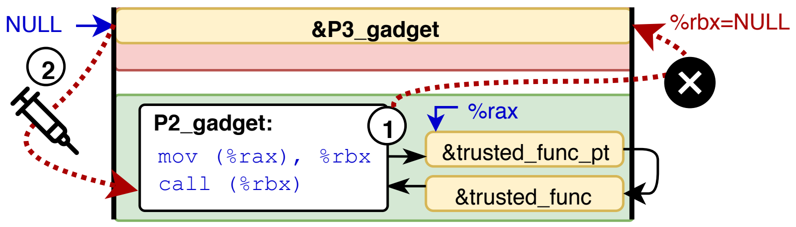


Figure 7.6: Transient control-flow hijacking using LVI-NULL: (1) a $\mathcal{P}2$ gadget inside the enclave dereferences a function pointer-to-pointer, leading to a faulting load which forwards the dummy value null; (2) the following indirect call transiently dereferences the attacker-controlled null page outside the enclave, causing execution to continue at an attacker-chosen $\mathcal{P}3$ gadget address.

We first consider the case of a data pointer stored in trusted memory, e.g., as a local variable on the stack. After revoking access rights on the respective enclave memory page, loading the pointer forces its value to zero, causing any following dereferences in the transient domain to read attacker-controlled data via the null page. This serves as a powerful "transient pointer-value hijacking" primitive to inject arbitrary data in a victim enclaved execution, which can be subsequently used in a $\mathcal{P}3$ gadget to disclose secrets or redirect control flow.

Figure 7.6 illustrates how the above technique can furthermore be exploited to arbitrarily hijack transient control flow in the case of function pointer-to-pointer dereferences, e.g., a function pointer in a heap object. The first dereference yields zero, and the actual function address is thereafter retrieved via the attacker-controlled null page. For the simpler case of single-level function pointers, we experimentally found that transient control flow cannot be directly redirected to the zero address outside the enclave, which is in line with architectural restrictions imposed by Intel SGX [47]. However, adversaries might load the relocatable enclave image at virtual address null. We, therefore, recommend that the first page is marked as non-executable or that a short infinite loop is included at the base of every enclave image to effectively "trap" any transient control flow redirections to virtual address null.

Finally, a special case is loading a stack pointer. Listing 7.3 shows a trusted code snippet from the Intel SGX SDK [116] to restore the enclave execution context when returning from an untrusted function.[1] An attacker can interrupt the victim code right before line 3, and revoke access rights on the trusted stack page used by the enclave entry code. After resuming the enclave, the victim

[1] Note that we also found similar, potentially exploitable gadgets in the `rsp-rbp` function epilogues emitted by popular compilers such as `gcc`.

```
1 asm_oret: ; (linux-sgx/sdk/trts/linux/trts_pic.S#L454)
2   ...
3   mov    0x58(%rsp),%rbp        ; %rbp <- NULL
4   ...
5   mov    %rbp,%rsp              ; %rsp <- NULL
6   pop    %rbp                   ; %rbp <- *(NULL)
7   ret                           ; %rip <- *(NULL+8)
```

Listing 7.3: LVI-NULL stack hijack gadget in Intel SGX SDK.

then page faults at line 3. However, the transient execution first continues with a zeroed `%rbp` register, which eventually gets written to the `%rsp` stack pointer register at line 5. Crucially, at this point, all subsequent `pop` and `ret` transient instructions dereference the attacker-controlled memory page mapped at virtual address null. This stack pointer zeroing primitive essentially allows LVI-NULL attackers to setup an arbitrary fake transient "shadow stack" at address null. We experimentally validated that this technique can furthermore be abused to mount a full transient ROP [228] attack by chaining together multiple subsequent `pop-ret` gadgets.

Applicability to non-SGX environments. LVI-NULL does not exploit any microarchitectural properties that are specific to Intel SGX, and may apply to other environments as well. However, we note that exploitation may be hindered by various architectural and software-level defensive measures that are in place to harden against well-known architectural null pointer dereference bugs. Some Linux distributions do not allow to map virtual address zero in user space. Furthermore, recent x86 Supervisor Mode Access Prevention (SMAP) and Supervisor Mode Execution Prevention (SMEP) architectural features further prohibit respectively user-space data and code pointer dereferences in kernel mode. SMAP and SMEP have been shown to also hold in the microarchitectural transient domain [115, 36].

7.7 LVI case studies on Intel SGX

7.7.1 Gadget in Intel's quoting enclave

In this section, we show that exploitable LVI gadgets may occur in real-world software. We analyze Intel's trusted quoting enclave (QE), which has been widely studied in previous transient-execution research [249, 39, 223] to dismantle remote attestation guarantees in the Intel SGX ecosystem. As a result, the

```
1 __intel_avx_rep_memcpy: ; libirc_2.4/efi2/libirc.a
2   ...                   ; P1: store to user address
3   vmovups %xmm0,-0x10(%rdi,%rcx,1)
4   ...
5   pop     %r12          ; P2: load from trusted stack
6   ret
```

Listing 7.4: LVI gadget in SGX SDK `intel_fast_memcpy` used in QE.

QE trusted codebase has been thoroughly vetted and hardened against all known Meltdown-type and Spectre-type attacks by manually inserting `lfence` instructions after potentially mispredicted branches, as well as flushing leaky microarchitectural buffers on every enclave entry and exit.

Gadget description. We started from the observation that most LVI variants first require a $\mathcal{P}1$ load-store gadget with an attacker-controlled address and data, followed by a faulting or assisted $\mathcal{P}2$ load that picks up the poisoned data. Similar to the `edger8r` gadget discussed in Section 7.6.2, we therefore focused our manual code review on pointer arguments which are passed to copy input and output data via untrusted memory outside the enclave [254]. Particularly, we found that QE securely verifies that the output pointer to hold the resulting quote falls outside the enclave while leaving the base address in unprotected memory under attacker control. An Intel SGX quote is composed of various metadata fields, followed by the asymmetric signature (cf. Appendix E.2). After computing the signature, but before erasing the EPID private key from enclave memory, QE invokes `memcpy` to copy the corresponding quote metadata fields from trusted stack memory to the output buffer outside the enclave. Crucially, we found that as part of the last metadata fields, a 64-byte attacker-controlled `report_data` value is written to the attacker-provided output pointer.

We reverse engineered the proprietary `intel_fast_memcpy` function used in QE and found that in this case, the quote is outputted using 128-bit vector instructions. Listing 7.4 provides the corresponding assembly code snippet, where the final 128-bit store at line 3 (including 12 bytes of attacker data) is closely followed by a `pop` and `ret` instruction sequence at lines 5-6 when returning from the `memcpy` invocation. This forms an exploitable LVI-SB transient control-flow hijacking gadget: the `vmovups` instruction ($\mathcal{P}1$) first fills the store buffer with user data at a user-controlled page offset aligned with the return address on the enclave stack, and closely afterwards the faulting or assisted `ret` instruction ($\mathcal{P}2$) incorrectly picks up the poisoned user data. The attacker now succeeded to redirect transient control flow to an arbitrary $\mathcal{P}3$ gadget address in the enclave code, which may subsequently lead to QE private key disclosure [39]. Note that when transiently executing the $\mathcal{P}3$ gadget, the attacker also controls the value of the `%r12` register popped at line 5 (which can be injected via the prior stores similarly to the return address). We further remark that Listing 7.4 is not limited to LVI-SB, since the store data may also have been committed from the store buffer to the L1 cache and subsequently picked up using LVI-L1D.

The Intel SGX SDK [116] randomizes the 11 least significant bits of the stack pointer on enclave entry. However, as return addresses are aligned, the entropy is only 7 bits, resulting on average in a correct alignment in 1 out of every 128 enclave entries when fixing the store address in $\mathcal{P}1$.

Experimental results. We validate the exploitability and success rate of the above assembly code using a benchmark enclave on an i7-8650U with the latest microcode `0xb4`. We inject both the return address and the value popped into `%r12` via the store buffer. For $\mathcal{P}3$, we can use the poisoned value in `%r12` to transmit data over an address outside the enclave. We ensure that the code in Listing 7.4 is page aligned to interrupt the victim enclave using a controlled-channel attack [277]. Before resuming the victim, we clear the user-accessible bit for the enclave stack. Additionally, to extend the transient window, we inserted a memory access which misses the cache before line 3.

In the first experiment, we disable stack randomization in the victim enclave to reliably quantify the success rate of the attack in the ideal case. LVI works very reliably, picking up the injected values 99 453 times out of 100 000 runs. With on average 9090 tries per second, we achieve an error-free transmission rate of 9.04 kB/s for our disclosure gadget.

In the second experiment, we simulate the full attack environment including stack randomization. As expected, the success rate drops by an average factor of 128. The injected return address is picked up 776 times out of 100 000 runs, leading to a transmission rate of 70.54 B/s. We did not reproduce this attack against Intel's officially signed quoting enclave, as we found it especially challenging to debug the attack for production QE binaries and to locate $\mathcal{P}3$ gadgets that fit within the limited transient window without excessive TLB misses. However, we believe that our experiments showcased all the required primitives to break Intel SGX's remote attestation guarantees, as demonstrated before by SGXPectre [39] and Foreshadow [249]. In response to our findings, Intel will harden all architectural enclaves with full LVI software mitigations (cf. Section 7.9) so as to restore trust and initiate TCB recovery for the SGX ecosystem [122].

7.7.2 Transient fault attack on AES-NI

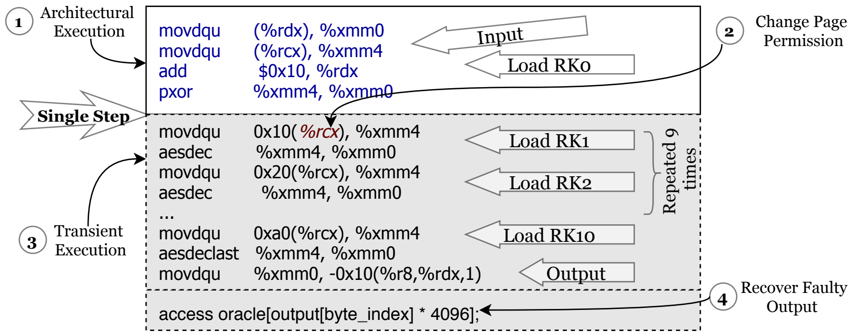
In this case study, we show that LVI-NULL can be exploited to perform a cryptographic fault attack [241, 188] on Intel's constant-time AES-NI hardware extension. We exploit that a privileged SGX attacker can induce faulty all-zero round keys into the transient data stream of a minimal AES-NI enclave. After the fault, the output of the decryption carries a faulty plaintext in the transient domain. To simplify the attack, we consider a known-ciphertext scenario and we assume a side channel in the post-processing which allows to recover the faulty decryption output from the transient domain. Note that prior research [261] on Spectre-type attacks has shown that transient execution may fit a significant number of AES-NI decryptions (over 100 rounds on modern Intel processors).

Intel AES-NI [90] is implemented as an x86 vector extension. The `aesdec` and `aesdeclast` instructions perform one round of AES on a 128-bit register using the round key provided in the first register operand. Round keys are stored in trusted memory and, depending on the available registers and the AES-NI software implementation, the key schedule is either preloaded or consulted at the start of each round. In our case study, we assume that round keys are securely fetched from trusted enclave memory before each `aesdec` instruction.

Attack outline. Figure 7.7 illustrates the different phases in our transient fault injection attack on AES-NI:

1. We use SGX-Step [257] to precisely interrupt the victim enclave after executing only the initial round of AES.
2. The root attacker clears the user-accessible bit on the memory page containing the round keys.
3. The attacker resumes the enclave, leading to a page fault when loading the next round keys from trusted memory. We abuse theses faulting load as $\mathcal{P}2$ gadgets which transiently forward dummy (all-zero) round keys to the remaining `aesdec` instructions. Note that we do not need a $\mathcal{P}1$ gadget, as the CPU itself is responsible for zero-injection.
4. Finally, we use a $\mathcal{P}3$ disclosure gadget after the decryption.

By forcing all but the first AES round key to zero, our attack essentially causes the victim enclave to compute a round-reduced AES in the transient domain. To recover the first round key, and hence the full AES key, the attacker can simply feed the faulty output plaintext recovered from the transient domain to an inverse AES function with all keys set to zero. This results in an output that holds the secret AES first round key, `xor`-ed with the (known) ciphertext.



```
movdqu      (%rdx), %xmm0
movdqu      (%rcx), %xmm4
add          $0x10, %rdx
pxor        %xmm4, %xmm0
movdqu      0x10(%rcx), %xmm4
aesdec       %xmm4, %xmm0
movdqu      0x20(%rcx), %xmm4
aesdec        %xmm4, %xmm0
...
movdqu      0xa0(%rcx), %xmm4
aesdeclast  %xmm4, %xmm0
movdqu      %xmm0, -0x10(%r8,%rdx,1)
access oracle[output[byte_index] * 4096];
```

Figure 7.7: Overview of the AES-NI fault attack: (1) the victim architecturally executes the initial AES round, which `xor`s the input with round key 0; (2) access rights on the memory page holding the key schedule are revoked; (3) upon the next key access ($\mathcal{P}2$), the enclave suffers a page fault, causing the CPU to transiently execute the next 10 AES rounds with zeroed round keys; (4) finally the faulty output is encoded ($\mathcal{P}3$) through a cache-based side channel.

Experimental results. We run the attack for 100 different AES keys on a Core i9-9900K with `RDCL_NO` and the latest microcode `0xae`. For each experiment, we run the attack to recover 10 candidates for each byte of the faulty output. On average, each recovered key candidate matches the expected faulty output 83 % of the time. Using majority vote for the 10 candidates, we recover the correct output for an average of 15.61 out of 16 bytes of the AES block, indicating that the output matches the attack model with 97 % accuracy. The attack takes on average 25.94 s (including enclave creation time) and requires 246 707 executions of the AES function.

For post-processing, we modified an AES implementation to zero out the round keys after the first round. We successfully recovered the secret round-zero key using any of the recovered faulty plaintext outputs to the inverse encryption function.

7.8 LVI in other contexts

7.8.1 User-to-kernel

The main challenge in a user-to-kernel LVI attack scenario is to provoke faulting or assisted loads during kernel execution. As any application, the kernel may encounter page faults or microcode assists, e.g., due to demand paging via the extended page tables setup by the hypervisor, or when swapping out supervisor heap memory pages in the Windows kernel [213]. We do not investigate the more straightforward scenario where the kernel encounters a page fault when accessing a user-space address, as in this case the user already architecturally controls the value read by the kernel.

Experimental setup. We focus on exploiting LVI-SB via microcode assists for setting the accessed bit in supervisor PTEs. In our case study, we execute the $\mathcal{P}1$ poisoning phase directly in user space by abusing that current microcode mitigations only flush the store buffer on kernel exit to prevent leakage [35, 108]. As the store buffer is not drained on kernel entry, it can be filled with attacker-chosen values by writing to arbitrary user-accessible addresses before performing the system call. Note that, alternatively, the store buffer could also be filled during kernel execution by abusing a selected $\mathcal{P}1$ gadget, similar to our SGX attacks.

In the $\mathcal{P}2$ phase, the attacker needs to trigger a faulting or assisted load micro-op in the kernel. In our proof-of-concept, we assume that the targeted supervisor page is swappable, as is the case for Windows kernel heap objects [213], but to the best of our knowledge not for the Linux kernel. In order to repeatedly execute the same experiment and assess the overall success rate, we simulate the workings of the page-replacement algorithm by means of a small kernel module, which artificially clears the accessed bit on the targeted kernel page.

As we only want to demonstrate the building blocks of the attack, we did not actively look for real-world gadgets in the kernel. For our evaluation, we manually added a simple $\mathcal{P}3$ disclosure gadget, which, similar to a Spectre gadget, indexes a shared memory region based on a previously loaded value as follows: `array[(*kernel_pt) * 4096]`. In case the trusted load on `kernel_pt` requires a microcode assist, the value written by the user-space attacker will be transiently injected from the store buffer and subsequently encoded into the CPU cache.

Experimental results. We evaluated LVI-SB on an Intel Core i7-8650U with Linux kernel 5.0. On average, 1 out of every 7739 ($n = 100\,000$) assisted loads in the kernel use the injected value from the store buffer instead of the architecturally correct value. For our non-optimized proof-of-concept, this results on average in a successfully injected value into the kernel execution every 6.5 s. One of the reasons for this low success rate is the context switch between $\mathcal{P}1$ and $\mathcal{P}2$, which reduces the probability that the attacker's value is still outstanding in the store buffer [35]. We verified this by evaluating the injection rate without a context switch, *i.e.*, if the store buffer is poisoned via a suitable $\mathcal{P}1$ gadget in the kernel. In this case, on average, 1 out of every 8 (n = 100 000) assisted loads in the kernel use the injected value.

7.8.2 Cross-process

We now demonstrate how LVI-LFB may inject poisoned data from a concurrently running attacker process.

Experimental setup. For the poisoning phase $\mathcal{P}1$, we assume that the attacker and the victim are co-located on the same physical CPU core [223, 216, 249]. The attacker directly poisons the line-fill buffer by writing or reading values to or from the memory subsystem. To ensure that the values travel through the fill buffer, the attacker simply flushes the accessed values using the unprivileged `cflflush` instruction. In case SMT is disabled, the adversary would have to find a suitable $\mathcal{P}1$ gadget that processes untrusted, attacker-controlled arguments in the victim code, similar to our SGX attacks.

In our proof-of-concept, the victim application loads a value from a trusted shared-memory location, e.g., a shared library. As shown by Schwarz et al. [223], Windows periodically clears the PTE accessed bit, which may cause microcode assists for trusted loads in the victim process. The attacker flushes the targeted shared-memory location from the cache, again using `clflush`, to ensure that the victim's assisted load $\mathcal{P}2$ forwards incorrect values from the line-fill buffer [223, 216] instead of the trusted shared-memory content.

Experimental results. We evaluated the success rate of the attack on an Intel i7-8650U with Linux kernel 5.0. We used the same software construct as in the kernel attack for the transmission phase $\mathcal{P}3$. Both attacker and victim run on the same physical core but different logical cores. On average, 1 out of 101 (n = 100 000) assisted loads uses the value injected by the attacker, resulting in an injection probability of nearly 1 %. With on average 1122 tries per second, we achieve a transmission rate of 11.11 B/s for our disclosure gadget.

7.9 Discussion and mitigations

In this section, we discuss both long-term silicon mitigations to rule out LVI at the processor design level, as well as compiler-based software workarounds that need to be deployed on the short-term to mitigate LVI on existing systems.

7.9.1 Eradicating LVI at the hardware design level

The root cause of LVI needs to be ultimately addressed through silicon-level design changes in future processors. Particularly, to rule out LVI, the hardware has to ensure that no illegal data flows from faulting or assisted load micro-ops exist at the microarchitectural level. That is, no transient computations depending on a faulting or assisted instruction are allowed. We believe this is already the behavior in certain ARM and AMD processors, where a faulting load does not forward any data [13]. Notably, we showed in Section 7.6.3 that it does *not* suffice to merely zero out the forwarded value, as is the case in the latest generation of acclaimed Meltdown-resistant Intel processors enumerating `RDCL_NO` [128].

7.9.2 A generic software workaround

Silicon-level design changes take considerable time, and at least for SGX enclaves a short-term solution is needed to mitigate LVI on current, widely deployed systems. In contrast to previous Meltdown-type attacks, merely flushing microarchitectural buffers before or after victim execution is *not* sufficient to defend against our novel, gadget-based LVI attack techniques. Instead, we propose a software-based mitigation approach which inserts explicit `lfence` speculation barriers to serialize the processor pipeline after every vulnerable load instruction. The `lfence` instruction is guaranteed by Intel to halt transient execution until all prior instructions have completed [128]. Hence, inserting an `lfence` after every potentially faulting or assisted load micro-op guarantees that the value forwarded from the load operation is not an injected value but the architecturally correct one. Relating to the general attack scheme of Fig. 7.3, we introduce an `lfence` instruction in between phases $\mathcal{P}2$ and $\mathcal{P}3$ to inhibit any incorrect transient forwarding by the processor. Crucially, in contrast to existing Spectre-PHT compiler mitigations [128, 36] which only insert `lfence` barriers after potentially mispredicted conditional jump instructions, fully mitigating LVI requires stalling the processor pipeline after potentially *every* explicit as well as implicit memory-load operation.

Explicit memory loads, *i.e.*, instructions with a memory address as input parameter, can be protected straightforwardly. A compiler, or even a binary rewriter [53], can add an `lfence` instruction to ensure that any dependent operations can only be executed after the load instruction has successfully retired. However, some x86 instructions also include *implicit* memory load micro-ops which cannot be mitigated in this way. For instance, indirect branches and the `ret` instruction load an address from the stack and immediately redirect control flow to the loaded, possibly injected value. As the faulting or assisted

Table 7.2: Indirect branch instruction emulations needed to prevent LVI and whether or not they require a scratch register which can be clobbered.

Instruction	Possible emulation	Clobber-free
`ret`	`pop %reg; lfence; jmp *%reg`	✗
`ret`	`not (%rsp); not (%rsp); lfence; ret`	✓
`jmp (mem)`	`mov (mem),%reg; lfence; jmp *%reg`	✗
`call (mem)`	`mov (mem),%reg; lfence; call *%reg`	✗

load micro-op in this case forms part of a larger ISA-level instruction, there is no possibility to add an `lfence` barrier between the memory load ($\mathcal{P}2$) and the control-flow redirection ($\mathcal{P}3$). Table 7.2 shows how indirect branch instructions have to be blacklisted and emulated through an equivalent sequence of two or more instructions, including an `lfence` after the formerly implicit memory load. Notably, as some of these emulation sequences clobber scratch registers, LVI mitigations for indirect branches cannot be trivially implemented using binary rewriting techniques and should preferably be implemented in the compiler back-end, before the register allocation stage.

Evaluation of our prototype solution. We initially implemented a prototypical compiler mitigation using LLVM [164] (8.3.0) and applied it to a recent OpenSSL [199] version (1.1.1d) with default configuration. We chose OpenSSL as it serves as the base of the official Intel SGX-SSL library [121] allowing to approximate the expected performance impact of the proposed mitigations. Our proof-of-concept mitigation tool allows to augment the building process of arbitrary C code by first instrumenting the compiler to emit LLVM intermediate code, adding the necessary `lfence` instructions after every explicit memory load, and finally proceeding to compile the modified file to an executable. Our prototype tool cannot mitigate loads which are not visible at the LLVM intermediate representation, e.g., the x86 back-end may introduce loads for registers spilled onto the stack after register allocation. To deal with assembly source files, our tool introduces an `lfence` after every `mov` operating on memory addresses. Our prototype does not mitigate all types of indirect branches, but can optionally replace `ret` instructions with the proposed emulation code, where `%r11` is used as a caller-save register that can be clobbered.

To measure the performance impact of the introduced `lfence` instructions and the `ret` emulation, we recorded the average throughput ($n = 10$) of various cryptographic primitives using OpenSSL's `speed` tool on an isolated core on an Intel i7-6700K. As shown in Fig. 7.8, the performance overhead reaches from a minimum of 0.91 % for a partial mitigation which only rewrites `ret`

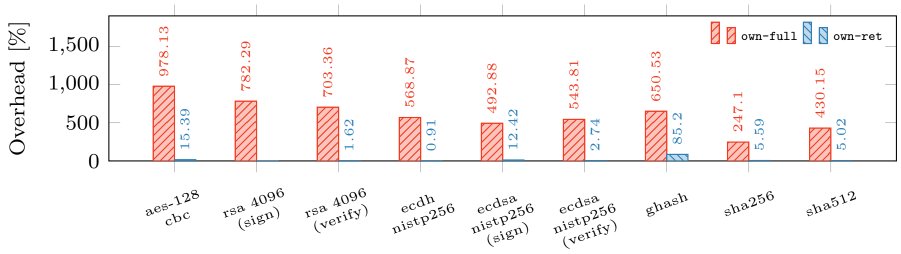


Figure 7.8: Performance overhead of our LLVM-based prototype (fence loads + ret vs. ret-only) for OpenSSL on an Intel i7-6700K CPU.

instructions to a maximum of 978.13 % for the full mitigation including `ret` emulation and load serialization. Note that for real-world deployment, the placement of `lfence` instructions should be evaluated for completeness and more optimized than in our prototype implementation. Still, our evaluation serves as an approximation of the expected performance impact of the proposed mitigations.

Evaluation of Intel's proposed mitigations. To further evaluate the overheads of more mature, production-quality implementations, we were provided with access to Intel's current compiler-based mitigation infrastructure. Hardening of existing code bases is facilitated by a generic post-compilation script that uses regular expressions to insert an `lfence` after every x86 instruction that has a load micro-op. Working exclusively at the assembly level, the script is inherently compiler-agnostic and can hence only make use of indirect branch emulation instruction sequences that do not clobber registers. In general, it is therefore recommended to first decompose indirect branches from memory using existing Spectre-BTB mitigations [126]. As not all code respects calling conventions, `ret` instructions are by default replaced with a clobber-free emulation sequence which first tests the return address, before serializing the processor pipeline and issuing the `ret` (cf. Table 7.2). We want to note that this emulation sequence still allows privileged LVI adversaries to provoke a fault or assist on the return address when leveraging a single-stepping framework like SGX-Step [257] to precisely interrupt and resume the victim enclave after the `lfence` and before the final `ret`. However, we expect that in such a case the length of the transient window would be severely restricted as `eresume` appears to be a serializing instruction itself [114]. Furthermore, as recent microcode flushes microarchitectural buffers on enclave entry, the poisoning phase would be limited to LVI-NULL. Any inadvertent transient control-flow redirections to virtual address null can be mitigated by marking the first enclave page as non-executable (cf. Section 7.6.3).

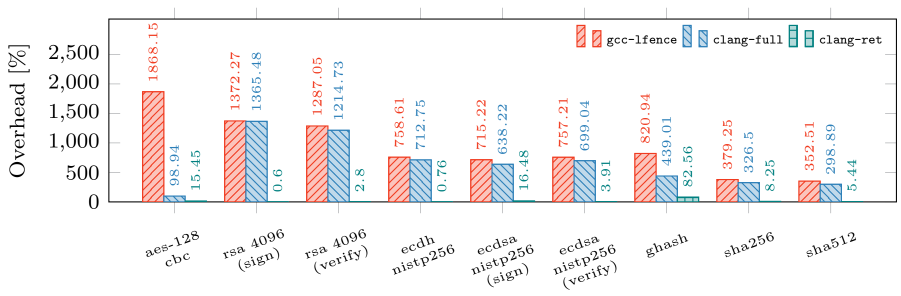


Figure 7.9: Performance overhead of Intel's mitigations for non-optimized assembler `gcc` (fence loads + ret) and optimized `clang` (fence loads + indirect branch + ret vs. ret-only) for OpenSSL on an Intel i7-6700K CPU.

Intel furthermore developed an optimized LVI mitigation pass for LLVM-based compilers. The pass operates at the LLVM intermediate representation and uses a constraint solver from integer programming to optimally insert `lfence` instructions along all paths in the control-flow graph from a load ($\mathcal{P}2$) to a transmission ($\mathcal{P}3$) gadget [21, 110]. As the pass operates at the LLVM intermediate representation, any additional loads introduced by the x86 back-end are not mitigated. We expect such implicit loads from e.g., registers that were previously spilled onto the stack to be difficult to exploit in practice, but we leave further security evaluation of the mitigations as future work. The pass also replaces indirect branches, and `ret` instructions are eliminated in an additional machine pass using a caller-save clobber register.

Figure 7.9 provides the OpenSSL evaluation for the Intel mitigations ($n = 10$). The unoptimized `gcc` post-compilation full mitigation assembly script for fencing all loads and `ret` instructions clearly incurs the highest overheads from 352.51 % to 1868.15 %, which is slightly worse than our own (incomplete) LLVM-based prototype. For the OpenSSL experiments, Intel's optimized `clang` LLVM mitigation pass for fencing loads, conditional branches, and `ret` instructions generally reduces overheads within the same order of magnitude, but more significantly in the AES case. Lastly, in line with our own prototype evaluation, smaller overheads from 2.52 % to 86.23 % are expected for a partial mitigation strategy which patches only `ret` instructions while leaving other loads and indirect branches potentially exposed to LVI attackers.

Finally, to assess expected overheads in larger and more varied applications, we evaluated Intel's mitigations on the SPEC2017 `intspeed` benchmark suite. Fig. 7.10 provides the results as executed on an isolated core on a i9-9900K

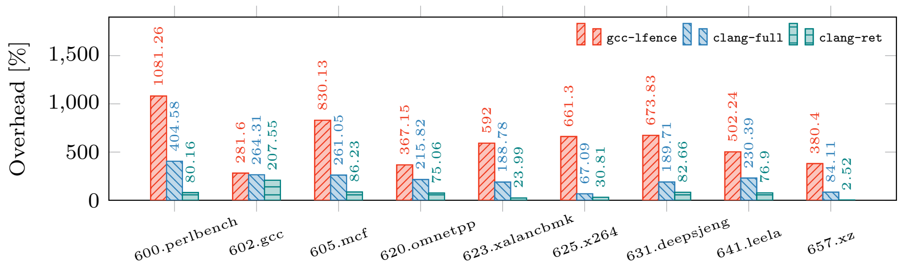


Figure 7.10: Performance overhead of Intel's mitigations for non-optimized assembler `gcc` (fence loads + ret) and optimized `clang` (fence loads + indirect branch + ret vs. ret-only) for SPEC2017 on an Intel i9-9900K CPU.

CPU, running Linux 4.18.0 with Ubuntu 18.10 ($n = 3$).[2] One clear trend is that Intel's optimized LLVM mitigation pass outperforms the naive post-compilation assembly script.

7.10 Outlook and future work

We believe that our work presents interesting opportunities for developing more efficient compiler mitigations and software hardening techniques for current, widely deployed systems.

7.10.1 Implications for transient-execution research

LVI again illustrates the constant race between attackers and defenders. With LVI, we introduced an advanced attack technique that bypasses existing software and hardware defenses. While potentially harder to exploit than previous Meltdown-type attacks, LVI shows that Meltdown-type incorrect transient forwarding effects are not as easy to fix as expected [162, 36, 278]. The main insight with LVI is that transient-execution attacks, as well as side-channel attacks, have to be considered from two viewpoints: observing and injecting data. It is not sufficient to only mitigate data leakage direction, as it was done so far, and the injection angle also needs to be considered. Hence, in addition to flushing microarchitectural buffers on context switch [108, 107], additional mitigations are required. We believe that our work has a substantial influence on

[2] Note that we had to exclude the `648.exchange2_s` benchmark program as it is written in Fortran and hence not supported by the mitigation tools.

future transient-execution attacks as new discoveries of Meltdown-type effects now need to be studied in both directions.

Although the most realistic LVI attack scenarios are secure enclaves such as Intel SGX, we demonstrated that none of the ingredients for LVI are unique to SGX and other environments can possibly be attacked similarly. We encourage future attack research to further investigate improved LVI gadget discovery and exploitation techniques in non-SGX settings, e.g., cross-process and sandboxed environments [146, 167].

An important insight for silicon mitigations is that merely zeroing out unintended data flow is insufficient to protect against LVI adversaries. At the compiler level, we expect that advanced static analysis techniques may further improve the extensive performance overheads of current `lfence`-based mitigations (cf. Section 7.9.2). Particularly, for non-control-flow hijacking gadgets, it would be desirable to serialize only those loads that are closely followed by an exploitable $\mathcal{P}3$ gadget for side-channel transmission.

7.10.2 Raising the bar for LVI exploitation

While not completely eliminated, our analysis in Section 7.6 and Appendix E.1 revealed that the LVI attack surface may be greatly reduced by certain system-level software measures in non-SGX environments. For instance, the correct sanitization of user-space pointers and the use of x86 SMAP and SMEP features in commodity OS kernels may greatly reduce the possible LVI gadget space. Furthermore, we found that certain software mitigations, which were deployed to prevent Meltdown-type data leakages, also unintentionally thwart their LVI counterparts, e.g., eager FPU switching [234] and PTE inversion [46]. LVI can also be inhibited by preventing victim loads from triggering exceptions and microcode assists. However, this may come with significant changes in system software, as e.g., PTE accessed and dirty bits must not be cleared anymore, and kernel pages must not be swapped anymore. Although such changes are possible for the OS, they are not possible for SGX, as the attacker is in control of the page tables.

As described in Section 7.9.2, Intel SGX enclaves require extensive compiler mitigations to fully defend against LVI. However, we also advocate architectural changes in the SGX design which may further help raising the bar for LVI exploitation. LVI is, for instance, facilitated by the fact that SGX enclaves share certain microarchitectural elements, such as the cache, with their host application [47, 181, 225]. Furthermore, enclaves can directly operate on untrusted memory locations passed as pointers in the shared address space [219, 254]. As a generic software hardening measure, we suggest that pointer sanitization logic [254] further restricts the attacker's control over page offset address bits for unprotected input and output buffers. To inhibit transient null-pointer dereferences in LVI-NULL exploits, we propose that microcode marks the memory page at virtual address zero as uncacheable [222, 26, 239]. Similarly, LVI-L1D could be somewhat restricted by terminating the enclave or disabling SGX altogether upon detecting a rogue PPN in the EPCM microcode checks, which can only indicate a malicious or buggy OS.

7.11 Conclusion

We presented Load Value Injection (LVI), a novel class of attack techniques allowing the direct injection of attacker data into a victim's transient data stream. LVI complements the transient-execution research landscape by turning around Meltdown-type data leakage into data injection. Our findings challenge prior views that, unlike Spectre, Meltdown threats could be eradicated straightforwardly at the operating system or hardware levels and ultimately show that future Meltdown-type attack research must also consider the injection angle.

Our proof-of-concept attacks against Intel SGX enclaves and other environments show that LVI gadgets exist and may be exploited. Existing Meltdown and Spectre defenses are orthogonal to and do not impede our novel attack techniques, such that LVI necessitates drastic changes at the compiler level. Fully mitigating LVI requires including `lfence`s after possibly *every* memory load, as well as blacklisting indirect jumps, including the ubiquitous x86 `ret` instruction. We observe extensive slowdowns of factor 2 to 19 for our prototype evaluation of this countermeasure. LVI demands research on more efficient and forward-looking mitigations on both the hardware and software levels.

Chapter 8

Conclusion

"As the level of program gets lower, these bugs will be harder and harder to detect. A well installed microcode bug will be almost impossible to detect."

— *Ken Thompson (ACM Turing award lecture, 1984)*

The need for hardware trust anchors to establish software security is universal. Recently, hardware-based Trusted Execution Environments (TEEs), such as Intel's Software Guard Extensions (SGX), have been developed as a promising new security paradigm to isolate enclaved software components directly in the processor, without having to trust the underlying operating system or hypervisor. This dissertation developed several innovative attack techniques that nuance the trust placed in TEEs in general and Intel SGX in particular.

Our research outcomes do not stand in a vacuum, but form part of a wider landscape of SGX attacks that unfolded in synergy with this PhD trajectory. We, therefore, conclude this dissertation by summarizing our results and placing them in the wider perspective of the contemporary TEE attack scene. Building upon the insights gained from the past five years of SGX attacks, this chapter then proceeds to look forward by formulating recommendations for future research and overviewing possible defense avenues and pitfalls for the next generation of fortified TEE designs.

8.1 Summary of contributions

In this thesis, we researched security limitations of TEE protection and asked which new attack vectors can be exploited by privileged adversaries. From our work, we conclude that the isolation offered by TEEs is relative and that several misconceptions existed about developing intrinsically secure enclave applications and protecting against privileged side-channel attacks.

First, in Chapter 2, we showed that the enclave software itself remains fully trusted. Our analysis uncovered a wide and reoccurring vulnerability landscape in real-world TEE runtime libraries, which hold the crucial responsibility of maintaining a secure bridge between the enclave and its untrusted environment. From this work, we conclude that the enclave interface forms a large attack surface and needs to be methodologically vetted to protect against various kinds of memory corruptions and side-channel leakages. We furthermore showed that the privileged adversary's control over the untrusted operating system offers a substantial advantage to reliably exploit such subtle interface sanitization oversights in practice.

Next, we changed focus from software to the underlying processor architecture, by developing a line of innovative side-channel attack techniques that leverage the privileged adversary's first-rate control over hardware-software interfaces. These attacks show that even if the processor safeguards confidentiality and integrity of enclave memory directly, enclaved execution is not completely opaque and various kinds of metadata, such as code and data access patterns, can still be reconstructed through side-channel analysis. Centering on the state-of-the-art Intel SGX [14, 176] TEE as a relevant case-study architecture shipped in recent Intel processors, we showed that several traditionally privileged x86 processor interfaces can be reliably abused to mount new and unexpected types of side-channel attacks. In Chapter 3, we exploited the adversary's control over untrusted page-table memory to design stealthy attack variants that can accurately reconstruct enclave memory accesses at a 4 KiB spatial granularity without provoking page faults. This work illustrates the security implications of traversing page tables in untrusted memory, even if the address translation result is afterwards verified, and defeats any defenses that are based on merely detecting page-fault events. In Chapter 4, we presented the SGX-Step framework which delegates traditionally privileged operating system powers to user space via a small kernel driver and a practical attack library. This work includes a novel APIC timer manipulation that allows to drastically increase the temporal resolution of enclave side-channel attacks. With SGX-Step we permanently refined the TEE threat landscape and enabled a new line of extremely high-resolution SGX attacks [91, 249, 256, 223, 254, 182, 6, 105, 251, 208] by showing that enclaves can be precisely interrupted exactly one instruction at a time. Finally, Chapter 5 presented a particular high-resolution attack, named Nemesis, which shows that carefully timed interrupts not only improve the temporal resolution of existing attacks, but also in themselves induce subtle microarchitectural timing leakage in the processor pipeline.

In the final chapters of this dissertation, we moved from execution metadata leakage to direct data extraction, *i.e.*, from side channels to transient-execution attacks. This work highlights that the processor itself remains trusted and that subtle oversights in the underlying microarchitectural pipeline design can undermine all of the TEE's pursued security guarantees. With Foreshadow, presented in Chapter 6, we revisited privileged page-table manipulations and contributed a novel attack technique to leak plaintext enclave secrets from the CPU cache. By extracting long-term, private attestation keys from Intel's architectural enclaves, we for the first time decisively dismantled security guarantees in the SGX ecosystem. This research led to Trusted Computing Base (TCB) recovery through microcode patches for existing processors, as well as silicon-level changes in the newest generations of Intel processors. With Load Value Injection (LVI), presented in Chapter 7, we highlighted the inadequacy of existing microcode and silicon mitigations by contributing innovative, gadget-driven reverse exploitation techniques for prior microarchitectural data leakage sources. Our findings challenge prior views that, unlike Spectre, Meltdown-type threats could be eradicated straightforwardly at the operating system or hardware levels [36, 278]. Instead, LVI necessitates improved silicon-level changes in future processors and SGX TCB recovery for current, widely deployed CPUs. In response to our findings, Intel developed extensive compiler mitigations that incur substantive slowdowns by serializing the processor pipeline after potentially every memory load operation, and rewriting ubiquitous x86 instructions, including `ret`.

8.2 Systematizing the SGX attack landscape

Following the launch of Intel SGX in 2015, several researchers have explored security limitations of this new enclave technology. This section overviews the past five years of SGX attacks with an explicit focus on the central theme of this thesis, *i.e.*, the increased advantages for privileged adversaries.

Table 8.1 summarizes the SGX attack scene by listing the different techniques that have been leveraged to extract secrets from enclaves, along with a characterization of the privileged x86 features they abuse plus any additional constraints they might have. In Sections 8.2.1 to 8.2.4, we first perform a vertical reading of this table by overviewing the different attack techniques and leakage

Table 8.1: Characterization of demonstrated SGX attacks in terms of the privileged x86 processor interfaces they abuse (left to right: page table, global descriptor table, interrupts, model-specific registers, control registers, performance-monitoring counters). Additional constraints (simultaneous multithreading, repetitions, shared address space) and the offered spatial resolution for side-channel observations are indicated on the right. Symbols indicate whether a feature is required (●) or optional (◐). Attacks contributed to over the course of this PhD are indicated in the highlighted rows.

	Properties / **Attack**	**x86 Interface**						**Constraints**			
		PTE	GDT	IRQ	MSR	CR0	PMC	SMT	REP	SAS	Granular
μ-arch contention	Cache priming [181, 92, 29, 225, 79]	◐	○	◐	◐	○	◐	◐	●	○	64 B
	Branch prediction [156, 59, 105]	◐	○	◐	◐	◐	◐	◐	●	○	Inst
	DRAM row buffer conflicts [263]	○	○	○	◐	◐	○	○	●	○	1-8 KiB
	False dependencies [180]	○	○	○	◐	○	○	●	●	○	4 B
	Interrupt latency [**256**, 95, 208]	◐	○	●	◐	○	○	○	●	○	Inst
	Port contention [7]	○	○	○	◐	○	○	●	●	○	μ-op
Control channel	Page faults [277]	●	○	○	○	○	○	○	○	○	4 KiB
	Page table A/D [**258**, 263]	●	○	◐	○	◐	○	◐	○	○	4 KiB
	Page table flushing [**258**]	●	○	◐	○	○	○	◐	○	○	32 KiB
	Interrupt counting [**182**]	●	○	●	○	○	○	○	○	○	Inst
	IA32 segmentation faults [**91**]	◐	●	◐	○	○	○	○	○	○	1 B-4 KiB
	Alignment faults [**254**]	○	○	●	○	●	○	○	○	○	1 B
Transient	Foreshadow L1D extraction [**249**]	●	○	◐	○	○	○	◐	○	○	–
	Data sampling [**223**, 216, 211]	●	○	◐	○	○	○	◐	◐	○	–
	Spectre [39, 148]	◐	○	◐	○	○	○	◐	◐	◐	–
	Load value injection [**251**]	●	○	◐	○	○	○	◐	◐	◐	–
Interface	Memory safety [**254**, 154, 24, 266]	◐	○	◐	○	○	○	○	◐	◐	–
	Undervolting [**188**, 137, 210]	◐	○	◐	●	○	○	◐	●	◐	–
	Off-chip memory address bus [152]	●	○	◐	○	◐	○	○	○	○	64 B

sources. Then, in Section 8.2.5, we proceed horizontally to identify tendencies across the spectrum and to draw conclusions that can lead to important insights for possible defenses.

8.2.1 Microarchitectural contention side-channel attacks

A first category of attacks observes that, although Intel SGX enclaves are strictly isolated at the architectural level, they still share various microarchitectural elements with untrusted, potentially malicious code executing on the same platform. Microarchitectural contention arises whenever these resources are competitively shared during enclave execution or not flushed on enclave exit. By competing for the same resources as the victim enclave, attackers can cause measurable timing differences in their own or the victim's execution, allowing to infer enclave-private control flow decisions or data access patterns with varying degrees of granularity. In general, such microarchitectural timing side-channel attacks have been recognized for decades already, and many of the specific leakage sources were known before SGX [71, 133, 47]. Attack research in this category has, therefore, mainly focused on demonstrating the increased advantage of privileged adversaries by either exploiting previously known leakage sources with significantly less noise and higher resolution, or devising innovative new types of contention that are specific to enclave attackers.

CPU cache. Since the public release of Intel SGX, several researchers [181, 92, 29, 225, 79] have demonstrated efficient PRIME+PROBE cache timing attacks that can accurately reconstruct enclave memory accesses at a 64-byte cache line granularity. These attacks exploit contention in either the shared CPU caches or off-core DRAM row buffers [204, 263]. Notably, one work [225] has shown that privileged adversary capabilities are not a strict requirement for PRIME+PROBE attacks on SGX, and effective last-level cache timing attacks can even be mounted from one unprivileged user-space enclave to another.

In the context of SGX-based PRIME+PROBE cache attacks, all other works [181, 92, 29, 79] have focused on a root adversary model and have developed several techniques to construct more accurate leakage. First, all these attacks reduce noise by instructing the operating system scheduler to pin the victim enclave to an isolated CPU core and possibly fix the clock by disabling dynamic frequency scaling features like TurboBoost through the processor's Model Specific Register (MSR) interface. Second, some attacks [92, 263] optionally disable the hardware prefetcher in the BIOS or through MSRs. Third, a subset of cache attacks [181, 92] abuses frequent enclave preemptions through a combination of page faults and APIC timer interrupts to sample the enclave's cache footprint at an improved temporal resolution. This technique has been demonstrated to defeat traditional side-channel hardening techniques, like software prefetching [181], and has furthermore been used to attack Intel's architectural quoting enclave which was vulnerable due to a non-constant-time cryptographic implementation [50]. Ultimately, using the SGX-Step [257] framework, such interrupt-driven cache attacks can achieve a *maximal* temporal resolution by probing the cache after every single instruction. Fourth, a perpendicular line of work [29, 79] has explored the opposite direction by developing stealthy cache attacks that proceed *without* interrupting the victim enclave at all, thereby defeating defenses [42, 229] that are based on detecting frequent enclave preemptions. These attacks exploit that the L1D cache is shared among logical CPUs and PRIME+PROBE-style contention can be induced in parallel to the victim enclave's execution using Simultaneous Multithreading (SMT) technology. Furthermore, to eliminate timing noise in the probing phase and to increase the overall accuracy of the attack, Performance Monitoring Counters (PMCs) can be abused as a privileged x86 processor interface to deterministically measure cache evictions in the unprotected code [29, 79].

Branch prediction. As a second important leakage source, several studies have exploited contention in the processor's Branch Target Buffer (BTB) [156] and directional branch predictors [59, 105] to spy on enclave-private control flow. These works essentially follow the general principle of the above Prime+Probe cache attacks by first forcing the internal branch predictor history buffer in a known state, before entering the enclave, and afterwards measuring a dedicated shadow branch in unprotected code so as to establish whether the secret-dependent victim branch in the enclave was executed or not. Importantly, unlike memory access side channels which are limited to a cache line granularity, branch shadowing attacks can leak control flow at the level of individual branch instructions, *i.e.*, basic blocks.

In the specific context of Intel SGX, several privileged adversary capabilities have been abused to improve the accuracy of branch prediction attacks. First, Lee et al. [156] mount a high-resolution BTB attack against SGX enclaves by similarly abusing page faults and APIC timer interrupts to interleave the unprotected branch shadowing code. Bluethunder [105] further uses SGX-Step to sample the directional branch predictor at a maximal temporal resolution, thereby improving the previously demonstrated BranchScope [59] attack. Second, all of these attacks abuse x86 performance-monitoring counters, including the last-branch record, to sample the unprotected shadow branches more precisely. As a third feature, Lee et al. [156] furthermore abuse the privileged `CR0` x86 control register to disable the CPU cache and slow down the victim enclave. Finally, it has been recognized that branch prediction attacks can also take place in an SMT setting when the indirect and directional predictor buffers are not partitioned across sibling cores [156, 59, 105].

CPU pipeline. Apart from amplifying conventional side channels, SGX attack research has also revealed new and unexpected sub-cache level leakage sources that arise from contention in the CPU pipeline itself. Chapter 5 presented our work on Nemesis which showed that, while single-stepping, the response time to service an interrupt may reveal which instruction is being executed in the enclave. Nemesis can be leveraged to derive several fine-grained microarchitectural properties, including instruction type, operand values, page-table walks, and cache misses. Concurrent to and independent from our work, He et al. [95] presented a coarse-grained covert channel based on interrupt latency correlations

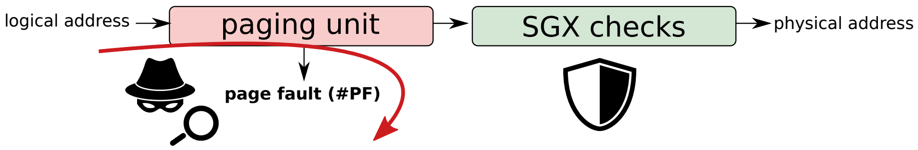


Figure 8.1: By provoking page faults in the untrusted address translation process, before SGX checks are applied, privileged adversaries can deterministically learn enclave memory accesses at a 4 KiB page-level granularity.

to the state of the store buffer, which has to be drained on enclave exit. More recently, Puddu et al. [208] leveraged the Nemesis framework to study subtle interrupt latency variations based on the alignment of enclaved store instructions. MemJam [180] furthermore exploits selective instruction timing penalties from false dependencies induced by an attacker-controlled SMT logical processor to reconstruct enclave-private memory access patterns at an improved, intra-cache line granularity. PortSmash [7] similarly exploits SMT-based contention to establish which microarchitectural execution ports are in use during execution of the victim enclave.

8.2.2 Controlled-channel attacks

A second attack category, referred to as *controlled channels* [277], observes that enclaved execution does not only occupy hidden microarchitectural resources, but also adheres to key architectural restrictions configured by the untrusted operating system. Particularly, as Intel SGX enclaves are confined to unprivileged ring-3 code [114], operating systems are free to restrict enclaved execution in space, through virtual memory, or in time, by means of interrupts. Microarchitectural contention attacks have abused these capabilities to amplify existing leakage sources, whereas controlled-channel attacks show that privileged processor interfaces can also in themselves be abused to construct new and dangerous types of deterministic side channels.

Page tables. While Intel SGX provides strong architectural protection against page remapping attacks [183, 97] by an untrusted operating system, several researchers have demonstrated new types of side-channel attacks that abuse SGX's untrusted address translation scheme. Even before the official launch of Intel SGX, Xu et al. [277] showed how privileged adversaries in control of the untrusted operating system can revoke access rights on a specific enclave code or data page and be deterministically notified by means of a page-fault signal when the enclave next accesses that page, as illustrated in Fig. 8.1. They overcome the relatively coarse-grained 4 KiB spatial granularity of the page-fault channel by observing that the *sequence* of page faults can uniquely identify specific points in the victim's execution. This attack technique has proven to be highly practical and effective, extracting full enclave secrets in a single run and without noise [277, 230, 270]. Notably, page-fault attacks have also been demonstrated for non-root adversaries, e.g., by registering a signal handler and abusing Linux's unprivileged `mprotect` system call [266, 249].

In Chapter 3, we contributed stealthy page-table attack techniques [258] that can reconstruct enclave page accesses without provoking page faults. These attacks proceed by issuing inter-processor interrupts to force Translation Lookaside Buffer (TLB) shootdowns and querying "accessed" and "dirty" attributes or monitoring the caching behavior of untrusted page-table memory. Wang et al. [263] furthermore showed that TLB entries can be evicted without interrupting the victim enclave by causing contention from a sibling SMT logical core. SGX-Step [257], presented in Chapter 4, leverages root capabilities to construct practical attack primitives for manipulating page-table entries and scheduling precise timer interrupts directly from the user-space enclave host application. Our work on CopyCat [182] finally proposes an innovative composite attack technique that combines coarse-grained 4 KiB page-table "accessed" bit patterns with deterministic interrupt counts harvested by SGX-Step to reconstruct control flow *within* a single enclave code page. Intuitively, where page-fault sequences only capture the relative order of consecutive page visits, CopyCat's notion of instruction-granular page access traces enriches the spatial resolution of the paging channel with an additional temporal dimension, revealing the exact number of enclave instructions between consecutive page visits. Figure 8.2 illustrates how CopyCat can deterministically reconstruct a page-aligned branch outcome by merely counting the number of instructions executed on the $P0$ code page containing the if branch, before control flow is eventually transferred to the $P1$ code page containing the `add` function. While traditional page-fault adversaries always observe the same fault sequence $(P0, P1)$, independent of the secret, CopyCat's interrupt counting technique results in distinguishable page access traces $(P0, P0, P1)$ vs. $(P0, P0, P0, P1)$.

Other vectors. Aside from the paging channel, we have demonstrated an alternative controlled-channel attack which abuses legacy IA32 segmentation faults [91]. Interestingly, before recent microcode updates, this attack could offer an improved, byte-level granularity in the first MiB of the enclave address space, but only for the unusual case of 32-bit SGX enclaves. Finally, Chapter 2 included a peculiar instance of a controlled-channel attack that abuses x86 `#AC` alignment-check exceptions to deterministically reveal unaligned data accesses in a victim enclave [254]. This attack can be mitigated by sanitizing processor

```
if (c == 0){ r = add(r, d); } else { r = add(r, s); }
```

```
 test %eax,%eax
 je 1f
 mov %edx,%esi
1:
 call add
 mov %eax,-0xc(%rbp)
```

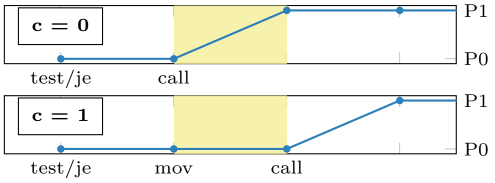


Figure 8.2: Balanced if/else statement (top), compiled to assembly (left). Precise page-aligned conditional control flow can be deterministically reconstructed with instruction-granular CopyCat page access traces (right).

flags on enclave entry and further relies on privileged control registers and single-stepping timer interrupts.

8.2.3 Transient-execution attacks

A third and more recent category of SGX attacks exploits side effects from speculative and out-of-order CPU pipeline optimizations. All of these attacks rely on the observation that processors may execute "transient" instructions out of the program's intended code or data paths, e.g., following a branch misprediction or exception event. While the processor's in-order commit scheme ensures that the illegitimate results of such unintended transient instructions are never persisted to the architectural state, attackers may inventively leverage side-channel analysis to reconstruct secret-dependent traces left by transient computations in the microarchitectural state. Crucially, where traditional side channels are restricted to leaking execution metadata, this new class of transient-execution attacks directly exposes raw enclave data and can hence not anymore be mitigated through constant-time code paradigms. Instead, over the last two years, transient-execution attack discoveries have necessitated a series of TCB recoveries [122] for the Intel SGX ecosystem, through a combination of processor microcode updates [107, 108], silicon fixes [149], and software patches [110, 117].

Leakage-oriented exploitation. Our work on Foreshadow [249], presented in Chapter 6, first explored the security implications of transient execution for Intel SGX enclaves. This research was conducted concurrently to Spectre [146] and Meltdown [162]. Where the latter is restricted to leaking privileged kernel data within the current address space and does not apply to SGX, Foreshadow contributes a highly practical attack primitive to leak arbitrary data from the L1D cache, regardless of access permissions or address space restrictions.

Foreshadow can extract enclave secrets on the current core, either preemptively, after exiting the victim enclave or swapping in an encrypted enclave page, or concurrently from a co-resident SMT logical processor. While this attack can even be mounted by unprivileged user-space adversaries through the `mprotect` system call,[1] underlying the root cause of Foreshadow is a privileged capability to manipulate page-table entries. We additionally developed several optimization techniques that may further improve the attack's overall effectiveness in a root adversary setting. These techniques include isolating the victim enclave on a dedicated core, flushing the cache hierarchy through the privileged `wbinvd` instruction, setting up aliased virtual memory mappings, and single-stepping enclaves with SGX-Step timer interrupts. Foreshadow furthermore first demonstrated an innovative "zero-stepping" technique which abuses page faults to force a victim enclave to repeatedly execute the same instruction and bring enclave register contents in the L1D cache without making forward progress. In response to Foreshadow, Intel issued microcode patches for existing processors and developed silicon mitigations that are now included in newer processors. The microcode mitigations flush the L1D cache on every enclave transition and furthermore extend attestation responses to enable remote parties to verify that SMT has been disabled [107].

The more recent class of Microarchitectural Data Sampling (MDS) attacks, including ZombieLoad [223] and RIDL [216], demonstrated that transient data leakage for faulting or assisted loads on Intel processors goes beyond the L1D cache and also affects internal line-fill buffers, load ports, and the store buffer. In the context of Intel SGX, both ZombieLoad and RIDL exploited line-fill buffer leakage to expose recently accessed enclave secrets. The more recent CrossTalk [211] attack furthermore showed that line-fill buffer leakage can also be leveraged to leak enclave requests from the on-chip hardware random number generator. The buffers exploited by MDS are an order of magnitude smaller than the L1D cache and offer restricted attacker control over addressing, necessitating data filtering and synchronization techniques. While MDS-style leakage is not exclusive to SMT scenarios, practical SGX attacks have focused on real-time data sampling and filtering from a concurrent logical processor. These attacks have even been demonstrated for unprivileged, user-space attackers [223, 216], but kernel-level adversaries typically have more leverage to manipulate page-table entries directly so as to cause faulting or assisted loads that trigger the transient data leakage. Furthermore, privileged adversaries can leverage aforementioned techniques using SGX-Step to precisely synchronize execution of the victim enclave through page faults, single-stepping, and zero-stepping [223]. In response to MDS attacks, Intel deployed microcode patches that flush affected

[1] Before PTE inversion [46] countermeasures were applied, the `mprotect` Linux system call could be used by the host application to clear the "present" bit in enclave page-table entries.

CPU buffers whenever entering or exiting an SGX enclave [108]. Furthermore, as with Foreshadow, remote attestation responses reflect whether or not SMT is enabled on the target platform.

Injection-oriented exploitation. In the aftermath of the original Spectre [146] attack, several researchers have demonstrated Spectre-type control-flow hijacking attacks against Intel SGX enclaves. While one initial proof-of-concept [197] validated the possibility of Spectre v1-style bounds check bypassing through mispredicted directional branches inside a victim enclave, practical SGX attack demonstrations have focused on poisoning more potent indirect branches which allow to arbitrarily hijack transient control flow through either the Branch Target Buffer (BTB) [39] or Return Stack Buffer (RSB) [148]. None of the ingredients for Spectre-type attacks strictly requires root privileges, but expectedly researchers have once more employed several of the aforementioned privileged attack techniques to construct more effective Spectre exploits in an SGX setting. Particularly, Chen et al. [39] propose to leverage enclave execution control primitives through page faults [277] and interrupts [257] to precisely advance the victim enclave to the desired attack point, before poisoning the BTB from the current core or a sibling SMT core. Spectre-style exploitation furthermore benefits from SGX's single-address-space design as transient code gadgets can directly access unprotected memory outside of the enclave, allowing to transmit secrets over highly effective FLUSH+RELOAD covert channels. In response to Spectre-type threats, Intel released software patches [117] that insert `lfence` speculation barriers after selected directional branches and furthermore issued microcode updates that flush indirect branch predictor state and disable speculative store bypass optimizations on enclave entry [128].

With Load Value Injection (LVI) [251], presented in Chapter 7, we showed that prior Meltdown-type microarchitectural data leakages, including Foreshadow and ZombieLoad, can also be reversely exploited as inventive injection primitives. By revoking access rights on enclave memory, LVI adversaries force a victim enclave to transiently compute on poisoned data forwarded from faulting or assisted load instructions. Similar to Spectre-type attacks, LVI relies on code gadgets in the victim enclave, which, when transiently executed with illegal data operands, expose secrets directly or allow to hijack transient control flow for second-stage gadget abuse. Crucially, in order to trigger page faults or microcode assists for trusted enclave load operations, LVI requires the privileged adversary's capability to manipulate untrusted page-table entries. Furthermore, analogous to Spectre-style exploitation, LVI attackers benefit from SGX's single-address-space design to setup convenient cache-based FLUSH+RELOAD covert channels in the untrusted memory region outside the enclave. SGX-Step single-stepping timer interrupts can further be leveraged to precisely advance the victim enclave to the desired code gadget location, before initiating the needed page-table manipulations. Importantly, LVI does not strictly rely on SMT, but gadget requirements may be somewhat relaxed when attacker-controlled data can be injected via a sibling logical processor. LVI essentially highlights the inadequacy of existing microcode and silicon mitigations that are based on zeroing illegal data flow or flushing leaky microarchitectural buffers after victim execution. In response, Intel developed extensive compiler mitigations that stall the processor pipeline by inserting explicit `lfence` speculation barriers after potentially every memory load operation. Additionally and even worse, due to implicit loads, certain instructions have to be blacklisted, including the ubiquitous x86 `ret` instruction.

Side-channel amplification. As a notable exemption to the rule that transient-execution attacks focus on data leakage, MicroScope [233] showed that transient instructions may also be abused to arbitrarily amplify traditional side-channel metadata leaks in only a single architectural run of a victim enclave. Similar to the aforementioned zero-stepping technique, first introduced by SGX-Step [257] and Foreshadow [249], their attack revokes access rights on a selected enclave trigger page and abuses that the processor's out-of-order pipeline may have transiently executed instructions following the faulting trigger instruction, before eventually exiting the enclave and delivering the exception to the untrusted OS. Crucially, this allows side-channel adversaries to gain insights into the microarchitectural resource utilization of a narrow amount of transient instructions following the trigger instruction, *without* actually advancing the architectural instruction pointer in the enclave. Hence, by repeatedly resuming the enclave, without re-instantiating access rights on the trigger page, privileged adversaries can sample arbitrarily many side-channel observations in only a single architectural invocation of the victim enclave. A similar effect of side-channel traces left by out-of-order transient instructions following a timer interrupt has also been observed in the CacheZoom [181] attack.

8.2.4 Attacks based on untrusted interfaces

A final category of attacks targets various kinds of interfaces between the enclaved execution environment and the untrusted world. This can be both explicit interfaces visible at the software level, as well as implicit interfaces used by the trusted processor to connect to untrusted hardware components like the voltage regulator or the DRAM controller.

Software interface. While TEEs, such as Intel SGX, provide strong architectural protection against malicious or buggy software running outside the enclave, these security guarantees are crucially dependent on the absence of exploitable bugs inside the enclave itself. Several authors have challenged this implicit assumption by either providing evidence [254] of real-world vulnerabilities in enclave software interfaces, or by developing improved attack techniques [154, 24, 218, 266] to practically exploit memory corruption vulnerabilities in an enclave setting.

AsyncShock [266] abuses enclave execution control through timers and page faults [277] to efficiently exploit synchronization bugs in multithreaded SGX enclaves. Alternatively, Schwarz et al. [218] abuse cache side channels to reliably expose double-fetch bugs for untrusted pointer dereferences in SGX enclaves. DarkROP [154] augments established return-oriented programming [228] attack techniques with information leakage from page fault side-channel oracles [277] to practically discover gadgets in unknown, encrypted enclave application binaries. This attack requires kernel privileges and a static enclave memory layout. More recently, Biondo et al. [24] relaxed these requirements by demonstrating an improved ROP attack which allows even non-privileged adversaries to hijack vulnerable enclave applications by leveraging gadgets in the enclave's trusted runtime, which is not considered by state-of-the-art enclave address-space randomization solutions [227]. Both of these attacks further benefit from SGX's single-address-space design, where code gadgets can directly read or write to attacker-controlled memory locations outside the enclave, and where the CPU register file is shared across enclave transitions. In a perpendicular line of research, Schwarz et al. [219] criticized SGX's design choice of providing enclaves with unlimited access to untrusted memory outside the enclave. They considered an inverse attacker model, where not the code inside the enclave, but the host application is under attack. Their attack demonstrates that malware code executing inside an SGX enclave can mount stealthy code-reuse attacks to arbitrarily hijack control flow in the host application.

In Chapter 2, we considered the question as to how prevalent these vulnerabilities are in real-world enclave software, uncovering a wide and reoccurring vulnerability landscape across 8 major open-source TEE runtime libraries. This work demonstrated several confused-deputy attacks that abuse pointer passing in the shared address space, and furthermore employed diverse exploitation techniques for privileged adversaries. One particular case-study attack, for instance, leverages SGX-Step interrupts and page-table "accessed" bits to deterministically exploit a subtle timing side-channel oversight in the pointer validation logic of Intel's SGX SDK.

Power management. To address ever-growing heat and power consumption concerns, modern processors commonly expose privileged software interfaces for dynamically adjusting the processor's operating frequency or voltage levels. These interfaces have traditionally received little security scrutiny as they are made exclusively available to the privileged operating system kernel, out of reach for traditional user-space adversaries. This changes, however, in a TEE setting, where kernel-level adversaries can now abuse power management features to forcibly operate the trusted processor at an increased frequency or a lowered voltage during enclaved execution [241]. Crucially, when this affects the processor's critical path circuitry, faulty computation results may arise in enclave mode. Our research on Plundervolt [188] for the first time broke Intel SGX's integrity guarantees by abusing an undocumented MSR privileged software interface for voltage scaling on recent Intel processors. Particularly, we showed that, by fixing the processor's frequency and slightly lowering its operating voltage before entering a victim enclave, predictable bit flips can be triggered in certain high-latency x86 operations, including ubiquitous multiplication and AES-NI instructions. Apart from breaking several cryptographic implementations, this research also demonstrated an inventive new kind of memory-corruption attacks that induces bit flips in pointer arithmetics so as to redirect trusted in-enclave pointers to attacker-controlled memory locations in the untrusted address space outside the enclave. Similar to many of the aforementioned SGX attacks, kernel-level adversaries may furthermore benefit from control over page tables and interrupts to precisely advance the victim enclave's execution, before eventually triggering the undervolting by writing to the privileged MSR. Concurrent to our work, Qiu et al. [210] demonstrated a similar fault attack against a software-based AES implementation, and Kenjar et al. [137] explored the effects of undervolting x86 vector operations in SGX enclaves while additionally exercising stress from a sibling SMT logical processor. In response to these attacks, Intel updated SGX remote attestation to reflect whether the undocumented voltage scaling MSR has been disabled [129].

Memory bus. Intel SGX was purposely designed to protect even against advanced adversaries who have unrestricted physical access to the enclave host machine, e.g., untrusted cloud providers under the jurisdiction of foreign nation states. More precisely, the Intel SGX architecture solely trusts the processor package, while leaving any external hardware components explicitly untrusted. SGX therefore includes a dedicated Memory Encryption Engine (MEE) [89] that protects the confidentiality, integrity, and freshness of all enclave memory while it resides in untrusted off-chip DRAM. However, the MEE does not protect address metadata and can hence not defend against physical side-channel adversaries who tap the unencrypted memory address bus to deterministically learn enclave last-level cache misses at a 64 B cache line granularity [89, 47]. The recent

Membuster [152] attack indeed practically demonstrated such an attack by physically placing a custom DRAM interposer to record all off-chip memory traffic. This attack further relies on the privileged adversary's control over untrusted page tables for strategically interrupting the enclave, back-translating the observed physical addresses to the corresponding enclave virtual addresses, and artificially increasing pressure on the last-level cache. Depending on the required level of stealthiness, attackers may furthermore consider disabling the CPU cache altogether through the privileged `CR0` x86 control register.

Finally, as another attack vector potentially related to the untrusted memory bus, Intel recently disclosed a processor vulnerability [123] which may allow privileged system adversaries to extract the first two words of every enclave cache line through an obscure interaction with the integrated graphics card. In the absence of more details or independent reproduction, we can only hypothesize on the root cause and impact of this vulnerability. The fact that the processor's integrated graphics card is involved hints at a possible interaction with the Direct Memory Access (DMA) subsystem. In response to this vulnerability, Intel extended SGX's remote attestation responses to reflect whether integrated processor graphics technology has been disabled [129]. Alarmingly, for SGX applications who are unable to disable integrated graphics, Intel recommends extensive software mitigations that should change all enclave-internal memory allocations to avoid the affected regions of a cache line [123].

8.2.5 Tendencies and lessons learned

This section aims to pave the way for reasoning about effective defense strategies by investigating trends and commonalities across the attack spectrum. We further identify challenges and future directions, and distill lessons for future TEE designs.

Privileged attack primitives. Looking back at Table 8.1, several clear tendencies can be observed. Perhaps the most important lesson is that potentially *every* privileged processor interface can be abused as an unconventional attack vector. Some of these privileged hardware-software interfaces have remained restricted to niche attack scenarios, e.g., segmentation faults for 32-bit enclaves [91] or control registers to spy on unaligned data accesses [254], whereas others have proven to be extremely versatile and have been re-purposed across many different attack scenarios. The question hence arises which privileged interfaces are considered most dangerous and should receive increased attention in next-generation fortified TEE designs (cf. Section 8.3).

In this respect, the systematization in Table 8.1 clearly reveals that untrusted page tables and interrupts have been most widely leveraged across different attack categories. Better understanding the security implications of these privileged interfaces was indeed the primary motivation behind the SGX-Step framework, presented in Chapter 4, which exposes both virtual memory page-table mappings and APIC timer interrupts via a practical user-space attack library. As further evident from Table 4.1 on page 101, these primitives have enabled a long line of high-resolution attacks. Untrusted page tables in particular have been taken advantage of for a wide variety of purposes: from constructing deterministic side-channel oracles [277, 258, 263], over coarse-grained enclave execution control [257, 266, 156], to ultimately the driving force behind many transient-execution pipeline vulnerabilities [249, 223, 216, 251]. Likewise, precise timer interrupts have been leveraged both to drastically improve the temporal resolution of existing attacks [257, 181, 92, 156, 249, 105, 251], as well as to mount new and unexpected types of side channels [256, 254, 182]. The tandem of interrupts and page tables became even more explicit in more recent versions of SGX-Step which leverage page-table "accessed" bits to deterministically filter out superfluous zero-step observations [256]. CopyCat [182] furthermore showed that accessed bit patterns can be leveraged in combination with SGX-Step interrupt counts to precisely reconstruct enclave control flow *within* a single 4 KiB code page.

As to the remaining processor interfaces, especially the model-specific registers related to privileged power management have proven to be particularly advantageous. Almost all microarchitectural timing attacks unanimously abuse MSRs for noise reduction by disabling dynamic frequency scaling, fixing the processor's clock, and optionally disabling the hardware prefetcher. Likewise, on-core performance-monitoring counters, which are disabled for production enclaves in the SGX design [114], have been widely abused as a noise-reduction technique for probing shared microarchitectural resources in the attacker's own, unprotected code execution [79, 29, 156, 59, 105]. More recently, Plundervolt [188] for the first time demonstrated the potentially devastating security impact of the model-specific register interface, beyond merely noise reduction, by abusing an undocumented MSR for voltage scaling to reliably induce bit flips in enclave computations. Given the vast amount of MSR and PMC registers in modern x86 processors, further analysis should tell whether and how they could be leveraged to mount new varieties of attacks against SGX enclaves.

SMT-based processor scheduling interfaces have been uniformly abused, across all attack categories, to induce microarchitectural contention from a sibling logical processor. SMT has proven to be particularly dangerous in the context of transient-execution attacks, which may directly leak values being processed on a co-located enclave thread [249, 223, 216], prompting Intel to include SMT status in remote attestation responses. Importantly, however, only very few attacks [7, 180] critically rely on SMT and almost all leakages can also be induced in a non-SMT setting, *i.e.*, by frequently interrupting the victim enclave and probing the microarchitectural state on the current physical processor [257, 181].

A last important tendency in Table 8.1 is that almost none of the microarchitectural-contention attacks strictly requires root capabilities, yet all of them repeatedly execute the victim enclave so as to compensate for measurement noise. The controlled-channel family of attacks, on the other hand, behaves fully deterministically in a single run of the victim enclave, yet all of these attacks require some form of control over page tables or interrupts.[2] This observation stems from the fundamentally different nature of these attack categories: controlled channels derive execution metadata via adversarial manipulations of traditionally privileged processor interfaces, whereas traditional side channels derive similar, potentially even finer-grained metadata from subtle timing differences caused by undocumented contention sources in the underlying microarchitecture. This separation essentially implies that it might be feasible to eradicate certain types of controlled channels by carefully re-designing the privileged hardware-software contract, for instance by removing attacker control over page tables [48, 58] or avoiding virtual memory altogether [189], while restricting the adversary's control over privileged processor interfaces would at best only somewhat raise the bar for microarchitectural-contention attacks. Indeed, effective microarchitectural timing attacks have even been demonstrated for unprivileged enclave adversaries [225].

Side-channel leakage. In summary, the above research results show that enclave code and data accesses on SGX platforms can be accurately reconstructed, both in space, at a 4 KiB page, 64-byte cache line, or even sub-cache line granularity, as well as in time, after every single instruction. Given that attacks only improve over time, we should expect this tendency to continue and lead to even more advanced side-channel exploits that leak execution metadata at ever-growing granularity and accuracy.

According to Intel SGX's official threat model [119], execution metadata leaks through side channels are explicitly considered out-of-scope, and it is recommended to employ constant-time cryptographic algorithms in enclave applications. Unfortunately, however, the past years of SGX attack experience have shown that adequately preventing side-channel leakage is particularly difficult—to the extent where even Intel's own vetted enclave entry code [254],

[2]Note that such control could potentially also be indirectly provided through the unprivileged system call interface, e.g., `mprotect` [249].

architectural quoting enclave [50], and cryptographic IPP library [180] all suffered from subtle yet dangerously exploitable side-channel vulnerabilities. Moreover, several attack studies [277, 29, 92, 256] have considered non-cryptographic SGX applications, showing that sensitive enclave data may be more ill-defined, and hence harder to keep track of in constant-time code paradigms, compared to the typical cryptographic keys of side-channel analysis.

Overall, while some side-channel attack vectors, like page faults, cache timing attacks, or port contention have been recognized early-on [47], the SGX attack scene has also revealed several new and unexpected leakage sources [256, 180, 91, 182]. Furthermore, by implementing practical attacks and exploring the offensive potential of privileged processor interfaces, researchers have shown that traditionally noisy side channels can be exploited at a considerably higher resolution and increased success rate in an enclave setting. This research on the interplay between the processor's architectural specification and the underlying microarchitecture ultimately paved the way for transient-execution attacks, which could not have been anticipated at SGX design time.

Transient execution. In the transient-execution attack landscape, that unfolded over the last two years, we have seen similar tendencies. Researchers first focused on relatively well-understood microarchitectural structures like the L1D cache [249, 162] and branch target buffer [146, 39], before moving on to smaller and more ill-defined structures, like the return address predictor [148], line-fill buffers [216, 223], store buffer [35], and load ports [61]. Interestingly, we notice that, as these attacks become more involved and target deeper realms of the processor pipeline, mitigations similarly become less clear-cut. Following the classic cat-and-mouse game, several iterations of processor microcode mitigations [109, 127] have been refined to protect against ever-changing attack variants. This tendency is likely to continue until more principled silicon mitigations are widely deployed. Finally, as a turning point, our research on LVI [251] showed that even microcode mitigations are fundamentally insufficient to protect against privileged SGX adversaries that can resort to an injection-oriented exploitation methodology for incorrect transient forwarding, even if all leaky microarchitectural buffers are cleared at the enclave boundary.

Fault attacks. Transient injection attacks, like LVI and Spectre, manifest entirely at the microarchitectural level: the faulty computations out of the intended path are only "transiently" executed and are never persisted to the architectural state. Our work on Plundervolt [188], however, for the first time induced *persistent* architectural faults by exploiting more fundamental properties of the logic fabric of the physical CPU itself—namely the need for a stable supply voltage. This research for the first time extended the software-exposed attack surface of SGX from the "high-level" microarchitectural design to the underlying physical properties of the electronic circuitry itself.

We can only expect more, yet-undiscovered physical effects to be exploited in the future. The smartcard industry has spent decades on defending much less complex chips against physical side-channel and fault attacks. It remains to be seen how many of those attacks apply to high-end processors, be that for physical or software-level adversaries, and whether Intel and others can learn from the smartcard experience to strike a cost-to-benefit balance between performance, functionality, and security.

TEE design. An important conclusion, overarching five years of continued SGX attack research, is that Intel SGX's *architectural* design [14, 176, 89] proved to be very stable and has so far never been fundamentally broken. This is in notable contrast with, for instance, AMD's Secure Encrypted Virtualization (SEV) technology [136] which similarly aims to protect against malicious hypervisors by means of hardware-level memory encryption. Yet, SEV repeatedly fell victim to a concurrent line of attacks. Some of the architectural limitations demonstrated in current and previous SEV versions include the lack of protecting register file contents across interrupts [97, 272], the lack of verifying untrusted extended page-table translations [183, 97], and the absence of freshness and integrity checks for encrypted memory [274]. Importantly, all of these limitations are properly prevented in the Intel SGX design, ruling out this considerable attack surface at the architectural level. Note that some of these attacks are also addressed in revised versions of AMD's SEV architecture: in the form of SEV-ES [135] which protects register state, similar to SGX's AEX scheme, and the recently announced SEV-SNP [11] extension which will safeguard untrusted address translations, similar to SGX's EPCM protections.

In fact, many of the issues we have witnessed stem from SGX's decision to implement high-assurance enclave "fortresses" on top of arguably weak foundations, *i.e.*, a complex out-of-order x86 processor. This has not only led to a proliferation of microarchitectural side-channel attacks, which were anticipated and considered out-of-scope at design time [133], but ultimately also paved the way for a dangerous new type of transient-execution processor vulnerabilities. Such transient leakages in the underlying microarchitecture were not foreseen at design time and led to dramatic consequences, subverting all of SGX's security guarantees on unpatched systems [249]. This attack surface will likely continue to be addressed by the development of several more iterations of processor microcode updates, silicon-level changes, and software patches. Fortunately, yet once more in contrast to alternative TEE designs such as AMD

SEV [32], SGX's remote attestation [14] and TCB recovery [122] schemes have proven to be sufficiently versatile and resilient to allow remote stakeholders to verify that all of these mitigations have indeed been properly applied.

A final TEE design consideration relates to the decision to embed SGX enclaves in the virtual address space of an untrusted containing host application, resembling alternative single-address-space TEEs such as Sancus [189] or Sanctum [48]. This design closely mimics existing user-to-kernel boundaries and allows for flexible enclave transitions by passing input and output buffers as pointers in the shared address space. However, several researchers have abused the enclave's unrestricted access to unprotected memory to facilitate confused-deputy attacks after hijacking the enclave through either transient execution [39, 251], memory-safety misbehavior [254, 154], or undervolting [188]. While none of these attacks critically relies on a shared address space, we expect that they would be considerably harder to exploit in alternative dual-world TEE designs, like ARM TrustZone [10], AMD SEV [136], or RISC-V Keystone [153], where enclaves live in their own designated virtual address space and all external communication is restricted to explicit shared memory regions.

Generalization to other TEEs. The past years have seen a remarkable synergy, where some of the insights gained from attack research on SGX platforms have been transposed to, or inspired by, other TEEs and ultimately even non-TEE settings. As a telling example, Ryan [214] recently adapted techniques from SGX-Step [257] and CacheZoom [181] to develop the Cachegrab framework for precisely measuring cache evictions and branch prediction leakage on ARM TrustZone TEEs. Mimicking privileged adversary advantages in the SGX world, Cachegrab reports significantly less noise and an improved temporal resolution compared to prior cache attacks on TrustZone [161, 37]. Likewise, several researchers [97, 183, 272, 274] have exploited page-fault side channels, first introduced in the context of Intel SGX [277], as a building block for various attacks on the AMD SEV architecture. In this respect, Werner et al. [272] further propose to abuse the debug trap flag as a "hyper-stepping" primitive for SEV enclaves, closely resembling the effect SGX-Step achieves with timer interrupts. The design of SGX-Step has similarly inspired the Sancus-Step [49] framework for single-stepping interruptible Sancus enclaves.

There have also been notable influences in the inverse direction. For instance, the CLKscrew [241] attack, first demonstrated on ARM TrustZone, paved the way for subsequent software-based fault attacks [188, 210, 137] on Intel SGX platforms. Likewise, our own work on Nemesis interrupt latency side-channel attacks, presented in Chapter 5, was first prototyped on the embedded Sancus TEE and only later generalized to Intel SGX enclaves. This once again illustrates that open-source academic architectures, such as Sancus, may serve as a research vehicle to explore limitations of and possible improvements for real-world TEEs like Intel SGX [34, 185].

A final tendency worth noting is that SGX attack research has often sparked generalizations to traditional unprivileged process isolation, as also observed by Schwarz et al. [221]. A telling example of this is our own work on Foreshadow-SGX, presented in Chapter 6, which directly led to the discovery of the even more impactful Foreshadow-NG [271] attack variant affecting widespread virtual machine isolation. A further example is provided by address translation side channels, which have been a historic area of interest for breaking kernel ASLR [104, 86, 132]. Informed by recent research results [277, 258] for SGX root adversaries, such address translation side channels have since received renewed attention in restricted environments as well [81, 259, 83]. Likewise, older research on branch prediction side channels [3, 2] was revived and significantly strengthened in the context of SGX enclaves [156, 59]. This renewed interest in branch prediction side channels led to an improved understanding of the underlying microarchitectural structures, ultimately contributing to the discovery of Spectre. We can assume that this fruitful interchange of ideas will only continue and lead to new and improved research results beyond the Intel SGX architecture.

8.3 Moving forward: Avenues for TEE hardening

Following a systematic understanding of the attack scene, this section formulates concrete recommendations for hardening current and next-generation TEE designs. Table 8.2 lists selected countermeasures from both industry and academia and summarizes their effectiveness across the different attack categories introduced in the previous section and Table 8.1. An important first observation is that no known silver-bullet solution exists to single-handedly address leakage from side channels, controlled channels, transient execution, and interface-based attacks. Instead, in practice, the fundamentally different nature of these threats requires an intricate composition of perpendicular protective measures. For instance, an exemplary "ideal" enclave would presently be developed using *(i)* a memory-safe language or formal verification tool [131] to rule out traditional software vulnerabilities, like buffer overflows; *(ii)* an expert developer or compiler-assisted solution [43] to safeguard constant-time behavior for secret-dependent code and data accesses; and, finally, *(iii)* a series of thoughtful `lfence` instructions [117, 251] to avert leakage from inadvertent transient-execution paths. Table 8.2 furthermore shows that present, rather ad-hoc solutions are characterized by a somewhat rigid separation between the

Table 8.2: Overview of selected countermeasures. Round symbols indicate defenses that can fully (●) or partially (◐) mitigate an attack category, whereas diamonds represent hardening techniques that can impede (⬖) or detect (◆) exploitation attempts. The colored plane demarcates academic proposals, which are not currently available. Defense techniques first proposed in this thesis are highlighted.

Attack \ Defense	Flush/ partition	**eresume padding**	eresume hook	**A/D mask**	Self paging	**Access prevention**	Uncacheable	Capability	lfence insertion	Cst time	Safe lang
μ-arch contention	◐	⬖	◆	○	○	○	◐	○	○	●	○
Page access pattern	○	○	◆	⬖	●	○	○	○	○	●	○
Single-stepping	○	⬖	◆	⬖	⬖	○	○	○	○	○	○
Transient domain-bypass	●	○	○	○	○	○	○	○	○	○	○
Transient cross-domain	◐	○	○	○	◐	⬖	◐	◐	●	○	○
Memory safety	○	○	○	○	○	⬖	○	●	○	○	●
Off-chip memory bus	○	○	○	○	○	○	○	○	○	●	○
	HW		**HW/SW contract**						**SW**		

hardware and software levels. In between lies an unfolding research space, where well-considered changes to the traditional hardware-software contract may lead to more principled and inclusive mitigation strategies.

In the following, we first overview mitigation strategies that can be transparently applied at the microarchitectural level. Next, we explore the more integrated spectrum of hardening techniques that aim to practically improve side-channel resistance via balanced revisions to the hardware-software contract. The final section is devoted to the question of efficiently restraining transient execution, rightly identified as one of the major challenges in the coming decade's "new golden age for computer architecture" [96].

8.3.1 Microarchitectural hardening

Side-channel leakage based on contention to a shared microarchitectural resource can in principle be mitigated by providing the enclave with exclusive access to said resource. This slicing can either proceed in time, by flushing core-private resources on enclave entry and exit, or in space, by statically partitioning larger structure that are shared across cores, e.g., the processor's last-level cache. Given the required hardware primitives, such approaches may ultimately shield application software in a transparent manner, as for instance exemplified by the ongoing "time protection" [70] effort for the seL4 microkernel.

Enclaves furthermore have the additional advantage that protection boundary transitions are well-marked through dedicated entry and exit instructions, which recently indeed have been extended in the light of transient-execution threats to progressively flush ever more microarchitectural buffers at the enclave boundary [128, 107, 108].

There is obviously an associated cost to be payed, however, by constraining some of the very optimizations that have traditionally driven microarchitectural performance advances. Furthermore, perhaps the key weakness of this approach lies in its singularity. That is, identifying and partitioning *every* individual microarchitectural contention source may well turn out to be an endless endeavor for modern x86 processors. Effective partitioning would hence be more likely to succeed for much simpler RISC-based processors [34, 63] where an open-source understanding of the underlying microarchitecture could further strengthen confidence in the resulting security guarantees [185, 19]. Ultimately, however, even in a fortified and fully partitioned microarchitecture, enclaves would be akin to isolated networked machines, where it is well-known that response time for remote API calls can still be abused as a capable side channel [31]. Researchers have, for instance, constructed fully remote, time-driven cache attacks [4], which rely on statistical inference to derive secrets from overall execution time correlations to the aggregate number of cache hits and misses. Likewise, NetSpectre [224] demonstrates that, provided with sufficient gadgets in the victim code, even transient-execution attacks can be mounted against a fully remote, isolated microarchitecture. As a final important limitation, microarchitectural partitioning cannot protect against side-channel leakage that stems from higher or lower levels in the system stack. As an example of the former, consider controlled-channel attacks [277, 258] that abuse ISA-level abstractions, and as an instance of the latter, consider off-chip memory address bus tapping [152] attacks that fully manifest at the physical level.

Flushing core-private resources. The Intel SGX architecture originally only flushed the TLB on enclave transitions to safeguard architectural consistency and protect against page remapping attacks, while leaving other core-private resources untouched [47]. This decision was likely made with performance of enclave transitions in mind. Interestingly, however, in response to the recent wave of transient-execution findings, a series of microcode updates now clears increasingly many core-private microarchitectural buffers at the enclave boundary, retarding context switch times with at least a factor $> 2\times$ [265]. At the time of this writing, these buffers include indirect branch prediction history via the BTB and RSB to mitigate a subset of Spectre [39, 148] variants; the L1D data cache to mitigate Foreshadow [249]; and the line-fill buffers, store buffer, and load ports to mitigate data sampling attacks [223, 216, 35]. Additionally, to prevent concurrent exploitation from a sibling SMT logical processor, SGX's remote attestation scheme has been extended to reflect whether SMT is disabled.

Importantly, the above measures were designed to counter a subset of transient data leakage attacks, but do currently *not* necessarily prevent metadata leakage through side-channel analysis. For instance, even when the L1D cache is flushed on enclave exit, adversaries can straightforwardly adjust to infer access patterns from the L2 and last-level caches [225]. Furthermore, Hosseinzadeh et al. [103] observe that up-to-date Spectre microcode patches, which properly flush or tag indirect branch predictor buffers on enclave entry, appear not to flush said buffers on enclave exit, allowing side-channel leakage through branch shadowing attacks [156]. Huo et al. [105] report a similar observation for the PHT directional branch predictor. As performance for enclave transitions is already impacted by the current measures anyway, it seems appropriate to extend the microcode to at least flush branch prediction history buffers on enclave exit as well.

Partitioning shared resources. In contrast to core-private buffers, some structures like the last-level cache are competitively shared across multiple, mutually distrusting CPU cores and hence present a much more conceptual challenge to partition securely. One possible solution could build on Intel's recent Cache Allocation Technology (CAT) [112], which allows to reserve portions of the last-level cache for exclusive use by a particular core. However, CAT was originally developed as a quality-of-service solution and its applicability to secure enclaves seems far from straightforward. Furthermore, adding to the tension with the operating system's traditional role as a resource manager, enclave memory in the last-level cache should either remain locked or be flushed on interrupts. Alternatively, solutions based on cache coloring schemes [48] relieve contention in the last-level cache by making sure to never assign physical addresses with identical set indices to different security domains. Unfortunately, however, this approach does not apply to Intel SGX as it would require somehow restricting the privileged adversary's control over enclave physical address allocations.

Alternatively, when rigid partitioning is considered infeasible, adequate randomization could be applied in the shared microarchitectural resource, such that exploiting the contention becomes heuristically infeasible. This might be especially opportune to preserve the performance benefits of competitively sharing a sizable last-level cache without the associated attack surface in terms of side-channel leakage. ScatterCache [273] recently proposed to replace the fixed cache-set address mappings of present processors with a keyed pseudo-random function so as to thwart the construction of targeted PRIME+PROBE eviction sets. ScatterCache's hardware design appears to lend it self especially well to an enclave context, as it permits to mutually isolate different security domains by including a configurable domain identifier in each key derivation. ScatterCache suggests to specify the domain identifier via page-table attributes configured by the trusted operating system. In the context of SGX, however, domain identifiers could be trivially set and reset by the processor on enclave transitions.

Partitioning the pipeline. A final microarchitectural hardening consideration relates to contention in the CPU pipeline itself. Our work on Nemesis, presented in Chapter 5, showed that interrupt latency may reveal instruction-granular microarchitectural state inside the enclave. From this research we can conclude that, when time slicing the CPU between code of different privilege levels, transitions should ideally be constant time. In this respect, a closely related requirement is that the microarchitectural flushing primitives themselves should also behave deterministically, which is not the case on current processors where, for instance, the time to flush L1D correlates with the number of dirty lines [70].

We have recently devised and verified a provably constant-time enclave interrupt mechanism for fully deterministic 16-bit embedded Sancus processors, relying on an intricate two-level execution time padding scheme [34]. This static padding solution relies on predictable instruction execution time and does not straightforwardly generalize to a complex x86 processor, however, where execution time is generally non-predictable and instructions may have many more observable side effects. In this setting, randomization might therefore pose a better alternative. Particularly, timing differences encoded via interrupt latency could be partially masked by including wide enough random timing delays for Asynchronous Enclave Exit (AEX) events, or at the end of the `eresume` instruction. Provided that these operations are implemented in microcode [47], such masking could likely even be retroactively applied to existing SGX processors. An important consideration, however, is that uncorrelated and uniformly distributed random noise can always be compensated for with repeated measurements over multiple runs [71]. This countermeasure would hence additionally require enclaves to refuse to repeatedly compute over the same data. From the Nemesis analysis in Chapter 5, we can furthermore conclude that random timing delays should in practice measure several hundreds of cycles and that the CPU's random number generator would require several hundreds more cycles on top, making this a potentially costly approach that might significantly delay high-priority interrupts. Depending on the real-time requirements of the underlying host operating system, such delays may or may not be acceptable.

Apart from masking interrupt latency, randomization seems tempting as well to thwart single-stepping frameworks like SGX-Step [257], which have proven

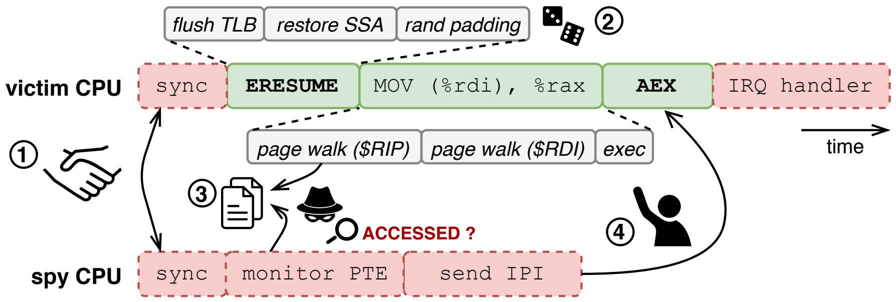


Figure 8.3: Adversaries can precisely single-step a victim enclaved execution (green, solid) without resorting to timers by sending inter-processor interrupts from a concurrent spy thread (red, dashed) that monitors unprotected page-table accesses.

to be particularly advantageous across many different attack categories (cf. Tables 4.1 and 8.1 on pages 101 and 230). Crucially, however, we do *not* consider mere randomization in itself to considerably raise the bar for SGX-Step adversaries. While ample random padding in `eresume` would indeed make it harder to reliably configure the APIC interrupt timer delay before entering the enclave, adversaries may adapt in at least three distinct ways. First, for many (but not all) scenarios true single-stepping is not a requirement and simply overestimating the delay would still result in a practical, yet coarser-grained multi-step primitive. Prior work on branch prediction side channels has, for instance, demonstrated practical attacks even with a noisy preemption primitive of up to 50 instructions [156]. Second, adversaries may conservatively underestimate the random padding delay and rely on page-table "accessed" bits to filter out the abounding zero-step observations [256], resulting in a substantially slower, yet accurate single-stepping primitive. Last and most crucial, we observe that any single-stepping defense based on restricting the adversary's control over timer interrupts on the current core is inherently insufficient in a multi-core setting. That is, attackers can always resort to a dedicated spy thread which monitors side effects of the enclaved execution and shoots down the victim CPU via inter-processor interrupts. Precisely this technique has indeed been demonstrated in Chapter 3 to reliably interrupt a victim enclave at instruction-level granularity, even in a non-SMT setting, by monitoring enclave page-table activity in the shared last-level cache [258].

Figure 8.3 summarizes how inter-processor interrupts can be leveraged to single-step a victim enclave on a hypothetical CPU with randomized `eresume` padding. The adversary first synchronizes ① with a dedicated spy thread on a separate physical CPU core, before entering the victim enclave with `eresume`. After restoring private register contents and stalling the processor with a random timing delay ② the victim eventually starts executing the first instruction (`mov` in this example). The execution fetch stage requires at least one page-table walk ③ to retrieve the physical address of the enclave code page, and optionally another page-table walk to resolve any data operand addresses. At this point, *i.e.*, *after* the random timing delay and *before* completion of the enclave instruction, the attacker-controlled spy thread observes an update of the "accessed" bit or caching behavior of the corresponding untrusted page-table entry and finally ④ sends an inter-processor interrupt, which instructs the victim CPU to halt enclaved execution and initiate AEX after completion of the current, single instruction. Notably the above example focuses on monitoring page-table entries, but in principle any observable side effect from the enclaved execution could be used as a side-channel oracle to trigger the inter-processor interrupt at the desired time. At an abstract level, the adversary only needs to distinguish the added dummy padding from the actual execution. While some additional protection against this type of inter-processor attacks could be achieved by shifting some of the instruction fetch and decode logic within `eresume`, before applying the random padding delay, we expect that it will generally be infeasible to masquerade all observable side effects for the complex x86 instruction set, where data access micro-ops, for instance, only occur later during the actual instruction execution phase.

8.3.2 Revising the hardware-software contract

As evident from Table 8.2, current solutions in the defensive landscape lean strongly toward either the hardware or software sides. However, such rigid abstraction levels are relative in the eyes of attackers, as side channels commonly allow to bypass high-level security restrictions by exploiting unconstrained optimizations at lower levels. In light of these findings, one of the major challenges in the field of computer architecture is how to rethink the traditional hardware-software interface so as to allow security restrictions to be more explicitly expressed and maintained across the system stack. The current and the next section overview several concrete proposals in this respect.

Interrupt-awareness for enclaves via `eresume` interception

Resembling Intel's virtual machine extensions, the current SGX architecture leaves enclave applications explicitly *interrupt-unaware* by design. That is, SGX enclaves can be transparently continued through the dedicated `eresume` instruction following a page fault or external interrupt event. Software handlers can optionally be registered by the enclave, but this requires the explicit cooperation of the untrusted application by first calling the exception handler through `eenter` before invoking `eresume`. This is in sharp contrast to alternative, interrupt-aware TEE designs such as Sancus [189] or Sanctum [48], which always require re-entering a previously interrupted enclave via its normal software entry point so as to inspect and resume the paused execution state from the trusted runtime inside the enclave itself. These alternative architectures strike an arguably cleaner balance between hardware and software concerns by allowing the enclave entry point to enforce flexible service contracts with the untrusted operating system, e.g., in terms of interrupt rates.

SGX's obliviousness to interrupts and page faults has been a key driving factor for many of the attacks summarized in Table 8.1. Making enclaves aware of asynchronous exits would allow to implement certain software-level security policies, e.g., heuristically detecting suspicious interrupt rates [229, 42] or prefetching [237, 175] secret-dependent memory locations into a trusted cache. Over the past years, several researchers have proposed such side-channel hardening techniques, summarized below, which all commonly aim to detect asynchronous attacker interferences through preemptions or evictions during the execution of a security-critical routine inside an enclave. Upon such detection, execution is redirected to a trusted software handler routine which can subsequently decide to terminate the enclave or to re-establish a security invariant by repopulating microarchitectural caches with evicted entries.

Hardware transactional memory. Recent Intel x86 processors ship with Transactional Synchronization Extensions (TSX) which simplify concurrent programming by synchronizing the critical sections of multiple threads without the overhead of software-based locks. Code executing in a TSX transaction is aborted and automatically rolled back whenever encountering a cache conflict or exception. Various authors [229, 42, 84, 237, 40, 9] have proposed to leverage TSX transactional aborts in SGX enclaves as an indicator for an ongoing, possibly interrupt-driven, attack. However, this approach suffers from several limitations. First, frequent transaction aborts may also occur in benign scenarios, e.g., under heavy system load with many device interrupts or concurrent activity in the last-level cache. Second, heuristic detection policies inevitably suffer from false positive or negative rates, as further discussed below. Lastly, wrapping code in TSX transactions may induce substantial performance overheads [229, 237] and TSX is furthermore not available on all SGX-enabled processors. We hence argue that the TSX-based defenses in this category should rather be regarded as academic exercises to explore the possibilities of the proposed native instruction set extensions discussed below.

T-SGX [229] aims to protect against page-fault-driven attacks via an LLVM-based compiler transformation that wraps each basic block in a TSX transaction.

Any page fault while in TSX mode will not be reported to the operating system, but instead causes execution to be redirected to the in-enclave abort handler. T-SGX proposes to terminate the enclave after counting too many consecutive transaction aborts. Instead of wrapping the entire enclave program in costly transactions, Déjà Vu [42] proposes to instead use TSX to construct a dedicated in-enclave reference counter thread that cannot be silently stopped by the operating system. The enclave program is further instrumented to time its own activity, relative to the counter thread, so as to detect the execution slowdown associated with an unusually high number of AEXs. S-FaaS [9] uses a similar mechanism for trusted, in-enclave resource accounting. Apart from recognizing AEXs, Cloak [84] leveraged TSX aborts to detect adversarial evictions in the processor's last-level cache. This work furthermore includes a clever mechanism to rule out SMT-based side channels by ensuring that two sibling logical CPU cores enter and exit the enclave in lock step. Cloak achieves this property by proving co-location through a cache-based microarchitectural covert channel and furthermore including an SSA marker that will be overwritten on interrupt in each of the thread's transaction read set. HyperRace [40] later refined this co-location test mechanism to also protect against man-in-the-middle attacks by abusing contrived data races on a shared variable. Finally Heisenberg [237] proposes to leverage TSX, or alternatively minimal hardware extensions, to hook enclave resumptions and preload selected physical address translations in the processor's TLB before continuing the enclave application. This ensures that subsequent secret-dependent code or data accesses in the preloaded pages will be directly served from the TLB, without being observable through page-table side channels [277, 258].

Instruction set extensions. From the above research results and the attack systematization in Table 8.1, we can conclude that minimal hardware support for `eresume` interception would lift one of the key limitations of the current SGX architecture and may turn out to be an important enabler for a new class of hardware-assisted enclave software hardening techniques. As a requirement closely related to `eresume` interception, software defenses would furthermore benefit from a processor interface to efficiently and securely enforce that both SMT cores enter and exit the enclave in lock step. Indeed, a long line of side channels [180, 7, 29] and more recently also transient-execution [249, 223, 216, 251] attacks have decidedly shown that SMT should *not* be trusted across security domains. In the aftermath of Foreshadow [249], Intel finally extended remote attestation to reflect whether SMT has been disabled at boot time. Unfortunately, however, this binary solution precludes the use of SMT altogether. A more flexible hardware mechanism that exposes the SMT abstraction to the enclave software would allow to securely use this important performance optimization, as long as it can be guaranteed that both logical cores always execute in the same enclave protection domain.

Interestingly, a recent Intel patent [175] describes how an interrupt interception mechanism could be integrated into the SGX architecture. In this proposal, the `eresume` instruction is automatically converted into `eenter`, so as to enable the enclave to invoke a custom interrupt handler procedure, which can implement arbitrary software-defined policies before eventually resuming the previously interrupted enclave thread through a new `popssa` instruction. Likewise, another recent Intel patent [163] describes instruction set extensions to support an "event-notify" mode of operation where any cache evictions or interrupt events redirect execution to a user-level exception handler. In both proposals, the software handler can, for instance, be extended to prefetch selected enclave address translations into the TLB, which would eliminate page-table-based side channels under the assumption that the set of secret-dependent code and data pages can be easily identified for prefetching. We suppose that hardware requirements for `eresume` interception may be further simplified by imposing a restricted programming model for enclave exception handlers, where only start-to-end atomicity is guaranteed and any additional interrupts during execution of the handler merely revert the program counter to the start of the handler.

Apart from the above prefetching mechanism, we expect that `eresume` interception may proof especially useful as a low-cost anomaly detection technique that can be transparently applied in the enclave's trusted runtime. A straightforward security policy would already practically raise the bar by blocking at least the brutal approach of existing tools like SGX-Step [257], which often trigger several hundreds of thousands of rapid interrupts in a single enclaved execution. It remains important to recognize, however, that heuristic detection approaches will inevitably suffer from both false positives and false negatives. The former can impede practical deployability by, for instance, incorrectly classifying benign "interrupt storms", which are known to occur under heavy system load in real-world operating systems, as an artifact of an ongoing attack. False negatives, on the other hand, would likely trigger a continuous attacker-defender race where adversaries are expected to develop improved, ever-more stealthy techniques that stay under the radar of the enclave's interrupt detection policy (e.g., by invoking different copies of the same enclave). A final limitation of detection-oriented approaches is that it is not always clear which actions can be taken in response to a suspected attack attempt. Certain secrets are long-lived and cannot be straightforwardly deleted, e.g., attestation keys or user fingerprints, and they may already have been partially exposed at the time of detection. In summary, interrupt detection approaches appear promising as an incomplete first line of defense, but future research should tell which detection strategies would be effective and strike a

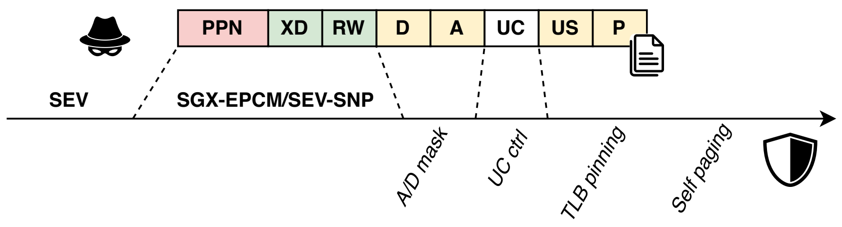


Figure 8.4: Spectrum of TEE resilience against address translation attacks, ordered by decreasing adversary control over enclave page-table attributes: from no restrictions in the SEV architecture, via physical address and accessibility checks in SGX and SEV-SNP, to academic proposals. The red page-table attributes (physical page number) affect enclaved execution integrity, green attributes (execute-disable, read-only) facilitate defense-in-depth access restrictions, and yellow attributes (accessed, dirty, supervisor, present) reveal enclave page accesses through side channels.

balance between performance, usability, and security.

Restricting adversary control over page tables

Conventional virtual address translation via page tables represents a major challenge across TEE designs [176, 58, 48, 11]. From a security perspective, the privileged adversary's control over untrusted page tables proved to be an important enabler in the SGX attack landscape. From the perspective of the operating system's traditional role as a resource manager, on the other hand, virtual memory represents an indispensable primitive for sandboxing and memory availability. This tension between security and usability has made page-table-based attacks very challenging to mitigate in a principled way, despite considerable research and industry attention. In fact, Fig. 8.4 shows that existing solutions span an entire spectrum: ranging from no protection at all, as in the current AMD SEV [136] architecture, via the trust-but-verify approach of Intel SGX [176], to ultimately moving page tables entirely in the enclave, as in the Sanctum [48] architecture. Since the processor's privileged paging interface was never designed with an inverse security model in mind, almost every field in an x86 page-table entry can be abused as an attack vector (cf. Table 4.1 on page 101). In the following, we move along this spectrum, and we recognize several opportunities for relatively low-cost and straightforward measures that could further reduce attack surface without necessarily moving all the way to the extreme end.

Physical address verification. Considering that kernel-level adversaries can arbitrarily modify page-table entries, TEE processors should at least verify the outcome of physical address translations while in enclave mode. At an abstract level, TEEs supporting untrusted virtual memory require some form of "back-translation" to ensure that every physical enclave page is only ever accessed from a corresponding, well-defined virtual address belonging to that enclave. If this is not the case, adversaries can straightforwardly modify page tables to fool a victim enclave to architecturally compute on code or data that belongs to an entirely different execution path. Such address-remapping attacks have, for instance, been repeatedly demonstrated [97, 183, 272, 274] to create arbitrary decryption oracles in current versions of the AMD SEV architecture, which only offers hardware-level memory encryption for virtual machines while leaving host page tables entirely under control of the untrusted hypervisor [136]. AMD recently announced SEV-SNP [11] processor extensions to thwart these attacks through a reverse-map table that back-translates every host-physical address to its expected guest-physical address. Interestingly, in contrast to SGX enclaves, SEV-SNP encrypted virtual machines remain in full control of internal guest page-table mappings. This may bring a considerable defensive advantage and somewhat raise the bar for side-channel adversaries, as the untrusted hypervisor now only observes guest physical page access patterns and should still somehow connect them to the virtual addresses referenced in the victim application.

The SGX architecture [176] natively protects against page-remapping attacks by maintaining explicit shadow EPCM entries in the processor to track ownership, type, expected virtual address, and permission metadata for every physical enclave page. A page fault is signaled to the untrusted operating system when accessing mapped pages that do not belong to the currently executing enclave, are accessed through an unexpected virtual address, or do not comply with the read/write/execute permissions imposed by the EPCM. SGX furthermore safeguards the consistency of address translation caches by flushing the TLB on enclave transitions and requiring all enclave threads to exit before evicting pages. However, despite all these measures, we showed in Chapter 7 that not all current SGX processors properly enforce EPCM restrictions in the transient domain, allowing to mount an innovative type of "inverse Foreshadow" transient page-remapping attacks known as LVI-L1D [251].

A/D attribute masking. Chapter 3 presented our research on stealthy page-table attacks [258], which for the first time called attention to the security implications of updating "accessed" and "dirty" bits in enclave mode. A long line of attacks [263, 257, 256, 254, 6, 182] has since abused A/D bits to spy on enclave memory access patterns without triggering page faults. Most notably, Nemesis [256] extended SGX-Step with fully deterministic zero-step filtering by observing that the accessed bit of the page-table entry mapping the current enclave code page is only ever set by the processor when the interrupt arrived after `eresume` and the enclave instruction has indeed been retired. CopyCat [182] furthermore abused this noiseless single-stepping primitive as a deterministic side channel to reveal intra-page conditional control flow by counting the number of enclave instructions between consecutive page accesses, as illustrated in Fig. 8.2 on page 235. Importantly, these research results clearly demonstrate that A/D bits, in combination with precise SGX-Step interrupts, provide a substantial *additional* advantage over prior page-fault adversaries [277] who merely clear the "present" bit to learn the coarse-grained sequence of enclave page accesses.

As a simple countermeasure, we hence recommend that SGX processors do not anymore update page-table accessed and dirty bits while in enclave mode. We expect that this would only have limited impact on memory-management functionality, as real-world operating systems appear to only make limited use of A/D bits anyway [223]. Furthermore, previous research [223, 47] suggests that the setting of accessed and dirty attributes is not implemented by the processor's page-miss handler silicon circuitry, but instead proceeds through microcode assists [74]. We hence presume that the responsible microcode procedure can be straightforwardly extended to not update A/D bits while in enclave mode. If this were the case, a microcode update could even be distributed that applies retroactively to existing SGX processors. An important consideration to assess the effectiveness of this countermeasure, however, is that we showed in Chapter 3 that enclave page access patterns can also be reliably deduced from the caching behavior of untrusted page-table entries. From this research we conclude that the root cause of enclave address translation side channels is the page-table walk in unprotected memory, and A/D masking would at best only reduce but decidedly *not* eliminate this attack surface.

Given the apparent simplicity of A/D masking, however, we expect that this countermeasure would still yield a high benefit-cost ratio for at least 3 reasons. First, monitoring page-table entries through the cache using FLUSH+FLUSH or FLUSH+RELOAD techniques is not completely noise-free and suffers from a reduced spatial granularity of 32 KiB, since one cache line can hold up to 8 adjacent page-table entries. If stealthiness is less of a concern, revoking access rights through the "present" bit and monitoring the associated page faults would still yield a 4 KiB spatial resolution, but at the cost of loosing the instruction-granular temporal dimension exploited by CopyCat. Adversaries would therefore likely be forced to fall back to a more involved and possibly weakened approach using a combination of SGX-Step interrupt counting and page fault marker events. Second, in the absence of page-table accessed bits that are set on instruction retirement, SGX-Step adversaries would loose their deterministic oracle to filter out zero-step observations when the APIC timer interrupt fires too early. The proposed A/D masking technique would hence break SGX-Step's perfect single-stepping functionality by introducing some zero-stepping noise. We do not expect that this in itself would pose a major hassle, since a well-configured APIC timer interval can achieve single-stepping ratios close to 99 % [257]. However, even a limited number of zero-step events may already complicate the correlation of repeated side-channel measurements for the same instruction over different enclave invocations. Third, A/D masking may be especially effective to complicate single-stepping timer interval configuration when applied in combination with the randomization-based hardening measures proposed in Section 8.3.1. In this respect, Fig. 8.3 on page 252 illustrated the alternative possibility of precise single-stepping through inter-processor interrupts. Chapter 3 indeed contributed a highly accurate Flush+Flush-based page-table monitoring technique to shoot down a victim enclave CPU at instruction-level granularity (cf. Table 3.1 on page 90). When applying Flush+Flush to monitor enclave page-table entries, we observed distinctly larger, highly recognizable timing differences. Interestingly, this observation is in notable contrast to the original Flush+Flush [87] attack, which suffers from considerable noise when spying on read-only shared memory mappings. Our subsequent analysis in Chapter 3 attributed this effect to implicit A/D updates by the enclave. Namely, our novel application of Flush+Flush for the first time spied on page-table memory that is concurrently being updated when the enclave processor sets the accessed bit, resulting in distinctly larger timing differences as the `clflush` instruction now needs to write back a *modified* cache line.[3] In light of these findings, we expect that the proposed A/D masking countermeasure would also offset our precise Flush+Flush spying technique, thereby forcing adversaries to fall back to significantly higher-latency Flush+Reload techniques that may not accommodate rapid single-stepping through inter-processor interrupts (cf. Fig. 3.3 on page 83).

Uncacheability control. Current SGX processors do not offer fine-grained cacheability control of enclave memory pages, only allowing the untrusted operating system to mark the *entire* physical memory range protected by SGX as either write-back or uncacheable at boot time [47]. The default write-back policy implies that all protected memory references by the enclave end up in the processor cache, enabling address metadata extraction through Prime+Probe side channels [181, 92, 29, 225, 79], or even direct data extraction through Foreshadow [249] on unpatched machines. Provably marking the entire enclave memory range as uncacheable would increase security, but expectedly at an

[3]Intel's software optimization manual [113] indeed confirms that "flushing cache lines in modified state are more costly than flushing cache lines in non-modified states".

unacceptable performance loss. A more flexible hardware-software interface would hence be desirable to allow trusted enclave software to selectively mark some of its own critical pages as uncacheable. We expect that extending current SGX processors with page-level uncacheability for enclaves may be relatively straightforward when leveraging existing hardware support for EPCM shadow entries and page-level cache-disable functionality.

Importantly, as with some of the previous proposals, selectively marking enclave pages as uncacheable may considerably raise the bar for attackers, but does not in itself represent a bullet-proof defense strategy. As evident from Table 8.1, adversaries can indeed fall back to alternative microarchitectural structures including DRAM row buffers [204, 263] or branch predictors [156, 105]. We argue, however, that processor support for trusted page-level uncacheability would offer a tempting and adaptable solution. Exposing cacheability, one of the major hidden and leaky microarchitectural optimizations, to the software level would ultimately provide enclave applications with more control over the performance vs. security trade-off. Interestingly, the idea of marking selected pages as uncacheable also echoes recent proposals for a new "non-speculative" memory type [222, 26, 239]. The ConTExT-light [222] solution, further discussed in the next section, indeed relies on the observation that uncacheable memory locations in recent Intel x86 processors can generally not be dereferenced in the transient-execution domain. Hence, a trusted processor interface to mark well-defined sensitive enclave pages, e.g., cryptographic keys and the state-save area holding interrupted register contents, as uncacheable would also enable a ConTExT-light approach to prevent these high-value secrets from leaking through Spectre or LVI-type unrestrained transient execution. One of the remaining challenges to strike a good balance between performance and security for this type of solution would be to accurately identify uncacheable candidate enclave pages that are either accessed in a secret-dependent fashion, e.g., lookup tables, or contain long-term secrets that should not be transiently dereferenced.

Self paging. Even with additional restrictions in place, enclave address translation side channels ultimately boil down to the tension between indispensable demand-paging interfaces for benign memory management OS functionality vs. information leakage through page-fault side channels. At the extreme end of the page-table defense spectrum are therefore more radical TEE research prototypes [48, 58] that place enclave page tables completely out of reach of the attacker. The Sanctum [48] architecture, for instance, allows enclaves to maintain their own virtual-to-physical mappings in a separate page-table hierarchy in enclave-private memory. However, the Sanctum design necessitates a trusted security monitor software layer plus additional hardware to verify that enclave physical address mappings always remain within selected DRAM regions owned by that particular enclave. Furthermore, when applying Sanctum's enclave-private page table design to modern x86 processors, it remains unclear how to securely integrate widespread virtualization extensions and particularly the extended page tables setup by the hypervisor.

Altogether, putting enclaves in full control of their own paging decisions would indeed eliminate an important side-channel attack vector, and may even preclude certain types of transient-execution attacks like LVI [251], but these solutions appear to sidestep the actual underlying problem. That is, moving page tables inside the enclave interferes with demand paging and the ability of the operating system to quickly regulate different users competing for scarce platform resources like encrypted EPC memory. In this respect, the more minimal hardware extensions for `eresume` interception and TLB prefetching [175, 237], discussed above, would strike a better balance that puts software in control without moving to the extreme end of the spectrum. TLB prefetching can indeed rule out page-table leakage for selected enclave pages, while still relying on the demand-paging interface for non-sensitive pages and allowing the operating system to arbitrarily swap out enclave pages at any time.

As a potentially more scalable alternative, Autarky [200] recently proposed minimal extensions to the SGX architecture, which would enable an enclave to retain control over its own paging decisions *without* moving page tables into enclave memory. In this proposal, the modified processor implements the A/D masking and `eresume` interception building blocks, described above, and additionally clears the reported page-fault address entirely. This forces the untrusted operating system to involve the enclave in its paging decisions, allowing the latter to hide private page accesses through any software-defined policy, e.g., ORAM, page clustering, or rate limiting [200]. While Autarky presents a compelling design that remains largely compatible with the existing SGX ISA, and indeed properly blocks controlled-channel attacks at the architectural level, an important limitation is that microarchitectural attack vectors remain out-of-scope. Particularly concerning in this respect is that Autarky explicitly leaves the untrusted page-table data structure in attacker-controlled memory. As shown in Chapter 3, this design decision allows adversaries to reliably deduce enclave-private page visits from the caching behavior of untrusted page-table entries [258]. While cache timing attacks in general could possibly be addressed through perpendicular measures, such as cache partitioning, these defenses may not always cover implicit memory accesses made by the processor's page-miss handler [259]. Page-table entries essentially reside in untrusted shared memory that is accessed by both the enclave and the operating system and would, hence, likely end up in the same cache partition.

Taming confused-deputy attacks via address-space restrictions

As a last opportunity for enclave hardening through the hardware-software interface, we observe that a considerable subset of the aforementioned SGX attacks takes advantage of the enclave's unrestricted access to unprotected memory to facilitate confused-deputy [93] attacks (cf. single-address-space column in Table 8.1). At an abstract level, these attacks first hijack execution inside the enclave through a variety of techniques, including memory-safety misbehavior [254, 154, 24], illegal transient data or control flow [39, 251], or even undervolting [188]. In the second phase of the attack, once the enclave has diverted from its intended execution path, the adversary abuses the victim enclave's unrestricted access to the unprotected address space to directly leak or encode secrets, or to setup fake data structures like a stack or linked list entries that allow to further steer the enclaved execution as desired. While none of these attacks critically relies on a shared address space and can be eradicated through different mitigations, experience tells that real-world hardware and software is not perfect. Hence, we advocate for a relatively simple hardware mechanism that would raise the bar for practical exploitation and allow enclave software to adhere to the principle of least privilege by restricting access rights to unprotected memory over time.

Particularly, we propose a new "enclave mode access prevention" processor feature that would allow enclave software to optionally disable access to unprotected memory outside the enclave after configuring a bit in the `rflags` x86 register. This would be similar in nature and purpose to the already existing Supervisor Mode Access Prevention (SMAP) processor extension that was added as an operating system hardening measure to hinder user-to-kernel confused-deputy attacks.[4] While SMAP is not intended to be a bullet-proof solution, it is known to practically raise the bar for memory-safety exploitation at a low cost. SMAP was furthermore recently shown to also be effective for impeding gadget-based transient-execution attacks [251, 36]. Since SGX already adds several checks to the address translation process while in enclave mode (cf. Fig. 3.1 on page 73), we expect that the implementation and runtime overhead of our proposed enclave mode access prevention feature would be minimal. Provided that the page-miss SGX logic is implemented in microcode [47], this functionality could even be retroactively applied to existing SGX processors. As a potential implementation caveat, unprotected address translations may have to be flushed from the TLB when enabling enclave mode access prevention. Alternatively, support for memory-protection keys in recent Intel processors [114]

[4]Note that the existing SGX architecture already precludes direct jumps to unprotected code outside the enclave, similar in effect to Supervisor Mode Execution Prevention (SMEP).

could potentially also be leveraged to dynamically restrict enclave accesses to the unprotected address space.

Note that prior work [268] has advocated for an inverse mechanism, where the operating system can restrict unprotected memory accesses to safeguard against potential enclave malware [219]. Both solutions aim to address complementary aspects of the SGX single-address-space design, and could even be combined into a unified solution that allows to configure restrictions from both the perspective of the enclave and the untrusted operating system.

8.3.3 Towards a notion of safe speculation

The threat of transient-execution attacks arose suddenly and, as a consequence, real-world defenses have so far mainly focused on stopping the bleeding. When setting out a longer-term road map for efficiently restraining transient execution, we want to reiterate the importance of differentiating Meltdown-type and Spectre-type threats. Where the former presents a critical mitigation challenge for current systems, the latter defines the research agenda for the coming decade. That is, building on the insights contributed in Chapters 6 and 7, Meltdown-type threats based on illegal data flow from faulting instructions will be efficiently addressed on the medium term, through relatively contained silicon-level design changes in the CPU pipeline. This is already evident from unaffected processor designs [13] and in the newest generations of hardened Intel processors [128]. Spectre-type threats, on the other hand, exploit indispensable CPU pipeline performance optimizations and, hence, represent an industry-wide, long-term challenge. In this regard, we argue that Spectre highlights the fallacy of not propagating down fine-grained security boundaries across abstraction levels and ultimately requires carefully redesigning the hardware-software boundary with a more explicit notion of security in mind.

The discussion below adopts some of Intel's recently refined transient-execution terminology [125] to reason about attack impact and mitigations. Particularly, the specifier "domain bypass" is reserved to denote the leakage aspect of prior Meltdown-type attacks and the "cross domain" specifier similarly refers to gadget-oriented exploitation with Spectre or LVI. We overview mitigation strategies according to the abstract phases in a transient-execution attack, as illustrated in Fig. 1.4 on page 11. Neutralizing any of these phases would in principle suffice to thwart practical attacks. A wide spectrum of possible defense avenues hence arises, each however with their security limitations and performance trade-offs.

Flushing or partitioning microarchitectural structures (phases 1, 4)

Transient-execution attacks bypass architectural access restrictions by exploiting shared microarchitectural elements between the attacker and the victim. They can therefore be limited to some extend by flushing or partitioning said elements when context switching between code of different security domains, similar to the side-channel hardening techniques discussed in Section 8.3.1. In the following, we consider three distinct ways in which microarchitectural isolation can impede certain attack variants: clearing buffers on entry to prevent cross-domain poisoning, on exit to prevent leakage, and when squashing the pipeline to hinder covert-channel transmission.

While defenses in this category have been proven to be very useful in practice for ruling out entire sub-classes of attacks, it is important to note that no amount of flushing or partitioning can offer full protection against transient-execution exploits. Indeed, some variants like Spectre-PHT [146] and LVI [251] proceed entirely in the victim domain, without relying on mistraining or secret encoding in the attacker domain. Ultimately, given suitable gadgets, NetSpectre [224] showed that even a fully isolated microarchitecture can be exploited through end-to-end timing differences for API calls.

Cross-domain poisoning. Flushing or partitioning global branch prediction history buffers is the preferred way to mitigate cross-domain branch target injection Spectre variants [146]. For this purpose, both Intel [128] and AMD [12] released microcode updates that extend the x86 ISA with mechanisms to restrict indirect branch speculation between applications, the kernel, and enclaves. This serves as a clear example of how transient-execution defenses benefit from extending the hardware-software contract with some notion of the underlying microarchitecture, so as to allow system software to restrict speculation across protection domains.

As a word of caution, however, clearing BTB and RSB structures on entry cannot rule out more advanced exploitation techniques which mistrain the indirect branch predictor entirely within the victim domain, e.g., speculative type confusion attacks [36].

Domain-bypass data extraction. With Foreshadow, introduced in Chapter 6, we were the first to show that address-space restrictions are fundamentally insufficient to rule out Meltdown-type microarchitectural data leakage. Instead, as discussed in Section 6.9, Foreshadow requires extensive mitigations that flush the L1D cache when context switching between enclaves or untrusted virtual machines. For this reason, Intel extended the ISA with a microcode mechanism that allows system software to flush the L1D cache on vulnerable processors [271, 107]. Likewise, in response to the more recent wave of MDS attacks [223, 216, 35], yet another microarchitectural primitive was added to the ISA, this time to allow flushing the line-fill buffers, store buffer, and load ports [108]. We can expect this tendency of flushing increasingly many microarchitectural buffers to continue for the coming time, at least until more principled silicon mitigations are in place to inhibit transient forwarding from faulting loads.

Our research on LVI, presented in Chapter 7, exposed the inherent limitations of the above microcode flush defenses by showing that, under certain adversarial conditions, Meltdown-type effects can also be inversely exploited in a gadget-oriented way. Essentially, LVI highlights that flushing leaky microarchitectural buffers on protection domain switches is a necessary, but not sufficient condition on current processors which transiently forward incorrect data from faulting or assisted load instructions.

Covert-channel transmission. Instead of clearing microarchitectural structures to restrict the attacker's access to residual secrets or her control over victim mispredictions, several recent studies [278, 138] have suggested to prevent transient instructions from creating covert channels altogether. These proposals have focused almost exclusively on preventing widespread cache-based covert channels by extending the microarchitecture with a small shadow structure that holds transient memory operations and is squashed after mispredictions.

While such approaches may indeed somewhat hinder exploitation with current techniques, it is important to note that closing one specific covert channel does not address the root cause of transient misbehavior. From the past decades of side-channel research, we can conclude that modern processors offer virtually endless possibilities to create covert channels. Several recent works have indeed demonstrated Spectre-style exploitation through non-cache-based covert channels, e.g., AVX units [224] or port contention [23].

Inhibiting the trigger for transient execution (phase 2)

The actual root cause for transient misbehavior is most evident in phase 2, where a "trigger instruction" causes all subsequent dependent operations to be eventually squashed. This trigger instruction can be either a faulting or assisted load instruction, or a mispredicted branch or data dependency. Since any transient operations following the trigger instruction reflect potentially dangerous computations, out of the program's intended code or data paths, inhibiting them from occurring in the first place would be the most solid defense to entirely eradicate transient-execution attacks.

We argue below that, in the case of Meltdown-type threats, preventing transient data flow from faulting loads indeed appears to be a reasonable requirement: the fault condition can be detected before forwarding the data, and the load and all of its dependent operations will have to be replayed anyway. In the case of Spectre, on the other hand, disabling speculation entirely may come with an unacceptable performance cost for commodity computers and servers. Current Spectre mitigations therefore only selectively disable speculation, by inserting explicit `lfence` serialization barriers after identifying vulnerable code patterns.

Blocking transient data flow. The root cause for Meltdown-type threats, including Foreshadow and LVI discussed in Chapters 6 and 7, needs to be ultimately addressed through silicon-level design changes in future processors. Particularly, the hardware has to ensure that no illegal data flows from faulting or assisted load micro-ops exist at the microarchitectural level. That is, no transient computations depending on a faulting or assisted instruction are allowed. We believe this is already the behavior in certain ARM and AMD processors, where a faulting load does not forward any data [13]. As an important contribution, the LVI-NULL attack variant in Chapter 7 showed that it does *not* suffice to merely zero out the forwarded value, as is the case in some acclaimed Meltdown-resistant recent Intel processors enumerating `RDCL_NO` [128]. We expect that these findings will contribute to improved silicon-level mitigations in future processors. In fact, Intel [110] explains that more recent processors exhibit "zero-at-ret" behavior, where the zero dummy value is only forwarded to dependent operations when the faulting load is the next instruction to retire. Intel claims that, on these processors, LVI-NULL cannot be exploited in practice, due to the extremely narrow size of the transient-execution window.

To date, the only known transient-execution attack that abuses illegal transient data flow not originating from faulting or assisted load instructions is Spectre-STL [102]. This variant abuses that the processor's memory disambiguation logic may optimistically predict that a load does not have a dependency on a prior store, even before all prior store addresses are known. In case of a misprediction, transient instructions may speculatively bypass a prior store and compute on unsanitized stale values. Importantly, in contrast to Meltdown-type threats, Spectre-STL exploits an important performance optimization that may not be desirable to fully eliminate in future processors. Instead, Intel [128] released a microcode update which allows the operating system to selectively disable memory disambiguation prediction for critical applications. Furthermore, the microcode update entirely disables memory disambiguation prediction, and hence eliminates Spectre-STL threats, while in enclave mode.

Serializing transient control flow. The recommended way to mitigate pervasive Spectre-PHT [146] speculative out-of-bounds access vulnerabilities is to insert a serializing instruction, like `lfence`, to stall the processor pipeline after potentially mispredicted directional branches [128]. Since serializing every branch may come with unacceptable performance overheads, current solutions focus on manually or semi-automatically identifying vulnerable code patterns, e.g., conditional array indices. Unfortunately, for sizable real-world code bases, this rather ad-hoc mitigation approach represents a vast and error-prone effort. Over the last two years, a steady and ongoing stream of Spectre-PHT vulnerabilities has been manually identified and patched in, for instance, the Intel SGX SDK [117] and the Linux kernel [36]. Similar to how programmers have been trained over the past decades to insert explicit conditional branches to properly shield array accesses, they are now expected to additionally serialize the processor pipeline before performing potentially dangerous speculative out-of-bounds computations. While compile-time static analysis techniques may assist in automating this process, striking a suitable balance between performance and protection appears to be particularly challenging, and initial static analysis tools have been shown to miss many exploitable gadgets [144].

Constraining transient data accesses (phase 3)

Current short-term solutions to mitigate Spectre impose poor trade-offs between performance and security. Namely, by flushing valuable branch prediction history and stalling the CPU pipeline, they tend to disable some of the very optimizations that have driven processor performance gains over the past decades. A promising approach on the longer term therefore is to investigate solutions that still allow processors to freely speculate, while constraining transient computations to some safe limits. Ideally, these solutions would constrain the processor pipeline in such a way that transient instructions never compute on secrets which would not be referenced in a plausible architectural execution, while still allowing useful computations that can be committed to the nominal CPU state in case the prediction turns out to be correct.

The key discrepancy exploited by Spectre is that current CPU microarchitectures are not sufficiently aware of the programmer's intentions at the source code level. This is especially apparent for the ubiquitous Spectre-PHT speculative bounds check bypass variant. That is, programmers and compilers reason about the security of programs at a fine-grained level, e.g., in terms of individual array boundaries, whereas mainstream Instruction Set Architectures (ISAs) only support coarse-grained expression of protection domains through virtual address spaces. This leaves too much freedom to the transient execution, essentially every accessible secret in the current process's address space can be leaked via confused-deputy gadgets, and forces unfortunate compromises, like site isolation [212] where every website executes in its own process. What we need, therefore, is a more explicit notion of protection domains that cross-cuts the system stack and allows to communicate fine-grained security constraints, like array lengths, from the level of the programming language to the target processor architecture. Such a richer architectural expression of security boundaries could be propagated into the CPU pipeline and has the potential to enable true hardware-software co-design, where microarchitects can constrain optimizations according to the security requirements of the application. In the following, we overview two concrete proposals in this respect.

Non-transient memory mappings. ConTExT [222], and concurrently also related industry proposals [26, 239], recently outlined minimal changes to the x86 architecture which would enable software to communicate access restrictions to the microarchitectural transient-execution domain. Particularly, ConTExT-enabled processors extend the hardware-software contract with a new "non-transient" page-table attribute and the guarantee that non-transient memory locations are never propagated to transient computations. To prevent leakage of secret data that is already loaded in registers, the processor furthermore taints CPU registers holding non-transient memory, and ConTExT-aware compilers are expected to maintain a separate non-transient stack when spilling CPU registers. Building upon this contract, a comprehensive hardware-software co-design solution can provide protection across all layers of the system stack. Application developers are expected to annotate high-value secrets, which subsequently will be allocated in non-transient pages by the aid of the compiler, linker, and operating system. ConTExT hence transforms the challenge of defending against Spectre from the error-prone effort of locating and serializing all possible gadgets to the appropriate annotation of critical secrets that should never be exposed to the transient execution.

In contrast to today's `lfence` software mitigations, a ConTExT-like approach maintains the benefits of important speculative execution optimizations by still allowing unrestricted transient access to non-critical data, without entirely stalling the CPU pipeline for every critical branch decision. From an implementation perspective, processors could likely leverage already existing support for non-cacheable memory types, which also serves as an over-approximation of the expected performance overhead [222]. While solutions like ConTExT provide a promising avenue forward, they critically rely on the correct annotation of relatively sparse secrets, and it remains to be investigated to what extend application developers or automated compiler approaches can accurately identify secrets in larger and general-purpose applications.

Capability architectures. Capabilities are a long-standing primitive in computer security and have more recently been revived in the CHERI [275] research processor, which implements a hybrid memory capability scheme to enforce fine-grained protection domains inside the virtual address space of a conventional process. On a capability machine, pointers are represented at runtime as unforgeable objects carrying associated permissions and length fields. Applications are only allowed to access the memory regions described by their current set of capability registers, and the CPU enforces that every memory access remains within the bounds of the associated capability. While initially designed to eradicate traditional memory-safety issues, like buffer overflows, Watson et al. [264] argue that microarchitectures for CHERI-based processors in principle hold all of the necessary information to ensure that speculative loads remain within the architecturally defined limits. In the common case of accessing successive array elements in a loop, for instance, the processor is free to speculatively access memory locations for future loop iterations, as long as the indices remain within the array bounds specified by the corresponding memory capability register.

In contrast to today's `lfence` software mitigations that unconditionally stall the CPU pipeline after sensitive branches, this scheme elegantly allows the processor to continue when the branch is correctly predicted, only stalling in the very last loop iteration that attempts to access out-of-bounds memory. In this regard, through their richer architectural expression of security boundaries, capability processors like CHERI hold the promise of propagating a notion of memory safety from the architectural level down to the microarchitectural transient-execution domain. However, a number of important challenges for this approach remain to be investigated, such as the security implications of transiently computing on inadvertent in-bounds memory locations when considering for instance an array of function pointers, the influence of capability checks on the processor's critical path, the general integration with out-of-order execution and serialization of capability register operations, and the handling of indirect branches.

8.4 Concluding thoughts

The start of this PhD trajectory happened to coincide approximately with the public release of Intel SGX. The advent of SGX was broadly acclaimed for bringing strong hardware-enforced trusted computing guarantees to mass consumer devices and for protecting end-user data in an untrusted cloud environment. Over the past years, however, a significant understanding of the microarchitectural limitations of this technology has been built up by the research community, in synergy with the results presented in this dissertation. This ongoing line of work nuances the security and privacy guarantees promised by today's TEEs and, more broadly, requires us to question our understanding of system security in general.

The security community has traditionally assessed the trustworthiness of applications at the software level by reasoning about source code as if it were executed on an idealized, abstract computing platform. In this regard, TEEs can be considered a real leap forward, as they allow to drastically reduce the trusted computing base by abstracting away the underlying operating system and supporting software. However, the recent wave of microarchitectural attacks shows once again that abstraction levels are only relative in the eyes of attackers. Perhaps the most important takeaway is that today's processors can no longer be considered opaque black boxes. It is, therefore, time to take TEE protection to the next level by maintaining strict enclave isolation at all levels of the system stack—not only at the architectural level but also in the CPU microarchitecture itself. Building such next-generation security architectures for the post-Spectre world will likely require rethinking the hardware-software interface and, thus, poses a challenge for the decades to come.

Appendix A

Transient-execution classification tree

This appendix provides a state-of-the-art classification tree for Meltdown-type attacks that have been demonstrated to date. The tree in Fig. A.1 is based on our original systematization of the transient-execution attack landscape [36], which has since been extended and maintained at `https://transient.fail/`.

In this systematization, the Foreshadow [249] attack, presented in Chapter 6, appears under the canonical name “Meltdown-P-L1”. Furthermore, LVI [251] attacks, presented in Chapter 7, are considered to be of “inverse Meltdown” type and hence principally cover the entire tree. Figure E.1 on page 295 provides an orthogonal classification tree and canonical naming scheme which covers the most important LVI attack variants.

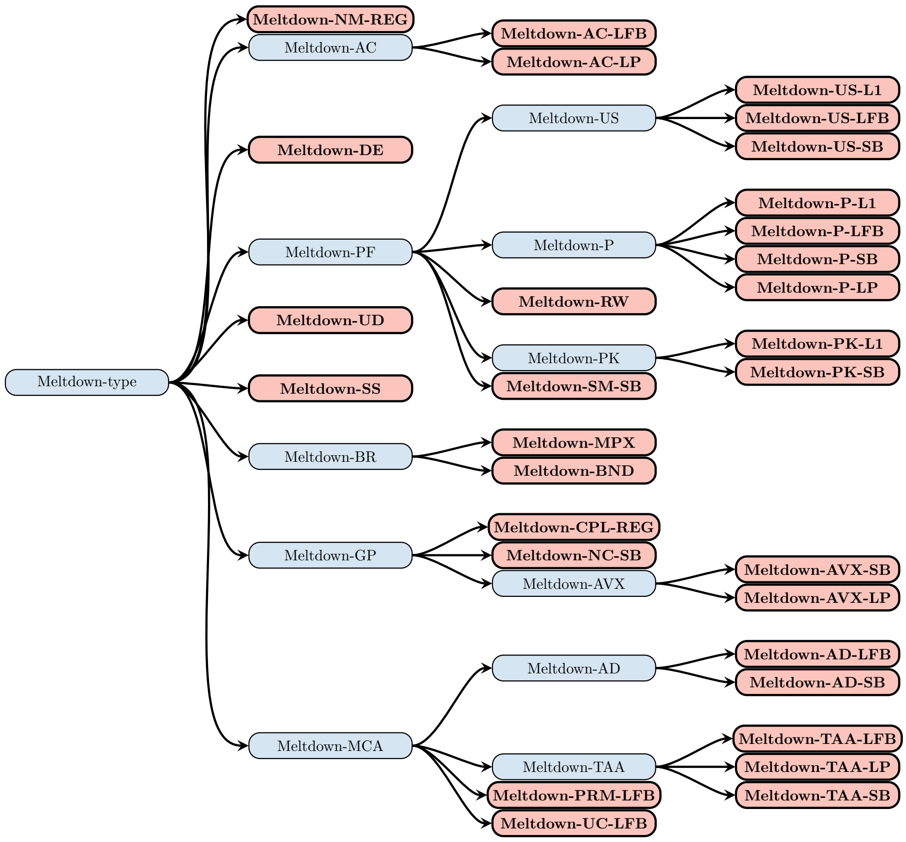


Figure A.1: State-of-the-art classification tree for Meltdown-type attacks [36].

Appendix B

Enclave shielding runtime vulnerable code samples

This appendix provides selected code snippets to demonstrate some of the real-world interface sanitization vulnerabilities found in the various enclave shielding runtime libraries studied in Chapter 2. For all of the vulnerabilities which are marked to have a proof-of-concept exploit in Table 2.1 on page 40, a reference to the vulnerable software version and full attack code can furthermore be retrieved at `https://github.com/jovanbulck/0xbadc0de`.

B.1 OE legacy ecall dispatcher

The (legacy) `ecall` interface `_handle_call_enclave()` does not validate that `arg_in.args` points outside the enclave. While this pointer is subsequently checked by the `oeedger8r`-generated entry code, an error code is still written to the in-enclave memory location on failure (cf. Listing 2.2). After our report, the legacy `handle_call_enclave()` dispatcher has been removed completely.

```
static oe_result_t _handle_call_enclave(uint64_t arg_in) {
  oe_call_enclave_args_t args, *args_ptr;
  ...
  if (!oe_is_outside_enclave((void*)arg_in,
                             sizeof(oe_call_enclave_args_t)))
    OE_RAISE(OE_INVALID_PARAMETER);
  args_ptr = (oe_call_enclave_args_t*) arg_in;
  args = *args_ptr;
  ...
```

```
★ func(args.args);
  ...
```

Listing B.1: `https://github.com/Microsoft/OpenEnclave/blob/93ac313a/enclave/core/sgx/calls.c#L216`

B.2 OE built-in attestation ecall

Evidently, a check that validates that `arg_in` points outside the enclave was overlooked. We thus can overwrite in-enclave memory through the write to `host_arg->result`. Note that the target buffer has to have a certain size to avoid segfaults in the function `_oe_get_local_report()` that is called within `_handle_get_sgx_report()` (this is because the parameter `oe_get_sgx_report_args_t` is a large struct). Because of that, `_oe_get_local_report()` will very likely fail with the return value `OE_INVALID_PARAMETER` (`0x3`) and overwrite the first four bytes of the memory at `host_arg` with `0x03000000`.

```
oe_result_t _handle_get_sgx_report(uint64_t arg_in) {
  oe_result_t result = OE_UNEXPECTED;
  oe_get_sgx_report_args_t* host_arg =
      (oe_get_sgx_report_args_t*)arg_in;
  oe_get_sgx_report_args_t enc_arg;
  size_t report_buffer_size = sizeof(sgx_report_t);

  if (host_arg == NULL)
    OE_RAISE(OE_INVALID_PARAMETER);

  // Validate and copy args to prevent TOCTOU issues.
★ enc_arg = *host_arg;

  OE_CHECK(_oe_get_local_report(NULL, 0,
    (enc_arg.opt_params_size != 0) ? enc_arg.opt_params : NULL,
    enc_arg.opt_params_size, (uint8_t*)&enc_arg.sgx_report,
    &report_buffer_size));

★ *host_arg = enc_arg;
  result = OE_OK;
done:
  if (host_arg)
★   host_arg->result = result;
  return result;
}
```

Listing B.2: `https://github.com/microsoft/OpenEnclave/blob/93ac313a/enclave/core/sgx/report.c#L388`

B.3 Asylo ecall entry point

Asylo's trusted `ecall` dispatcher is declared in Intel SGX SDK EDL specification as follows: `public int ecall_dispatch_trusted_call(uint64_t selector, [user_check] void *buffer)`. However, in the code below, it becomes apparent that the `[user_check]` argument buffer is never properly validated before being unmarshalled. This issue can most easily be mitigated by properly declaring the argument buffer using `edger8r`'s `[in]` pointer attribute instead of the problematic `[user_check]` attribute. Further, the validation logic at line 16 contains a logic mistake which incorrectly assumes that outside == ¬inside (cf. Section 2.5.3).

```
 1 int ecall_dispatch_trusted_call(uint64_t selector, void *buffer) {
 2   return asylo::primitives::asylo_enclave_call(selector, buffer);
 3 }
 4
 5 int asylo_enclave_call(uint64_t selector, void *buffer) {
 6   SgxParams *const sgx_params = reinterpret_cast<SgxParams *>(buffer);
 7
 8 ★ const void *input = sgx_params->input;
 9 ★ size_t input_size = sgx_params->input_size;
10 ★ sgx_params->input = nullptr;
11 ★ sgx_params->input_size = 0;
12   void *output = nullptr;
13   size_t output_size = 0;
14
15   if (input) {
16 ★   if (TrustedPrimitives::IsTrustedExtent(input, input_size)) {
17       PrimitiveStatus status{error::GoogleError::INVALID_ARGUMENT,
18                              "input should lie within untrusted memory."};
19       return status.error_code();
20     }
```

Listing B.3: `https://github.com/google/asylo/blob/e4810bdbac/asylo/platform/primitives/sgx/trusted_sgx.cc#L98`

B.4 SGX-LKL `SIGILL` signal handler exploit

SGX-LKL intercepts the `SIGILL` (undefined instruction) to handle instructions like `rdtsc` inside the enclave. In this case, the host executes `rdtsc` and the result is passed back into the enclave through the enclave's signal handler interface. In case of `SIGILL`, an adversary can change the untrusted `siginfo` argument to point into the enclave, which will then yield the memory contents at that location as the 64-bit result of `rdtsc`, as shown by our PoC. This specific vulnerability can only be exploited if the target in-enclave memory starts with

`0x04000000` (*i.e.*, `siginfo->signum == SIGILL`). In addition, the `rdtsc` result needs to be outputted back to the untrusted side (e.g., our PoC simply prints it to the terminal). Note that adversaries can also use the in-enclave signal handler's execution itself as a side-channel. Depending on the contents of the memory pointed to by `siginfo->signum` different code paths are taken, so established side-channel approaches may reconstruct the secret-dependent control through differences in timing [180], page tables [277, 258], or other microarchitectural elements [156, 256].

```
void __enclave_signal_handler(gprsgx_t *regs,
                              enclave_signal_info_t *siginfo) {
    ...
    int ret;
★    switch (siginfo->signum) {
    case SIGSEGV:
★        ret = handle_sigsegv(regs, siginfo->arg);
        break;
    case SIGILL:
★        ret = handle_sigill(regs, siginfo->arg);
        break;
    default:
        ret = -1;
    }
    ...
```

Listing B.4: `https://github.com/lsds/sgx-lkl/blob/664eb25a/src/sgx/enclave_signal.c#L17`

B.5 Sancus authentic execution stub

Passing a ciphertext pointer argument that points inside the enclave may unintentionally decrypt enclave memory, potentially leading to information disclosure. Interestingly, we observed that untrusted array index arguments were properly sanitized to safeguard against well-understood buffer overflow vulnerabilities.

```
void SM_ENTRY __sm_handle_input(uint16_t conn_id,
                         const void* payload, size_t len)
{
  if (conn_id >= SM_NUM_INPUTS) return;

  size_t data_len = len - AD_SIZE - SANCUS_TAG_SIZE;
★  uint8_t* cipher = (uint8_t*)payload + AD_SIZE;
★  uint8_t* tag = cipher + data_len;

  uint8_t* input_buffer = alloca(data_len);

★ if (sancus_unwrap_with_key(__sm_io_keys[conn_id],
                             payload, AD_SIZE, cipher,
                             data_len, tag, input_buffer))
  {
   __sm_input_callbacks[conn_id](input_buffer, data_len);
  }
}
```

Listing B.5: `https://github.com/sancus-tee/sancus-compiler/blob/5d5cbff/src/stubs/sm_input.c#L7`

B.6 Sancus trusted loader enclave

Sancus may optionally safeguard code confidentiality in a two-stage approach [78]. In the first stage, a trusted infrastructural "loader enclave" is deployed. Next, the loader enclave can be called by multiple mutually distrusting clients to securely decrypt and load second-stage application enclaves. However, the code for the loader enclave below lacks input pointer validation. This allows untrusted software developers to build an arbitrary write primitive in the shared infrastructural loader enclave. Specifically, after successful decryption, the loader proceeds to copy the adversary-controlled plaintext application enclave to its intended destination in the single-address-space. After a malicious software provider has successfully hijacked the loader enclave, all code confidentiality guarantees for future application enclaves from other providers are lost.

```
int sm_loader_load(struct SancusCryptModule *scm) {
★  size_t pstart   = (size_t)scm->public_start;
★  size_t pend     = (size_t)scm->public_end;
★  size_t pcstart  = (size_t)scm->public_start_crypt;
★  size_t pcend    = (size_t)scm->public_end_crypt;
   size_t i;
   int ret;

   /* check boundaries */
   if (pend < pstart || pcend < pcstart)
       return 0;
   /* check sizes */
   if ((pend - pstart) != (pcend - pcstart))
       return 0;
   /* check if sizes are a multiple of AES block size */
   if ((pend - pstart) % SM_LOADER_KEYLEN != 0)
       return 0;
   /* get scm->name length */
★  for (i = 0; scm->name[i] != '\0'; i++);
   /* derive decryption key */
   if (hmac_sign(key, scm->name, i) == 0)
       return 0;
   /* derive the IV for CCM mode */
```

```
    if (hmac_sign(iv, key, SM_LOADER_KEYLEN) == 0)
        return 0;
    /* decrypt module and check integrity in CCM mode */
★   if (decrypt(key, iv, scm->public_start_crypt,
                scm->public_start, pend - pstart) == 0)
        return 0;
    /* protect the module (should prevent read access) */
    ret = protect_sm((struct SancusModule *)scm);

    return ret;
}
```

Listing B.6: `https://github.com/sancus-tee/soteria-test/blob/3551360/crypttest_fpga/sm_loader.c#L49`

B.7 OE string ecall edge wrapper

As part of OE's "deep copy" marshalling scheme, the `_handle_call_enclave_function()` from the trusted runtime properly copies the entire marshalled input buffer into the enclave (including the string argument and alleged length which are put into the serialized `input_buffer` by the untrusted runtime). The `oeedger8r` bridge then takes care to redirect all pointers to the marshalled input buffer. However, when doing so the auto-generated `oeedger8r` entry code below does *not* explicitly null-terminate the untrusted string argument. Hence, the trusted user function will incorrectly assume that the string is properly terminated and may perform out-of-bounds memory read/writes beyond the end of the string.

```
void ecall_my_ecall(uint8_t* input_buf,
    size_t input_buf_size, uint8_t* output_buf,
    size_t output_buf_size, size_t* output_bytes_written)
{
  oe_result_t _result = OE_FAILURE;
  /* NOTE: output buf code removed for sake of space */
  my_ecall_args_t* pargs_in =(my_ecall_args_t*) input_buf;
  size_t input_buf_offset = 0;

  /* Make sure buffers lie within the enclave */
  OE_ADD_SIZE(input_buf_offset, sizeof(*pargs_in));
  if (!input_buf || !oe_is_within_enclave(input_buf, input_buf_size))
     goto done;
  /* OE_SET_IN_POINTER(s, s_len * sizeof(char)) */
  if (pargs_in->s) {
★    *(uint8_t**)&pargs_in->s = input_buf + input_buf_offset;
    OE_ADD_SIZE(input_buf_offset, (size_t)(s_len*sizeof(char)));
    if (input_buf_offset > input_buf_size) {
        _result = OE_BUFFER_TOO_SMALL;
        goto done;
    }
  }
  oe_lfence();               /* lfence after checks */
★ my_ecall(pargs_in->s);     /* Call user function */
  ...
}
```

Listing B.7: Proxy function generated by `oeedger8r` for the EDL specification: `public void my_ecall([in,string] char *s)`.

B.8 Keystone integer overflow

We discovered a potential vulnerability that originates from an integer overflow in the `detect_region_overlap()` function which is used during the process of creating an enclave. Evidently, there is no check to guarantee that the integer additions do not overflow. Suppose that `epm_base = 0x82800000` and `epm_size = 100000`. If one passes `addr=0x1` and `size = 0xffffffffffffffff`, there is an overlap between both regions. However, when these values are put into the above condition, this evaluates to "no overlap" (zero). The above issue was not exploitable at the time of discovery: various constraints imposed on the size prevented the exploitation of this issue, but it might have been problematic in the future if the overlap check was used in different parts of the code.

```
static int detect_region_overlap(uintptr_t addr, uintptr_t size)
{
  ...
★ region_overlap |= ((uintptr_t) epm_base < addr + size)
        && ((uintptr_t) epm_base + epm_size > addr);
  ...
```

Listing B.8: `https://github.com/keystone-enclave/riscv-pk/blob/e24d47c/sm/pmp.c#L71`

Appendix C

Additional resources for Nemesis attacks

This appendix provides the full instruction timing tables for the MSP430 microprocessor, plus source code and compiled assembly for the Nemesis interrupt latency timing attacks presented in Chapter 5. All of the attack code and case study applications for both Intel SGX and Sancus were made open-source available at `https://github.com/jovanbulck/nemesis`. For the Sancus attacks, continuous integration has furthermore been setup in a cycle-accurate simulator which can be viewed at `https://travis-ci.org/jovanbulck/nemesis`.

C.1 MSP430 instruction cycles

This appendix provides the full instruction timings for the MSP430 architecture, as published by Texas Instruments [243]. All jump instructions require two clock cycles to execute, regardless of whether the jump is taken or not. The number of CPU cycles required for other instructions depends on the addressing modes of the source and destination operands, not the instruction type itself. Tables C.1 and C.2 list the number of cycles for respectively single and double operand instructions. Note that a number of MSP430 assembly operations (including `nop`, `incd`, `rla`, `ret`, and `tst`) are emulated by means of the listed machine instructions.

Table C.1: MSP430 single operand instruction cycles.

Addressing Mode	No. of Cycles RRA, RRC, SWPB, SXT	PUSH	CALL	Example
Rn	1	3	4	`SWPB R5`
@Rn	3	4	4	`RRC  @R9`
@Rn+	3	5	5	`SWPB @R10+`
#N	–	4	5	`CALL #0F000H`
x(Rn)	4	5	5	`CALL 2(R7)`
EDE	4	5	5	`PUSH EDE`
&EDE	4	5	5	`SXT  &EDE`

C.2 Secure keypad implementation

The enclaved keypad program below was derived from a recently published open-source[1] automotive Sancus application scenario [252], which we had to minimally modify in order to run without function callbacks in a stand-alone enclave.

The start-to-end timing of the `poll_keypad` function only reveals the number of times the if statement was executed, *i.e.*, the number of keys that were down (cf. return value). By carefully interrupting the function each loop iteration, an untrusted ISR can learn the value of the secret PIN code.

```c
int        SM_DATA(secure) init      = 0x0;
int        SM_DATA(secure) pin_idx   = 0x0;
uint16_t   SM_DATA(secure) key_state = 0x0;
char       SM_DATA(secure) pin[PIN_LEN];
const char SM_DATA(secure) keymap[NB_KEYS] =
{
  '1', '4', '7', '0', '2', '5', '8', 'F',
  '3', '6', '9', 'E', 'A', 'B', 'C', 'D'
};

int SM_ENTRY(secure) poll_keypad( void )
{
  int is_pressed, was_pressed, mask = 0x1;

  /* Securely initialize SM on first call. */
  if (!init) return do_init();

  /* Fetch key state from MMIO driver SM. */
```

[1] https://github.com/sancus-tee/vulcan/blob/master/demo/ecu-tcs/sm_tcs_kypd.c

Table C.2: MSP430 double operand instruction cycles.

Addressing Mode		**No. of**	
Src	**Dst**	**Cycles**	**Example**
Rn	Rm	1	`MOV R5,R8`
	PC	2	`BR R9`
	x(Rm)	4	`ADD R5,4(R6)`
	EDE	4	`XOR R8,EDE`
	&EDE	4	`MOV R5,&EDE`
@Rn	Rm	2	`AND @R4,R5`
	PC	2	`BR @R8`
	x(Rm)	5	`XOR @R5,8(R6)`
	EDE	5	`MOV @R5,EDE`
	&EDE	5	`XOR @R5,&EDE`
@Rn+	Rm	2	`ADD @R5+,R6`
	PC	3	`BR @R9+`
	x(Rm)	5	`XOR @R5+,8(R6)`
	EDE	5	`MOV @R9+,EDE`
	&EDE	5	`MOV @R9+,&EDE`
#N	Rm	2	`MOV #20, R9`
	PC	3	`BR #2AEH`
	x(Rm)	5	`MOV #0300H,0(SP)`
	EDE	5	`ADD #33,EDE`
	&EDE	5	`ADD #33,&EDE`
x(Rn)	Rm	3	`MOV 2(R5),R7`
	PC	3	`BR 2(R6)`
	x(Rm)	6	`ADD 2(R4),6(R9)`
	EDE	6	`MOV 4(R7),EDE`
	&EDE	6	`MOV 2(R4),&EDE`
EDE	Rm	3	`AND EDE,R6`
	PC	3	`BR EDE`
	x(Rm)	6	`MOV EDE,0(SP)`
	EDE	6	`CMP EDE,EDE`
	&EDE	6	`MOV EDE,&EDE`
&EDE	Rm	3	`MOV &EDE,R8`
	PC	3	`BR &EDE`
	x(Rm)	6	`MOV &EDE,0(SP)`
	EDE	6	`MOV &EDE,EDE`
	&EDE	6	`MOV &EDE,&EDE`

```
  uint16_t new_key_state = read_key_state();

  /* Store down keys in private PIN array. */
  for (int key = 0; key < NB_KEYS; key++)
  {
    is_pressed  = (new_key_state & mask);
    was_pressed = (key_state & mask);
    if (is_pressed
        /* INTERRUPT SHOULD ARRIVE HERE */
        && !was_pressed && (pin_idx < PIN_LEN))
    {
      pin[pin_idx++] = keymap[key];
    }
    /* .. OR HERE. When configuring the timer
    for the key comparison in the next loop
    iteration, ISR should take into account
    key presses from previous runs to be able
    to detect key releases. */
    mask = mask << 1;
  }
  key_state = new_key_state;

  /* Return the number of characters still
  to be entered by the user. */
  return (PIN_LEN - pin_idx);
}
```

Listing C.1: Secure keypad Sancus enclave.

For completeness, we also provide a disassembled version of this function, as compiled with LLVM/Clang v3.7.0.

```
poll_keypad:
   push    r4
   mov     r1, r4
   push    r11
   push    r10
   push    r9
   tst     &init
   jz      3f
   call    #read_key_state
   mov     #1, r12
   clr     r13
   mov     &key_state, r14
1: mov     &pin_idx, r11
   cmp     #4, r11
   jge     2f
   mov     r12, r10
   and     r15, r10
   tst     r10               ; test key state
   jz      2f                ; V no. of cycles
   mov     r14, r10          ; 1
   and     r12, r10          ; 1
```

```
    tst     r10                ; 1
    jnz     2f
    mov.b   518(r13), r10
    mov     r11, r9
    inc     r9
    mov     r9, &pin_idx
    mov.b   r10, 550(r11)      ; V no. of cycles
2:  rla     r12                ; 1
    incd    r13                ; 1
    cmp     #32, r13           ; 2
    jnz     1b
    mov     r15, &key_state
    mov     #4, r15
    sub     &pin_idx, r15
    jmp     4f
3:  call    #do_init
4:  pop     r9
    pop     r10
    pop     r11
    pop     r4
    ret
```

Listing C.2: Secure keypad Sancus enclave (compiled assembly).

C.3 Intel SGX SDK binary search implementation

In this appendix, we provide the full C source code of the `bsearch` function from the trusted in-enclave `libc` in the official Intel SGX Linux SDK v2.1.2 (`linux-sgx/sdk/tlibc/stdlib/bsearch.c`).

```
/*
 * Copyright (c) 1990 Regents of the University
 * of California. All rights reserved.
 */

#include <stdlib.h>

/*
 * Perform a binary search.
 *
 * The code below is a bit sneaky.  After a
 * comparison fails, we divide the work in half
 * by moving either left or right. If lim is
 * odd, moving left simply involves halving
 * lim: e.g., when lim is 5 we look at item 2,
 * so we change lim to 2 so that we will look
 * at items 0 & 1.  If lim is even, the same
 * applies.  If lim is odd, moving right again
 * involves halving lim, this time moving the
 * base up one item past p: e.g., when lim is 5
 * we change base to item 3 and make lim 2 so
 * that we will look at items 3 and 4.  If lim
 * is even, however, we have to shrink it by
 * one before halving: e.g., when lim is 4, we
 * still looked at item 2, so we have to make
 * lim 3, then halve, obtaining 1, so that we
 * will only look at item 3.
 */

void *
bsearch(const void *key, const void *base0,
    size_t nmemb, size_t size,
    int (*compar)(const void *, const void *))
{
   const char *base = (const char *)base0;
   size_t lim; int cmp; const void *p;

   for (lim = nmemb; lim != 0; lim >>= 1) {
      p = base + (lim >> 1) * size;
      cmp = (*compar)(key, p);
      if (cmp == 0)
         return ((void *)p);
      if (cmp > 0) {  /* key > p: move right */
         base = (char *)p + size;
         lim--;
      } /* else move left */
   }
   return (NULL);
}
```

Listing C.3: Binary search routine.

Since Nemesis-type IRQ latency attacks exploit information leakage at an instruction-level granularity, we also provide a disassembled version of this function, as compiled with gcc v5.4.0.

```
bsearch:
   push   %r15
   push   %r14
   push   %r13
   push   %r12
   push   %rbp
   push   %rbx
   sub    $0x18,%rsp
   test   %rdx,%rdx
   mov    %rdi,0x8(%rsp)
   je     3f
   mov    %rsi,%r12
   mov    %rdx,%rbx
   mov    %rcx,%rbp
   mov    %r8,%r13
   jmp    2f

```

```
; base = (char *)p + size; lim--;
1: sub     $0x1,%rbx
   lea     (%r14,%rbp,1),%r12
   shr     %rbx
   test    %rbx,%rbx
   je      3f

; for (lim = nmemb; lim != 0; lim >>= 1)
2: mov     %rbx,%r15
   mov     0x8(%rsp),%rdi
   shr     %r15
   mov     %r15,%rdx
   imul    %rbp,%rdx
   lea     (%r12,%rdx,1),%r14
   mov     %r14,%rsi
   callq   *%r13
   cmp     $0x0,%eax
   je      4f                    ; cmp == 0
   jg      1b                    ; cmp > 0
   mov     %r15,%rbx             ; else move left
   test    %rbx,%rbx
   jne     2b

; return (NULL);
3: add     $0x18,%rsp
   xor     %eax,%eax
   pop     %rbx
   pop     %rbp
   pop     %r12
   pop     %r13
   pop     %r14
   pop     %r15
   ret

; return ((void *)p);
4: add     $0x18,%rsp
   mov     %r14,%rax
   pop     %rbx
   pop     %rbp
   pop     %r12
   pop     %r13
   pop     %r14
   pop     %r15
   ret
```

Listing C.4: Binary search routine (compiled assembly).

For completeness, we finally list the source code and dissassemly of the integer comparison function we used in the macrobenchmark evaluation of Section 5.5.3.

```
int int_comp(const void *p1, const void *p2)
{
    int a = *((int*) p1), b = *((int*) p2);

    if (a == b)
        return 0;
    else if (a > b)
        return 1;
    else
        return -1;
}
```

Listing C.5: Integer comparison routine.

```
int_comp:
   xor    %eax, %eax
   mov    (%rsi), %edx
   cmp    %edx, (%rdi)
   je 1f
   setg   %al
   movzbl %al,%eax
   lea    -0x1(%rax,%rax,1),%eax
1: ret
```

Listing C.6: Integer comparison routine (compiled assembly).

Appendix D

Foreshadow's cache requirements

In this appendix, we provide experimental evidence that Foreshadow requires enclaved data to be present in the L1D CPU cache. We attribute this condition to SGX's microarchitectural implementation, for previous Meltdown-type exploits targeting hierarchical kernel memory, do *not* have such strict caching requirements.[1]

Placing secrets at specific cache levels. We rely on Intel's Transactional Synchronization Extensions (TSX) to ensure that secrets only reside in the L2 and L3 cache levels, but not in L1. Particularly, we abuse that after a TSX transaction has started writes are cached in the L1 cache, without being propagated down to L2 and L3. When a transaction aborts and needs to be rolled back, all cache lines in the write set are simply marked invalid in the L1

[1]Note that more recent insights [226], developed after our original publication, indicate that both Foreshadow and Meltdown are equally restricted to leak data from the L1D cache. In the case of Meltdown, however, misspeculated execution paths in the kernel may inadvertently dereference user-space registers when handling system calls or interrupts. These Spectre gadgets in the kernel essentially act as a confused deputy that prefetches uncached secrets into L1D, at which point they can later be leaked from user space using Meltdown. Importantly, such prefetching does *not* apply to Foreshadow-SGX, as any attempt to speculatively access enclave memory from the kernel will be supressed by the processor. In the context of virtual machine isolation, however, similar speculative register dereference gadgets in hypervisors have been successfully combined with Foreshadow-VMM to leak host memory not initially present in the L1D cache [226].

```
1 void load_in_L2( uint64_t *secret ) {
2   asm volatile ( "mfence\n" );
3   if ( rtm_begin() == 0 ) {
4     *(secret) += 1;
5     rtm_abort();
6   }
7   asm volatile ( "mfence\n" );
8 }
```

Listing D.1: We evict secrets from the L1D cache by including them in the write set of an aborted TSX transaction.

cache. Future references to these addresses only hit the L2 cache, which still holds their original value.

Listing D.1 displays how we leverage this mechanism to ensure that the secret is only present in the L2 and L3 caches. At Line 3 we start a new transaction. Next the secret is modified to ensure its updated value is located in the L1 cache. When finally the transaction is aborted, the L1 cache line holding the secret is marked as invalid, but the corresponding L2/L3 cache lines remain unaffected. Execution is rolled back to Line 3 where from a programmer's perspective `rtm_begin()` returned −1 immediately. The `mfence` instructions ensure that memory accesses cannot be reordered.

Verifying cache levels. As enclave memory is exclusively accessible to the enclave, we rely on a carefully crafted benchmark enclave that places a secret at the intended cache level. Unfortunately returning execution control from the enclave (`eexit`), may inadvertently evict enclave secrets to secondary cache levels or even to main memory. To detect such events, we confirm their current cache level after every attack iteration.

Verifying at which level enclave data is currently cached is challenging. SGX's abort page semantics prevent us from directly measuring the access times of enclave data: we did not observe any timing difference between accessing cached and non-cached secrets from outside the enclave. Moving such cache verification code into the enclave, on the other hand, is infeasible as `rdtsc` instructions cannot be executed in enclave mode on SGXv1 machines [114]. We therefore resort to creating a debug benchmark enclave and measure access times of reading enclave data through the `edbgrd` instruction. As `edbgrd` may inadvertently move enclave data to caches closer to the processor, we only perform this additional verification step *after* the actual Foreshadow attack attempt.

We carefully benchmarked the access times for enclave secrets residing in L1,

Table D.1: Access times for enclave and non-enclave memory at various cache levels (median over 100,000 runs).

Cache event	Unprotected (cycles)	`edbgrd` (cycles)
L1 cache hit	40	1,400
L2 cache hit	46	1,406
Cache miss	238	1,734

L2, and main memory. Table D.1 displays the median timing results for 100,000 runs. As expected, accessing enclave secrets in the L1 cache is only slightly faster than when they need to be fetched from the second-level L2 cache. This timing difference (6 cycles) is furthermore identical to L1/L2 cache hits of non-enclave memory. When SGX memory needs to be fetched from main memory, however, it needs to be decrypted by the memory encryption engine which adds significant additional latency.

Experimental setup. As we are only interested in whether the attack variations succeed, not their bandwidth, we made some changes to our attack setting. Each attack operates in a guess/verify fashion; for every 256 possible values of the secret byte, we performed 100,000 Foreshadow rounds. Each round starts by first entering the benchmark enclave to explicitly place the secret at the desired cache level. After Foreshadow's transient-execution phase, a single oracle slot (the current guess) is reloaded to receive the output of the transient instruction sequence. Finally we verify whether the enclave secret is still located at its intended cache level by measuring `edbgrd` timing. Any attack results from inadvertently evicted enclave secrets are discarded.

Success rates. We first execute the Foreshadow-L1 attack 100,000 times against an enclave secret residing in the L1 cache. When we observe `edbgrd` timings larger than 1,405 cycles after the attack attempt, we assume the secret must have been evicted from the L1 cache and discard the result. For *every* of the remaining 96,594 attack rounds, we successfully received the secret.

We repeated the same test for enclave secrets residing in the L2 cache. This time, we discarded results with `edbgrd` timings exceeding 1,408 ticks after the attack. Out of the 98,610 remaining attack attempts, *none* succeeded in speculatively loading a secret-dependent oracle buffer slot in the transient-execution phase.

To rule out the possibility that the transient instructions may need more attempts to elevate the enclave secret from the L2 to the L1 cache, we ran the same benchmark with 1,000 repeated transient executions before actually reloading the oracle buffer. This severely reduced the number of accepted attack attempt down to 10,205. Still, *all* Foreshadow-L2 attack attempts failed.

Conclusions. As long as enclave secrets reside in the L1 cache, we observe 100% success rates. Even though L2 cache accesses only take a mere 6 cycles longer, the success rates sharply drop to zero. The Meltdown [162] attack to extract supervisor data does *not* suffer from such a hard limit, and has even been successfully applied to read kernel secrets directly from main memory. When applying Foreshadow against kernel data, we could indeed trivially extract kernel secrets from the L2 cache without noticing a significant success rate drop.

We conclude that both Meltdown and Foreshadow exploit a similar race condition vulnerability in the CPU's out-of-order pipeline behavior, but Intel SGX's abort page semantics apparently have a profound microarchitectural impact. Attack conditions are much more stringent to breach enclave than kernel isolation.

Appendix E

Additional resources for LVI attacks

This appendix first presents a universal classification tree for Load Value Injection (LVI) attacks discussed in Chapter 7. Next, we provide details on the Intel SGX quote layout, which is relevant for the case-study gadget discussed in Section 7.7.1. Finally, to provide a baseline for comparing future defenses, we report on the `lfence` counts observed for the tested prototype compiler mitigations.

E.1 LVI classification tree

In this appendix, we propose an unambiguous naming scheme to reason about and distinguish LVI variants, following the (extended) transient-execution attack classification tree by Canella et al. [36]. Particularly, in a first level, we distinguish the *fault or assist type* triggering the transient execution, and at a second level we specify the *microarchitectural buffer* which is used as the injection source. Figure E.1 shows the resulting two-level LVI classification tree. Note that, much like in the perpendicular Spectre class of attacks [36], not all CPUs from all vendors might be susceptible to all of these variants.

Applicability to Intel SGX. We remark that some of the fault types that may trigger LVI in Fig. E.1 are specific to Intel SGX's root attacker model. Particularly, LVI-US generates supervisor-mode page faults by clearing the

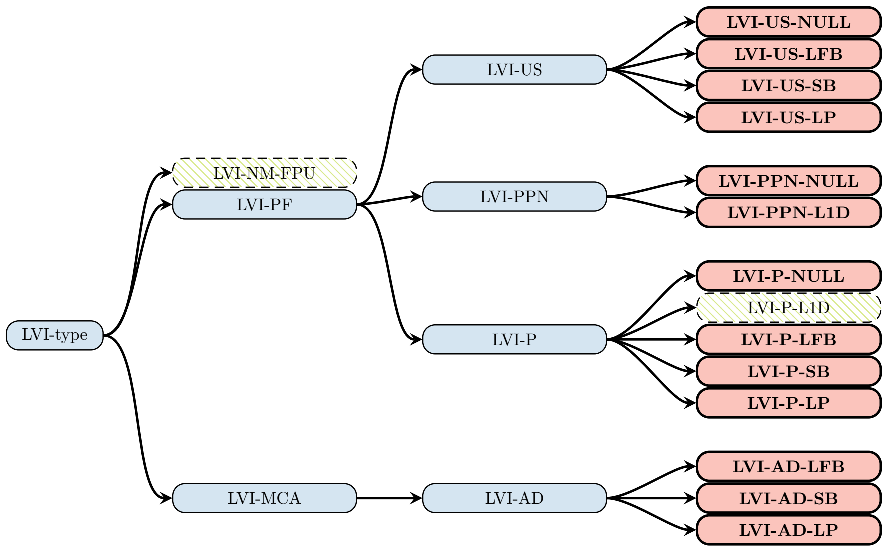


Figure E.1: Extensible LVI classification tree (generated using `https://transient.fail/`) with possible attack variants (red, bold), and neutralized variants that are already prevented by current software and microcode mitigations (green, dashed).

user-accessible bit in the untrusted page-table entry mapping a trusted enclave memory location. The user-accessible bit can only be modified by root attackers that control the untrusted OS, and hence does not apply in a user-to-kernel or user-to-user LVI scenario. Furthermore, LVI-PPN generates SGX-specific EPCM page faults by supplying a rogue physical page number in a page-table entry mapping trusted enclave memory (cf. Section 7.6.1). This variant is specific to Intel SGX's EPCM memory access control model.

Finally, as explored in Section 7.8, LVI-P and LVI-AD are not specific to Intel SGX, and might apply to traditional kernel and process isolation as well.

Neutralized variants. Interestingly, as part of our analysis, we found that some LVI variants are in principle feasible on unpatched systems, but are already properly prevented as an unintended side effect of software mitigations that have been widely deployed in response to Meltdown-type cross-domain leakage attacks.

We considered whether virtual machine or OS process Foreshadow variants [271] may also be reversely exploited through an injection-based LVI methodology, but we concluded that no additional mitigations are required. In the case of virtual machines, the untrusted kernel can only provoke non-present page faults (and hence LVI-P-L1D injection) for less-privileged applications, and never for more privileged hypervisor software. Alternatively, we find that cross-process LVI-P-L1D is possible in demand-paging scenarios when the kernel does not properly invalidate the PPN field when unmapping a victim page and assigning the underlying physical memory to another process. The next page dereference in the victim process provokes a page fault leading to the L1TF condition and causing the L1D cache to inject potentially poisoned data from the attacker process into the victim's transient data stream. However, while this attack is indeed feasible on unpatched systems, we found that it is already properly prevented by the recommended PTE inversion [46] countermeasure which has been widely deployed in all major operating systems in response to Foreshadow.

Secondly, we considered that some processors transiently compute on unauthorized values from the FPU register file before delivering a device-not-available exception (`#NM`) [234]. This may be abused in a "reverse LazyFP" LVI-NM-FPU attack to inject attacker-controlled FPU register contents into a victim application's transient data stream. However, we concluded that no additional mitigations are required for this variant as all major operating systems inhibit the `#NM` trigger completely by unconditionally applying the recommended eager FPU switching mitigation. Likewise, Intel confirmed that for every enclave (re-)entry SGX catches and signals the `#NM` exception before any enclave code can run.

Finally, we concluded that the original Meltdown [162] attack to read (cached) kernel memory from user space cannot be inverted into an LVI-L1D equivalent. The reasoning here is that the user-accessible page-table entry attribute is only enforced in privilege ring 3, and a benign victim process would never dereference kernel memory.

E.2 Intel SGX quote layout

We first provide the C data structure layout representing a quote in Listing E.1. Note that the `report_data` field in the `sgx_report_body_t;` is part of the (untrusted) input provided as part of the QE invocation. The only requirement on this data is that it needs to have a valid SGX report checksum, and hence needs to be filled in by a genuine enclave running on the target system (but this can also be for instance an attacker-controlled debug enclave).

Furthermore, Listing E.3 provides the `get_quote` entry point in Intel SGX SDK Enclave Definition Language (EDL) specification. Note that the quote data structure holding the asymmetric cryptographic signature is relatively big, and

```
typedef struct _sgx_report_data_t {
    uint8_t                 d[64];
} sgx_report_data_t;

typedef struct _report_body_t {
    ...
    /* (320) Data provided by the user */
    sgx_report_data_t       report_data;
} sgx_report_body_t;

typedef struct _quote_t {
    uint16_t            version;        /* 0   */
    uint16_t            sign_type;      /* 2   */
    sgx_epid_group_id_t epid_group_id;  /* 4   */
    sgx_isv_svn_t       qe_svn;         /* 8   */
    sgx_isv_svn_t       pce_svn;        /* 10  */
    uint32_t            xeid;           /* 12  */
    sgx_basename_t      basename;       /* 16  */
    sgx_report_body_t   report_body;    /* 48  */
    uint32_t            signature_len;  /* 432 */
    uint8_t             signature[];    /* 436 */
} sgx_quote_t;
```

Listing E.1: `https://github.com/intel/linux-sgx/blob/master/common/inc/sgx_quote.h#L87`

Table E.1: Number of `lfence`s inserted by different compiler and assembler mitigations for the OpenSSL and SPEC benchmarks (cf. Figs. 7.8 to 7.10).

Benchmark	**Unoptimized assembler (Intel)**		**Optimized compiler (Intel)**			**Unoptimized LLVM-IR (ours)**	
	`gcc-plain`	`gcc-lfence`	`clang-plain`	`clang-full`	`clang-ret`	`load+ret`	`ret-only`
`libcrypto.a`	0	73 998	0	24 710	5608	39 368	5119
`libssl.a`	0	15 034	0	5248	1615	10 228	1415
600.perlbench	0	104 475	0	32 764	2584	-	-
602.gcc	10	458 799	1	148 069	17 198	-	-
605.mcf	0	1191	0	266	44	-	-
620.omnetpp	0	78 968	0	36 940	5578	-	-
623.xalancbmk	2	252 080	0	110 353	10 750	-	-
625.x264	0	31 748	0	5582	528	-	-
631.deepsjeng	0	4315	0	545	118	-	-
641.leela	0	8997	0	1669	340	-	-
657.xz	0	7820	0	1534	419	-	-

hence is not transparently cloned into enclave memory. Instead this pointer is declared as `user_check` and explicitly verified to lie outside the enclave in the QE implementation, allowing to directly read from and write to this pointer

```
/* emp_quote: Untrusted pointer to quote output
 *            buffer outside enclave.
 * quote_body: sgx_quote_t holding quote metadata
 *             (without the actual signature).
 */
ret = qe_epid_sign(...
                   emp_quote,   /* fill in signature */
                   &quote_body, /* fill in metadata */
                   (uint32_t)sign_size);
...

/* now copy sgx_quote_t metadata (including user-
   provided report_data) into untrusted output buffer*/
memcpy(emp_quote, &quote_body, sizeof(sgx_quote_t));

/* now erase enclave secrets (EPID private key) */
CLEANUP:
  if(p_epid_context)
      epid_member_delete(&p_epid_context);
  return ret;
}
```

Listing E.2: `https://github.com/intel/linux-sgx/blob/master/psw/ae/qe/quoting_enclave.cpp#L1139`

```
public uint32_t get_quote(
  [size = blob_size, in, out] uint8_t *p_blob,
  uint32_t blob_size,
  [in] const sgx_report_t *p_report,
  sgx_quote_sign_type_t quote_type,
  [in] const sgx_spid_t *p_spid,
  [in] const sgx_quote_nonce_t *p_nonce,
  // SigRL is big, so we cannot copy it into EPC
  [user_check] const uint8_t *p_sig_rl,
  uint32_t sig_rl_size,
  [out] sgx_report_t *qe_report,
  // Quote is big, we should output it in piece meal.
  [user_check] uint8_t *p_quote,
  uint32_t quote_size, sgx_isv_svn_t pce_isvnsvn);
```

Listing E.3: `https://github.com/intel/linux-sgx/blob/master/psw/ae/qe/quoting_enclave.edl#L43`

from the trusted enclave code.

Listing E.2 finally provides the C code fragment including the `memcpy` invocation discussed in Section 7.7.1.

E.3 `lfence` counts for compiler mitigations

Table E.1 additionally provides the number of `lfence` instructions inserted by the various compiler and assembler mitigations introduced in Section 7.9.2 for the OpenSSL and SPEC2017 benchmarks.

Bibliography

[1] O. Aciiçmez and Ç. K. Koç. “Trace-Driven Cache Attacks on AES”. In: *International Conference on Information and Communications Security (ICICS)*. 2006, pp. 112–121.

[2] O. Aciiçmez, Ç. K. Koç, and J.-P. Seifert. “On the Power of Simple Branch Prediction Analysis”. In: *2nd ACM Asia Conference on Computer and Communications Security (AsiaCCS)*. 2007, pp. 312–320.

[3] O. Aciiçmez, Ç. K. Koç, and J.-P. Seifert. “Predicting Secret Keys via Branch Prediction”. In: *Cryptographers’ Track at the RSA Conference*. 2007, pp. 225–242.

[4] O. Aciiçmez, W. Schindler, and Ç. K. Koç. “Cache Based Remote Timing Attack on the AES”. In: *Cryptographers’ Track at the RSA Conference*. 2007, pp. 271–286.

[5] A. C. Aldaya and B. B. Brumley. “Microarchitecture Online Template Attacks”. In: *arXiv preprint arXiv:2007.05337* (2020).

[6] A. C. Aldaya and B. B. Brumley. “When One Vulnerable Primitive Turns Viral: Novel Single-Trace Attacks on ECDSA and RSA”. In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* (2020), pp. 196–221.

[7] A. C. Aldaya, B. B. Brumley, S. ul Hassan, C. P. García, and N. Tuveri. “Port Contention for Fun and Profit”. In: *40th IEEE Symposium on Security and Privacy (S&P)*. 2019, pp. 870–887.

[8] A. C. Aldaya, C. P. García, and B. B. Brumley. “From A to Z: Projective Coordinates Leakage in the Wild”. In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* (2020).

[9] F. Alder, N. Asokan, A. Kurnikov, A. Paverd, and M. Steiner. “S-Faas: Trustworthy and Accountable Function-as-a-Service Using Intel SGX”. In: *ACM Conference on Cloud Computing Security Workshop*. 2019, pp. 185–199.

[10] T. Alves and D. Felton. *TrustZone: Integrated Hardware and Software Security*. White Paper. 2004.

[11] AMD. *AMD SEV-SNP: Strengthening VM Isolation with Integrity Protection and More*. White Paper. Jan. 2020.

[12] AMD. *Software Techniques for Managing Speculation on AMD Processors*. White Paper. Revison 7.10.18. 2018.

[13] AMD. *Speculation Behavior in AMD Micro-Architectures*. White Paper. 2019.

[14] I. Anati, S. Gueron, S. Johnson, and V. Scarlata. "Innovative Technology for CPU Based Attestation and Sealing". In: *2nd ACM International Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*. 2013.

[15] M. Andrysco, D. Kohlbrenner, K. Mowery, R. Jhala, S. Lerner, and H. Shacham. "On Subnormal Floating Point and Abnormal Timing". In: *36th IEEE Symposium on Security and Privacy (S&P)*. 2015, pp. 623–639.

[16] ARM. *Cortex-M0 Technical Reference Manual r0p0*. 2009.

[17] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. L. Stillwell, et al. "SCONE: Secure Linux Containers with Intel SGX". In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2016, pp. 689–703.

[18] M. Balliu, M. Dam, and R. Guanciale. "InSpectre: Breaking and Fixing Microarchitectural Vulnerabilities by Formal Analysis". In: *arXiv preprint arXiv:1911.00868* (2019).

[19] A. Baumann. "Hardware is the New Software". In: *16th Workshop on Hot Topics in Operating Systems*. 2017, pp. 132–137.

[20] A. Baumann, M. Peinado, and G. Hunt. "Shielding Applications from an Untrusted Cloud with Haven". In: *11th USENIX conference on Operating Systems Design and Implementation (OSDI)*. 2014, pp. 267–283.

[21] J. Bender, M. Lesani, and J. Palsberg. "Declarative Fence Insertion". In: *ACM SIGPLAN Notices*. Vol. 50. 10. 2015, pp. 367–385.

[22] D. J. Bernstein, N. Duif, T. Lange, P. Schwabe, and B.-Y. Yang. "High-Speed High-Security Signatures". In: *Journal of Cryptographic Engineering* 2.2 (2012), pp. 77–89.

[23] A. Bhattacharyya, A. Sandulescu, M. Neugschwandtner, A. Sorniotti, B. Falsafi, M. Payer, and A. Kurmus. "SMoTherSpectre: Exploiting Speculative Execution through Port Contention". In: *26th ACM Conference on Computer and Communications Security (CCS)*. 2019.

[24] A. Biondo, M. Conti, L. Davi, T. Frassetto, and A.-R. Sadeghi. "The Guard's Dilemma: Efficient Code-Reuse Attacks Against Intel SGX". In: *27th USENIX Security Symposium.* 2018, pp. 1213–1227.

[25] T. K. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang. "Jump-Oriented Programming: A New Class of Code-Reuse Attack". In: *6th ACM Asia Conference on Computer and Communications Security (AsiaCCS).* 2011.

[26] D. D. Boggs, R. Segelken, M. Cornaby, N. Fortino, S. Chaudhry, D. Khartikov, A. Mooley, N. Tuck, and G. Vreugdenhil. *Memory Type Which is Cacheable yet Inaccessible by Speculative Instructions.* US Patent App. 16/022,274. 2019.

[27] F. Brasser, S. Capkun, A. Dmitrienko, T. Frassetto, K. Kostiainen, and A.-R. Sadeghi. "DR. SGX: Automated and Adjustable Side-Channel Protection for SGX Using Data Location Randomization". In: *35th Annual Computer Security Applications Conference (ACSAC).* 2019, pp. 788–800.

[28] F. Brasser, B. El Mahjoub, A.-R. Sadeghi, C. Wachsmann, and P. Koeberl. "TyTAN: Tiny Trust Anchor for Tiny Devices". In: *52nd ACM/IEEE Design Automation Conference (DAC).* 2015, pp. 1–6.

[29] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A.-R. Sadeghi. "Software Grand Exposure: SGX Cache Attacks are Practical". In: *11th USENIX Workshop on Offensive Technologies (WOOT).* 2017.

[30] Z. Bridges. *Speculative Execution Side Effect Suppression for Mitigating Load Value Injection.* Mar. 2020. URL: `https://lists.llvm.org/pipermail/llvm-dev/2020-March/140057.html`.

[31] D. Brumley and D. Boneh. "Remote Timing Attacks are Practical". In: *12th USENIX Security Symposium.* 2003, pp. 1–13.

[32] R. Buhren, C. Werling, and J.-P. Seifert. "Insecure Until Proven Updated: Analyzing AMD SEV's Remote Attestation". In: *26th ACM Conference on Computer and Communications Security (CCS).* 2019, pp. 1087–1099.

[33] Y. Bulygin. "CPU Side-Channels vs. Virtualization Malware: The Good, the Bad, or the Ugly". In: *ToorCon.* 2008.

[34] M. Busi, J. Noorman, J. Van Bulck, L. Galletta, P. Degano, J. T. Mühlberg, and F. Piessens. "Provably Secure Isolation for Interruptible Enclaved Execution on Small Microprocessors". In: *33rd IEEE Computer Security Foundations Symposium (CSF).* June 2020, pp. 262–276.

[35] C. Canella, D. Genkin, L. Giner, D. Gruss, M. Lipp, M. Minkin, D. Moghimi, F. Piessens, M. Schwarz, B. Sunar, J. Van Bulck, and Y. Yarom. "Fallout: Leaking Data on Meltdown-Resistant CPUs". In: *26th ACM Conference on Computer and Communications Security (CCS).* Nov. 2019, pp. 769–784.

[36] C. Canella, J. Van Bulck, M. Schwarz, M. Lipp, B. von Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss. "A Systematic Evaluation of Transient Execution Attacks and Defenses". In: *28th USENIX Security Symposium*. Aug. 2019, pp. 249–266.

[37] D. Cerdeira, N. Santos, P. Fonseca, and S. Pinto. "SoK: Understanding the Prevailing Security Vulnerabilities in TrustZone-assisted TEE Systems". In: *41st IEEE Symposium on Security and Privacy (S&P)*. 2020, pp. 18–20.

[38] S. Checkoway and H. Shacham. "Iago Attacks: Why the System Call API is a Bad Untrusted RPC Interface". In: *18th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2013, pp. 253–264.

[39] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai. "SgxPectre Attacks: Stealing Intel Secrets from SGX Enclaves via Speculative Execution". In: *4th IEEE European Symposium on Security and Privacy (Euro S&P)*. 2019.

[40] G. Chen, W. Wang, T. Chen, S. Chen, Y. Zhang, X. Wang, T.-H. Lai, and D. Lin. "Racing in Hyperspace: Closing Hyper-Threading Side Channels on SGX with Contrived Data Races". In: *39th IEEE Symposium on Security and Privacy (S&P)*. 2018.

[41] H. Chen, Y. Mao, X. Wang, D. Zhou, N. Zeldovich, and M. F. Kaashoek. "Linux Kernel Vulnerabilities: State-of-the-Art Defenses and Open Problems". In: *2nd Asia-Pacific Workshop on Systems*. ACM, 2011, 5:1–5:5.

[42] S. Chen, X. Zhang, M. K. Reiter, and Y. Zhang. "Detecting Privileged Side-Channel Attacks in Shielded Execution with Déjà Vu". In: *12th ACM Asia Conference on Computer and Communications Security (AsiaCCS)*. 2017, pp. 7–18.

[43] J. V. Cleemput, B. Coppens, and B. De Sutter. "Compiler Mitigations for Time Attacks on Modern x86 Processors". In: *ACM Transactions on Architecture and Code Optimization* 8.4 (2012), p. 23.

[44] B. Coppens, I. Verbauwhede, K. De Bosschere, and B. De Sutter. "Practical Mitigations for Timing-Based Side-Channel Attacks on Modern x86 Processors". In: *30th IEEE Symposium on Security and Privacy (S&P)*. 2009, pp. 45–60.

[45] J. Corbet. *Structure Holes and Information Leaks*. Dec. 2010. URL: `https://lwn.net/Articles/417989/`.

[46] J. Corbet. *Meltdown Strikes Back: The L1 Terminal Fault Vulnerability*. Aug. 2018. URL: `https://lwn.net/Articles/762570/`.

[47] V. Costan and S. Devadas. “Intel SGX Explained.” In: *IACR Cryptology ePrint Archive* 2016.086 (2016), pp. 1–118.

[48] V. Costan, I. Lebedev, and S. Devadas. “Sanctum: Minimal Hardware Extensions for Strong Software Isolation”. In: *25th USENIX Security Symposium*. 2016, pp. 857–874.

[49] S. Cuyt. “A Security Analysis of Interrupts in Embedded Enclaved Execution”. MA thesis. KU Leuven, 2019.

[50] F. Dall, G. De Micheli, T. Eisenbarth, D. Genkin, N. Heninger, A. Moghimi, and Y. Yarom. “CacheQuote: Efficiently Recovering Long-Term Secrets of SGX EPID via Cache Attacks”. In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2 (2018), pp. 171–191.

[51] R. De Clercq, F. Piessens, D. Schellekens, and I. Verbauwhede. “Secure Interrupts on Low-End Microcontrollers”. In: *25th IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP)*. 2014, pp. 147–152.

[52] F. Dewald, H. Mantel, and A. Weber. “AVR Processors as a Platform for Language-Based Security”. In: *European Symposium on Research in Computer Security (ESORICS)*. 2017, pp. 427–445.

[53] S. Dinesh, N. Burow, D. Xu, and M. Payer. “RetroWrite: Statically Instrumenting COTS Binaries for Fuzzing and Sanitization”. In: *41st IEEE Symposium on Security and Privacy (S&P)*. 2020.

[54] C. Disselkoen, D. Kohlbrenner, L. Porter, and D. Tullsen. “Prime+Abort: A Timer-Free High-Precision L3 Cache Attack Using Intel TSX”. In: *26th USENIX Security Symposium*. 2017, pp. 51–67.

[55] J. Edge. *GCC 4.3.0 Exposes a Kernel Bug*. Mar. 2008. URL: `https://lwn.net/Articles/272048/`.

[56] K. Eldefrawy, G. Tsudik, A. Francillon, and D. Perito. “SMART: Secure and Minimal Architecture for (Establishing a Dynamic) Root of Trust.” In: *19th Annual Network and Distributed System Security Symposium (NDSS)*. 2012, pp. 1–15.

[57] K. Elphinstone and G. Heiser. “From L3 to seL4 What Have We Learnt in 20 Years of L4 Microkernels?” In: *24th ACM Symposium on Operating Systems Principles (SOSP)*. 2013, pp. 133–150.

[58] D. Evtyushkin, J. Elwell, M. Ozsoy, D. Ponomarev, N. A. Ghazaleh, and R. Riley. “Iso-x: A Flexible Architecture for Hardware-Managed Isolated Execution”. In: *47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2014, pp. 190–202.

[59] D. Evtyushkin, R. Riley, N. C. Abu-Ghazaleh, D. Ponomarev, et al. “BranchScope: A New Side-Channel Attack on Directional Branch Predictor”. In: *23rd ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2018, pp. 693–707.

[60] R. S. Fabry. “Capability-Based Addressing”. In: *Communications of the ACM* 17.7 (1974), pp. 403–412.

[61] B. Falk. *CPU Introspection: Intel Load Port Snooping*. Dec. 2019. URL: `https://gamozolabs.github.io/metrology/2019/12/30/load-port-monitor`.

[62] A. Ferraiuolo, A. Baumann, C. Hawblitzel, and B. Parno. “Komodo: Using Verification to Disentangle Secure-Enclave Hardware from Software”. In: *26th ACM Symposium on Operating Systems Principles (SOSP)*. 2017, pp. 287–305.

[63] A. Ferraiuolo, Y. Wang, R. Xu, D. Zhang, A. Myers, and E. Suh. *Full-Processor Timing Channel Protection with Applications to Secure Hardware Compartments*. Computing and Information Science Technical Report. Cornell University, Nov. 2015.

[64] A. Ferraiuolo, R. Xu, D. Zhang, A. C. Myers, and G. E. Suh. “Verification of a Practical Hardware Security Architecture through Static Information Flow Analysis”. In: *22th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2017, pp. 555–568.

[65] A. Fog. *Calling Conventions for Different C++ Compilers and Operating Systems*. 2018.

[66] A. Fog. *Instruction Tables. Lists of Instruction Latencies, Throughputs and Micro-Operation Breakdowns for Intel, AMD and VIA CPUs*. 2018.

[67] A. Fogh. *Negative Result: Reading Kernel Memory from User Mode*. July 2017. URL: `https://cyber.wtf/2017/07/28/`.

[68] Fortanix. *Fortanix Enclave Development Platform – Rust EDP*. 2019. URL: `https://edp.fortanix.com/`.

[69] Y. Fu, E. Bauman, R. Quinonez, and Z. Lin. “SGX-LAPD: Thwarting Controlled Side Channel Attacks via Enclave Verifiable Page Faults”. In: *20th International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*. 2017, pp. 357–380.

[70] Q. Ge, Y. Yarom, T. Chothia, and G. Heiser. “Time Protection: The Missing OS Abstraction”. In: *14th European Conference on Computer Systems (EuroSys)*. ACM, 2019, pp. 1–17.

[71] Q. Ge, Y. Yarom, D. Cock, and G. Heiser. “A Survey of Microarchitectural Timing Attacks and Countermeasures on Contemporary Hardware”. In: *Journal of Cryptographic Engineering* 8.1 (2018), pp. 1–27.

[72] N. van Ginkel, R. Strackx, J. T. Mühlberg, and F. Piessens. “Towards Safe Enclaves”. In: *Hot Issues in Security Principles and Trust (HotSpot).* 2016, pp. 1–16.

[73] N. van Ginkel, R. Strackx, and F. Piessens. “Automatically Generating Secure Wrappers for SGX Enclaves from Separation Logic Specifications”. In: *15th Asian Symposium on Programming Languages and Systems (APLAS).* 2017, pp. 105–123.

[74] A. Glew, G. Hinton, and H. Akkary. *Method and Apparatus for Performing Page Table Walks in a Microprocessor Capable of Processing Speculative Instructions.* US Patent 5,680,565. 1997.

[75] J. D. Golić and C. Tymen. “Multiplicative Masking and Power Analysis of AES”. In: *5th International Conference on Cryptographic Hardware and Embedded Systems (CHES).* 2003, pp. 198–212.

[76] T. Goodspeed. “Practical Attacks Against the MSP430 BSL”. In: *25th Chaos Communications Congress.* 2008.

[77] Google. *Asylo: An open and Flexible Framework for Enclave Applications.* 2019. URL: `https://asylo.dev/`.

[78] J. Götzfried, T. Müller, R. De Clercq, P. Maene, F. Freiling, and I. Verbauwhede. “Soteria: Offline Software Protection within Low-Cost Embedded Devices”. In: *31st Annual Computer Security Applications Conference (ACSAC).* 2015, pp. 241–250.

[79] J. Götzfried, M. Eckert, S. Schinzel, and T. Müller. “Cache Attacks on Intel SGX”. In: *10th European Workshop on Systems Security (EuroSec).* 2017.

[80] Graphene Project. *Graphene: A Library OS for Unmodified Applications.* 2019. URL: `https://grapheneproject.io/`.

[81] B. Gras, K. Razavi, E. Bosman, H. Bos, and C. Giuffrida. “ASLR on the Line: Practical Cache Attacks on the MMU”. In: *24th Annual Network and Distributed System Security Symposium (NDSS).* 2017.

[82] D. Gruss, D. Hansen, and B. Gregg. “Kernel Isolation: From an Academic Idea to an Efficient Patch for Every Computer”. In: *;login: the USENIX magazine* 43.4 (2018), pp. 10–14.

[83] D. Gruss, E. Kraft, T. Tiwari, M. Schwarz, A. Trachtenberg, J. Hennessey, A. Ionescu, and A. Fogh. “Page Cache Attacks”. In: *26th ACM Conference on Computer and Communications Security (CCS).* 2019.

[84] D. Gruss, J. Lettner, F. Schuster, O. Ohrimenko, I. Haller, and M. Costa. "Strong and Efficient Cache Side-Channel Protection Using Hardware Transactional Memory". In: *26th USENIX Security Symposium.* 2017.

[85] D. Gruss, M. Lipp, M. Schwarz, R. Fellner, C. Maurice, and S. Mangard. "KASLR is Dead: Long Live KASLR". In: *International Symposium on Engineering Secure Software and Systems (ESSoS).* 2017, pp. 161–176.

[86] D. Gruss, C. Maurice, A. Fogh, M. Lipp, and S. Mangard. "Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR". In: *23rd ACM Conference on Computer and Communications Security (CCS).* 2016, pp. 368–379.

[87] D. Gruss, C. Maurice, K. Wagner, and S. Mangard. "Flush+Flush: A Fast and Stealthy Cache Attack". In: *13th Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA).* 2016.

[88] D. Gruss, R. Spreitzer, and S. Mangard. "Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches." In: *24nd USENIX Security Symposium.* 2015, pp. 897–912.

[89] S. Gueron. "A Memory Encryption Engine Suitable for General Purpose Processors." In: *IACR Cryptology ePrint Archive* (2016).

[90] S. Gueron. *Intel Advanced Encryption Standard (Intel AES) Instructions Set – Rev 3.01.* 2012.

[91] J. Gyselinck, J. Van Bulck, F. Piessens, and R. Strackx. "Off-limits: Abusing Legacy x86 Memory Segmentation to Spy on Enclaved Execution". In: *International Symposium on Engineering Secure Software and Systems (ESSoS).* June 2018, pp. 44–60.

[92] M. Hähnel, W. Cui, and M. Peinado. "High-Resolution Side Channels for Untrusted Operating Systems". In: *USENIX Annual Technical Conference (ATC).* 2017.

[93] N. Hardy. "The Confused Deputy (or Why Capabilities Might Have Been Invented)". In: *ACM SIGOPS Operating Systems Review* 22.4 (1988), pp. 36–38.

[94] S. ul Hassan, I. Gridin, I. M. Delgado-Lozano, C. P. García, J.-J. Chi-Domínguez, A. C. Aldaya, and B. B. Brumley. "Déjà Vu: Side-Channel Analysis of Mozilla's NSS". In: *arXiv preprint arXiv:2008.06004* (2020).

[95] W. He, W. Zhang, S. Das, and Y. Liu. "Sgxlinger: A New Side-Channel Attack Vector Based on Interrupt Latency Against Enclave Execution". In: *36th IEEE International Conference on Computer Design (ICCD).* 2018, pp. 108–114.

[96] J. L. Hennessy and D. A. Patterson. “A New Golden Age for Computer Architecture”. In: *Communications of the ACM* 62.2 (Jan. 2019), pp. 48–60.

[97] F. Hetzelt and R. Buhren. “Security Analysis of Encrypted Virtual Machines”. In: *ACM SIGPLAN Notices* 52.7 (2017), pp. 129–142.

[98] M. D. Hill, J. Masters, P. Ranganathan, P. Turner, and J. L. Hennessy. “On the Spectre and Meltdown Processor Security Vulnerabilities”. In: *IEEE Micro* 39.2 (2019), pp. 9–19.

[99] M. Hoekstra, R. Lal, P. Pappachan, V. Phegade, and J. Del Cuvillo. “Using Innovative Instructions to Create Trustworthy Software Solutions.” In: *2nd ACM International Workshop on Hardware and Architectural Support for Security and Privacy (HASP).* 2013.

[100] O. S. Hofmann, S. Kim, A. M. Dunn, M. Z. Lee, and E. Witchel. “InkTag: Secure Applications on an Untrusted Operating System”. In: *18th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS).* 2013, pp. 265–278.

[101] J. Horn. *Reading Privileged Memory with a Side-Channel.* Jan. 2018. URL: `https://googleprojectzero.blogspot.com/2018/01/`.

[102] J. Horn. *Speculative Execution, Variant 4: Speculative Store Bypass.* 2018. URL: `https://bugs.chromium.org/p/project-zero/issues/detail?id=1528`.

[103] S. Hosseinzadeh, H. Liljestrand, V. Leppänen, and A. Paverd. “Mitigating Branch-Shadowing Attacks on Intel SGX Using Control Flow Randomization”. In: *3rd Workshop on System Software for Trusted Execution (SysTEX).* 2018, pp. 42–47.

[104] R. Hund, C. Willems, and T. Holz. “Practical Timing Side Channel Attacks Against Kernel Space ASLR”. In: *34th IEEE Symposium on Security and Privacy (S&P).* 2013, pp. 191–205.

[105] T. Huo, X. Meng, W. Wang, C. Hao, P. Zhao, J. Zhai, and M. Li. “Bluethunder: A 2-level Directional Predictor Based Side-Channel Attack Against SGX”. In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* (2020), pp. 321–347.

[106] Intel. *An Optimized Mitigation Approach for Load Value Injection.* Mar. 2020. URL: `https://intel.ly/2CSsHwp`.

[107] Intel. *Deep Dive: Intel Analysis of L1 Terminal Fault.* Aug. 2018. URL: `https://intel.ly/3j81dlT`.

[108] Intel. *Deep Dive: Intel Analysis of Microarchitectural Data Sampling.* May 2019. URL: `https://intel.ly/3latfih`.

[109] Intel. *Deep Dive: Intel Transactional Synchronization Extensions Asynchronous Abort.* Nov. 2019. URL: `https://intel.ly/3gkXwav`.

[110] Intel. *Deep Dive: Load Value Injection.* Mar. 2020. URL: `https://intel.ly/3gnAYGj`.

[111] Intel. *Hyperledger Sawtooth.* URL: `https://sawtooth.hyperledger.org/docs/core/releases/latest/introduction.html`.

[112] Intel. *Improving Real-Time Performance by Utilizing Cache Allocation Technology.* White Paper. Apr. 2015.

[113] Intel. *Intel 64 and IA-32 Architectures Optimization Reference Manual.* Order no. 248966-040. 2020.

[114] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual – Combined volumes.* Reference no. 325462-062US. 2020.

[115] Intel. *Intel Analysis of Speculative Execution Side Channels.* White Paper. Reference no. 336983-001. Jan. 2018.

[116] Intel. *Intel Software Guard Extensions – Get Started with the SDK.* 2019. URL: `https://software.intel.com/en-us/sgx/sdk`.

[117] Intel. *Intel Software Guard Extensions (SGX) SW Development Guidance for Potential Bounds Check Bypass Side Channel Exploits.* Revision 1.3. July 2018.

[118] Intel. *Intel Software Guard Extensions (SGX) SW Development Guidance for Potential edger8r Generated Code Side Channel Exploits.* White Paper. Revision 1.0. Mar. 2018.

[119] Intel. *Intel Software Guard Extensions Developer Guide: Protection from Side-Channel Attacks.* June 2017.

[120] Intel. *Intel Software Guard Extensions SDK for Linux OS: Developer Reference.* Nov. 2017.

[121] Intel. *Intel Software Guard Extensions SSL.* 2019. URL: `https://github.com/intel/intel-sgx-ssl`.

[122] Intel. *Intel Software Guard Extensions Trusted Computing Base Recovery.* White Paper. 2018.

[123] Intel. *Intel-SA-00219 SGX SW Developer Guidance.* White Paper. Revision 1.0. Nov. 2019.

[124] Intel. *L1 Terminal Fault Software Guidance.* Aug. 2018. URL: `https://intel.ly/2YoSOCr`.

[125] Intel. *Refined Speculative Execution Terminology.* Mar. 2020. URL: `https://intel.ly/3j15iYW`.

[126] Intel. *Retpoline: A Branch Target Injection Mitigation.* White Paper. Reference no. 337131-001. Feb. 2018.

[127] Intel. *SA-00329: L1D Eviction Sampling.* Jan. 2020. URL: `https://intel.ly/34ni28l`.

[128] Intel. *Speculative Execution Side Channel Mitigations.* White Paper. Reference no. 336996-002. May 2018.

[129] Intel. *Technical Note: Intel SGX Attestation Technical Details.* White Paper. Revision 1.0. Nov. 2019.

[130] S. Islam, A. Moghimi, I. Bruhns, M. Krebbel, B. Gulmezoglu, T. Eisenbarth, and B. Sunar. "SPOILER: Speculative Load Hazards Boost Rowhammer and Cache Attacks". In: *28th USENIX Security Symposium.* 2019.

[131] B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens. "VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java". In: *NASA Formal Methods Symposium.* 2011, pp. 41–55.

[132] Y. Jang, S. Lee, and T. Kim. "Breaking Kernel Address Space Layout Randomization with Intel TSX". In: *23rd ACM Conference on Computer and Communications Security (CCS).* 2016, pp. 380–392.

[133] S. Johnson. *Intel SGX and Side-Channels.* Mar. 2017. URL: `https://software.intel.com/articles/intel-sgx-and-side-channels`.

[134] S. Johnson, V. Scarlata, C. Rozas, E. Brickell, and F. Mckeen. *Intel Software Guard Extensions: EPID Provisioning and Attestation Services.* White Paper. 2016.

[135] D. Kaplan. *Protecting VM Register State with SEV-ES.* White Paper. Feb. 2017.

[136] D. Kaplan, J. Powell, and T. Woller. *AMD Memory Encryption.* White Paper. 2016.

[137] Z. Kenjar, T. Frassetto, D. Gens, M. Franz, and A.-R. Sadeghi. "V0LTpwn: Attacking x86 Processor Integrity from Software". In: *29th USENIX Security Symposium.* Aug. 2020.

[138] K. N. Khasawneh, E. M. Koruyeh, C. Song, D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh. "Safespec: Banishing the Spectre of a Meltdown with Leakage-Free Speculation". In: *56th ACM/IEEE Design Automation Conference (DAC).* 2019, pp. 1–6.

[139] D. Kim, D. Jang, M. Park, Y. Jeong, J. Kim, S. Choi, and B. B. Kang. "SGX-LEGO: Fine-grained SGX Controlled-Channel Attack and its Countermeasure". In: *Computers & Security* 82 (2019), pp. 118–139.

[140] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu. "Flipping Bits in Memory without Accessing Them: An Experimental Study of DRAM Disturbance Errors". In: *41st International Symposium on Computer Architecture (ISCA)*. 2014.

[141] V. Kiriansky and C. Waldspurger. "Speculative Buffer Overflows: Attacks and Defenses". In: *arXiv preprint arXiv:1807.03757* (2018).

[142] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, et al. "seL4: Formal Verification of an OS Kernel". In: *22nd ACM Symposium on Operating Systems Principles (SOSP)*. 2009, pp. 207–220.

[143] W. Koch and M. Schulte. *The Libgcrypt Reference Manual.* Version 1.7.4. Dec. 2016.

[144] P. Kocher. *Spectre Mitigations in Microsoft's C/C++ Compiler.* 2018. URL: `https://www.paulkocher.com/doc/MicrosoftCompilerSpectreMitigation.html`.

[145] P. C. Kocher. "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems". In: *Annual International Cryptology Conference.* 1996, pp. 104–113.

[146] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. "Spectre Attacks: Exploiting Speculative Execution". In: *40th IEEE Symposium on Security and Privacy (S&P)*. 2019.

[147] P. Koeberl, S. Schulz, A.-R. Sadeghi, and V. Varadharajan. "TrustLite: A Security Architecture for Tiny Embedded Devices". In: *9th European Conference on Computer Systems (EuroSys)*. ACM, 2014, 10:1–10:14.

[148] E. M. Koruyeh, K. N. Khasawneh, C. Song, and N. Abu-Ghazaleh. "Spectre Returns! Speculation Attacks Using the Return Stack Buffer". In: *12th USENIX Workshop on Offensive Technologies (WOOT)*. 2018.

[149] B. Krzanich. *Intel (INTC) CEO Brian Krzanich on Q4 2017 results.* Jan. 2018. URL: `https://seekingalpha.com/article/4140338`.

[150] M. Larabel. *LLVM Lands Performance-Hitting Mitigation For Intel LVI Vulnerability.* Apr. 2020. URL: `https://www.phoronix.com/scan.php?page=article&item=llvm-intel-lvi`.

[151] M. Larabel. *The Brutal Performance Impact From Mitigating The LVI Vulnerability.* Mar. 2020. URL: `https://www.phoronix.com/scan.php?page=article&item=lvi-attack-perf`.

[152] D. Lee, D. Jung, I. T. Fang, C.-C. Tsai, and R. A. Popa. "An Off-Chip Attack on Hardware Enclaves via the Memory Bus". In: *29th USENIX Security Symposium.* Aug. 2020.

[153] D. Lee, D. Kohlbrenner, S. Shinde, K. Asanović, and D. Song. “Keystone: An Open Framework for Architecting Trusted Execution Environments”. In: *15th European Conference on Computer Systems (EuroSys)*. 2020, pp. 1–16.

[154] J. Lee, J. Jang, Y. Jang, N. Kwak, Y. Choi, C. Choi, T. Kim, M. Peinado, and B. B. Kang. “Hacking in Darkness: Return-Oriented Programming Against Secure Enclaves”. In: *26th USENIX Security Symposium*. 2017, pp. 523–539.

[155] S. Lee and T. Kim. “Leaking Uninitialized Secure Enclave Memory via Structure Padding”. In: *arXiv preprint arXiv:1710.09061* (2017).

[156] S. Lee, M.-W. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado. “Inferring Fine-Grained Control Flow Inside SGX Enclaves with Branch Shadowing”. In: *26th USENIX Security Symposium*. 2017, pp. 557–574.

[157] G. Lehel and N. Matsakis. *RFC #560: Integer Overflows in Rust*. 2017. URL: `https://github.com/rust-lang/rfcs/pull/560`.

[158] A. Leiserson. *Side Channels and Runtime Encryption Solutions with Intel SGX*. White Paper. 2018.

[159] J. Liedtke. “On Micro-Kernel Construction”. In: *15th ACM Symposium on Operating Systems Principles (SOSP)*. 1995, pp. 237–250.

[160] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen. “Value Locality and Load Value Prediction”. In: *ACM SIGPLAN Notices* 31.9 (1996), pp. 138–147.

[161] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard. “ARMageddon: Cache Attacks on Mobile Devices”. In: *25th USENIX Security Symposium*. 2016, pp. 549–564.

[162] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. “Meltdown: Reading Kernel Memory from User Space”. In: *27th USENIX Security Symposium*. 2018.

[163] F. Liu, B. Xing, M. Steiner, M. Vij, C. Rozas, F. McKeen, M. Ozsoy, M. Fernandez, K. Zmudzinski, M. Shanahan, et al. *Processor Instruction Support to Defeat Side-Channel Attacks*. US Patent App. 16/024,733. Jan. 2020.

[164] LLVM. *The LLVM Compiler Infrastructure*. 2019. URL: `https://llvm.org`.

[165] A. Machiry, E. Gustafson, C. Spensky, C. Salls, N. Stephens, R. Wang, A. Bianchi, Y. R. Choe, C. Kruegel, and G. Vigna. “Boomerang: Exploiting the Semantic Gap in Trusted Execution Environments”. In: *24th Annual Network and Distributed System Security Symposium (NDSS)*. 2017.

[166] P. Maene, J. Götzfried, R. de Clercq, T. Müller, F. Freiling, and I. Verbauwhede. “Hardware-Based Trusted Computing Architectures for Isolation and Attestation”. In: *IEEE Transactions on Computers* 67.3 (2017), pp. 361–374.

[167] G. Maisuradze and C. Rossow. “ret2spec: Speculative Execution Using Return Stack Buffers”. In: *25th ACM Conference on Computer and Communications Security (CCS)*. 2018.

[168] G. Maisuradze and C. Rossow. “Speculose: Analyzing the Security Implications of Speculative Execution in CPUs”. In: *arXiv preprint arXiv:1801.04084* (2018).

[169] M. Marlinspike. *Technology Preview: Private Contact Discovery for Signal*. Sept. 2017. URL: `https://signal.org/blog/private-contact-discovery/`.

[170] J. Masters. *Thoughts on NetSpectre*. 2018. URL: `https://www.redhat.com/en/blog/thoughts-netspectre`.

[171] C. Maurice, M. Weber, M. Schwarz, L. Giner, D. Gruss, C. A. Boano, S. Mangard, and K. Römer. “Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud.” In: *24th Annual Network and Distributed System Security Symposium (NDSS)*. 2017, pp. 8–11.

[172] D. McCandless. *Codebases: Millions of lines of code*. 2015. URL: `https://informationisbeautiful.net/visualizations/million-lines-of-code/`.

[173] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. D. Gligor, and A. Perrig. “TrustVisor: Efficient TCB Reduction and Attestation”. In: *31st IEEE Symposium on Security and Privacy (S&P)*. 2010, pp. 143–158.

[174] J. M. McCune, B. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. “Flicker: An Execution Infrastructure for TCB Minimization”. In: *3rd European Conference on Computer Systems (EuroSys)*. ACM, 2008, pp. 315–328.

[175] F. McKeen, B. Xing, K. Zmudzinski, C. Rozas, and M. Vij. *Mechanism to Prevent Software Side Channels*. US Patent App. 15/897,406. Aug. 2019.

[176] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar. “Innovative Instructions and Software Model for Isolated Execution”. In: *2nd ACM International Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*. 2013.

[177] Microsoft. *Open Enclave SDK*. 2019. URL: `https://openenclave.io/`.

[178] M. Mitchell. *Implementing an Ultralow-Power Keypad Interface with the MSP430*. Tech. rep. Texas Instruments, Feb. 2002.

[179] MobileCoin. *MobileCoin: A Crypto-Currency Delivering Best User Experience in Blockchain World*. White Paper. Nov. 2017.

[180] A. Moghimi, T. Eisenbarth, and B. Sunar. “MemJam: A False Dependency Attack Against Constant-Time Crypto Implementations in SGX”. In: *Cryptographers’ Track at the RSA Conference*. 2018, pp. 21–44.

[181] A. Moghimi, G. Irazoqui, and T. Eisenbarth. “CacheZoom: How SGX Amplifies the Power of Cache Attacks”. In: *19th International Conference on Cryptographic Hardware and Embedded Systems (CHES)*. 2017.

[182] D. Moghimi, J. Van Bulck, N. Heninger, F. Piessens, and B. Sunar. “CopyCat: Controlled Instruction-Level Attacks on Enclaves”. In: *29th USENIX Security Symposium*. Aug. 2020, pp. 469–486.

[183] M. Morbitzer, M. Huber, J. Horsch, and S. Wessel. “SEVered: Subverting AMD’s Virtual Machine Encryption”. In: *11th European Workshop on Systems Security (EuroSec)*. 2018, pp. 1–6.

[184] J. T. Mühlberg, S. Cleemput, A. M. Mustafa, J. Van Bulck, B. Preneel, and F. Piessens. “Implementation of a High Assurance Smart Meter using Protected Module Architectures”. In: *10th WISTP International Conference on Information Security Theory and Practice (WISTP)*. Aug. 2016, pp. 53–69.

[185] J. T. Mühlberg and J. Van Bulck. “Reflections on Post-Meltdown Trusted Computing: A Case for Open Security Processors”. In: *;login: the USENIX magazine* 43.3 (2018), pp. 6–9.

[186] J. T. Mühlberg and J. Van Bulck. “Tutorial: Building Distributed Enclave Applications with Sancus and SGX”. In: *48th International Conference on Dependable Systems and Networks (DSN)*. June 2018.

[187] K. Murdock, D. Oswald, F. D. Garcia, J. Van Bulck, D. Gruss, and F. Piessens. “Plundervolt: How a Little Bit of Undervolting Can Create a Lot of Trouble”. In: *IEEE Security & Privacy Magazine Special Issue on Hardware-Assisted Security* (2020).

[188] K. Murdock, D. Oswald, F. D. Garcia, J. Van Bulck, D. Gruss, and F. Piessens. “Plundervolt: Software-Based Fault Injection Attacks Against Intel SGX”. In: *41st IEEE Symposium on Security and Privacy (S&P)*. May 2020, pp. 1466–1482.

[189] J. Noorman, J. Van Bulck, J. T. Mühlberg, F. Piessens, P. Maene, B. Preneel, I. Verbauwhede, J. Götzfried, T. Müller, and F. Freiling. “Sancus 2.0: A Low-Cost Security Architecture for IoT Devices”. In: *ACM Transactions on Privacy and Security* 20.3 (2017), pp. 1–33.

[190] J. Noorman. “Sancus: A Low-Cost Security Architecture for Distributed IoT Applications on a Shared Infrastructure”. PhD thesis. KU Leuven, 2017.

[191] J. Noorman, P. Agten, W. Daniels, R. Strackx, A. Van Herrewege, C. Huygens, B. Preneel, I. Verbauwhede, and F. Piessens. "Sancus: Low-Cost Trustworthy Extensible Networked Devices with a Zero-Software Trusted Computing Base". In: *22nd USENIX Security Symposium*. 2013, pp. 479–494.

[192] J. Noorman, J. T. Mühlberg, and F. Piessens. "Authentic Execution of Distributed Event-Driven Applications with a Small TCB". In: *13th International Workshop on Security and Trust Management (STM)*. 2017, pp. 55–71.

[193] G. Noubir and A. Sanatinia. "Trusted Code Execution on Untrusted Platform Using Intel SGX". In: *Virus Bulletin* (2016).

[194] I. D. O. Nunes, K. Eldefrawy, N. Rattanavipanon, M. Steiner, and G. Tsudik. "VRASED: A Verified Hardware/Software Co-Design for Remote Attestation". In: *28th USENIX Security Symposium*. 2019, pp. 1429–1446.

[195] S. Nürnberger and C. Rossow. "– vatiCAN – Vetted, Authenticated CAN Bus". In: *18th International Conference on Cryptographic Hardware and Embedded Systems (CHES)*. 2016, pp. 106–124.

[196] S. Nürnberger and C. Rossow. *VatiCAN Library Documentation*. Online, last consulted March 16, 2020. URL: `http://automotive-security.net/vatican/doc/`.

[197] D. O'Keeffe, D. Muthukumaran, P.-L. Aublin, F. Kelbert, C. Priebe, J. Lind, H. Zhu, and P. Pietzuch. *Spectre Attack Against SGX Enclave*. 2018. URL: `https://github.com/lsds/spectre-attack-sgx`.

[198] O. Oleksenko, B. Trach, R. Krahn, M. Silberstein, and C. Fetzer. "Varys: Protecting SGX Enclaves from Practical Side-Channel Attacks". In: *USENIX Annual Technical Conference (ATC)*. 2018, pp. 227–240.

[199] OpenSSL. *OpenSSL: The Open Source Toolkit for SSL/TLS*. 2019. URL: `http://www.openssl.org`.

[200] M. Orenbach, A. Baumann, and M. Silberstein. "Autarky: Closing Controlled Channels with Self-Paging Enclaves". In: *15th European Conference on Computer Systems (EuroSys)*. ACM, 2020, pp. 1–16.

[201] L. Orosa, R. Azevedo, and O. Mutlu. "AVPP: Address-First Value-Next Predictor with Value Prefetching for Improving the Efficiency of Load Value Prediction". In: *ACM Transactions on Architecture and Code Optimization* 15.4 (2018), p. 49.

[202] D. A. Osvik, A. Shamir, and E. Tromer. "Cache Attacks and Countermeasures: The Case of AES". In: *Cryptographers' Track at the RSA Conference*. 2006, pp. 1–20.

[203] M. Patrignani, P. Agten, R. Strackx, B. Jacobs, D. Clarke, and F. Piessens. "Secure Compilation to Protected Module Architectures". In: *ACM Transactions on Programming Languages and Systems* 37.2 (2015).

[204] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard. "DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks". In: *25th USENIX Security Symposium.* 2016, pp. 565–581.

[205] S. Pinto and N. Santos. "Demystifying ARM TrustZone: A Comprehensive Survey". In: *ACM Computing Surveys* 51.6 (2019), pp. 1–36.

[206] S. Pouyanrad, J. T. Mühlberg, and W. Joosen. "SCFMSP: Static Detection of Side Channels in MSP430 Programs". In: *15th International Conference on Availability, Reliability and Security (ARES).* 2020, pp. 1–10.

[207] C. Priebe, D. Muthukumaran, J. Lind, H. Zhu, S. Cui, V. A. Sartakov, and P. Pietzuch. "SGX-LKL: Securing the Host OS Interface for Trusted Execution". In: *arXiv preprint arXiv:1908.11143* (2019).

[208] I. Puddu, M. Schneider, M. Haller, and S. Čapkun. "Frontal Attack: Leaking Control-Flow in SGX via the CPU Frontend". In: *arXiv preprint arXiv:2005.11516* (2020).

[209] P. Puschner, R. Kirner, B. Huber, and D. Prokesch. "Compiling for Time Predictability". In: *International Conference on Computer Safety, Reliability, and Security.* 2012, pp. 382–391.

[210] P. Qiu, D. Wang, Y. Lyu, and G. Qu. "VoltJockey: Breaking SGX by Software-Controlled Voltage-Induced Hardware Faults". In: *IEEE Asian Hardware Oriented Security and Trust Symposium (AsianHOST).* 2019, pp. 1–6.

[211] H. Ragab, A. Milburn, K. Razavi, H. Bos, and C. Giuffrida. "CrossTalk: Speculative Data Leaks Across Cores Are Real". In: *42nd IEEE Symposium on Security and Privacy (S&P).* May 2021.

[212] C. Reis, A. Moshchuk, and N. Oskov. "Site Isolation: Process Separation for Web Sites Within the Browser". In: *28th USENIX Security Symposium.* 2019, pp. 1661–1678.

[213] M. Russinovich. *Pushing the Limits of Windows: Paged and Nonpaged Pool.* Oct. 2009. URL: `https://docs.microsoft.com/en-us/archive/blogs/markrussinovich/pushing-the-limits-of-windows-paged-and-nonpaged-pool`.

[214] K. Ryan. "Hardware-Backed Heist: Extracting ECDSA Keys from Qualcomm's TrustZone". In: *26th ACM Conference on Computer and Communications Security (CCS).* 2019, pp. 181–194.

[215] S. van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida. *Addendum to RIDL: Rogue In-Flight Data Load.* Nov. 2019.

[216] S. van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida. "RIDL: Rogue In-Flight Data Load". In: *40th IEEE Symposium on Security and Privacy (S&P).* May 2019.

[217] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich. "VC3: Trustworthy Data Analytics in the Cloud Using SGX". In: *36th IEEE Symposium on Security and Privacy (S&P).* 2015, pp. 38–54.

[218] M. Schwarz, D. Gruss, M. Lipp, C. Maurice, T. Schuster, A. Fogh, and S. Mangard. "Automated Detection, Exploitation, and Elimination of Double-Fetch Bugs Using Modern CPU Features". In: *13th ACM Asia Conference on Computer and Communications Security (AsiaCCS).* 2018, pp. 587–600.

[219] M. Schwarz, S. Weiser, and D. Gruss. "Practical Enclave Malware with Intel SGX". In: *16th Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA).* 2019, pp. 177–196.

[220] M. Schwarz, C. Canella, L. Giner, and D. Gruss. "Store-to-Leak Forwarding: Leaking Data on Meltdown-Resistant CPUs". In: *arXiv preprint arXiv:1905.05725* (2019).

[221] M. Schwarz and D. Gruss. "How Trusted Execution Environments Fuel Research on Microarchitectural Attacks". In: *IEEE Security & Privacy Magazine Special Issue on Hardware-Assisted Security* (2020).

[222] M. Schwarz, M. Lipp, C. Canella, R. Schilling, F. Kargl, and D. Gruss. "Context: A generic approach for mitigating spectre". In: *27th Annual Network and Distributed System Security Symposium (NDSS).* 2020.

[223] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss. "ZombieLoad: Cross-Privilege-Boundary Data Sampling". In: *26th ACM Conference on Computer and Communications Security (CCS).* Nov. 2019, pp. 753–768.

[224] M. Schwarz, M. Schwarzl, M. Lipp, J. Masters, and D. Gruss. "NetSpectre: Read Arbitrary Memory Over Network". In: *European Symposium on Research in Computer Security (ESORICS).* 2019, pp. 279–299.

[225] M. Schwarz, S. Weiser, D. Gruss, C. Maurice, and S. Mangard. "Malware Guard Extension: Using SGX to Conceal Cache Attacks". In: *14th Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA).* 2017.

[226] M. Schwarzl, T. Schuster, M. Schwarz, and D. Gruss. "Speculative Dereferencing of Registers: Reviving Foreshadow". In: *arXiv preprint arXiv:2008.02307* (2020).

[227] J. Seo, B. Lee, S. Kim, and M.-W. Shih. "SGX-Shield: Enabling Address Space Layout Randomization for SGX Programs". In: *24th Annual Network and Distributed System Security Symposium (NDSS)*. 2017.

[228] H. Shacham. "The Geometry of Innocent Flesh on the Bone: Return-into-libc Without Function Calls (on the x86)". In: *14th ACM Conference on Computer and Communications Security (CCS)*. 2007, pp. 552–561.

[229] M.-W. Shih, S. Lee, T. Kim, and M. Peinado. "T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs". In: *24th Annual Network and Distributed System Security Symposium (NDSS)*. Feb. 2017.

[230] S. Shinde, Z. L. Chua, V. Narayanan, and P. Saxena. "Preventing Page Faults from Telling Your Secrets". In: *11th ACM Asia Conference on Computer and Communications Security (AsiaCCS)*. 2016, pp. 317–328.

[231] S. Shinde, D. L. Tien, S. Tople, and P. Saxena. "Panoply: Low-TCB Linux Applications With SGX Enclaves". In: *24th Annual Network and Distributed System Security Symposium (NDSS)*. 2017.

[232] S. Shinde, S. Tople, D. Kathayat, and P. Saxena. *Podarch: Protecting Legacy Applications with a Purely Hardware TCB*. Tech. rep. National University of Singapore, 2015.

[233] D. Skarlatos, M. Yan, B. Gopireddy, R. Sprabery, J. Torrellas, and C. W. Fletcher. "MicroScope: Enabling Microarchitectural Replay Attacks". In: *46th International Symposium on Computer Architecture (ISCA)*. 2019, pp. 318–331.

[234] J. Stecklina and T. Prescher. "LazyFP: Leaking FPU Register State using Microarchitectural Side-Channels". In: *arXiv preprint arXiv:1806.07480* (2018).

[235] R. Strackx, J. Noorman, I. Verbauwhede, B. Preneel, and F. Piessens. "Protected Software Module Architectures". In: *ISSE 2013 Securing Electronic Business Processes*. 2013, pp. 241–251.

[236] R. Strackx and F. Piessens. "Fides: Selectively Hardening Software Application Components Against Kernel-Level or Process-Level Malware". In: *19th ACM Conference on Computer and Communications Security (CCS)*. 2012, pp. 2–13.

[237] R. Strackx and F. Piessens. "The Heisenberg Defense: Proactively Defending SGX Enclaves Against Page-Table-Based Side-Channel Attacks". In: *arXiv preprint arXiv:1712.08519* (Dec. 2017).

[238] R. Strackx, F. Piessens, and B. Preneel. “Efficient Isolation of Trusted Subsystems in Embedded Systems”. In: *Security and Privacy in Communication Networks*. 2010, pp. 344–361.

[239] K. Sun, R. Branco, and K. Hu. *A New Memory Type Against Speculative Side Channel Attacks*. White Paper. 2019.

[240] A. S. Tanenbaum and A. S. Woodhull. *Operating Systems: Design and Implementation*. 2009.

[241] A. Tang, S. Sethumadhavan, and S. Stolfo. “CLKscrew: Exposing the Perils of Security-Oblivious Energy Management”. In: *26th USENIX Security Symposium*. 2017.

[242] OP-TEE. *Security Advisories*. 2019. URL: `https://www.op-tee.org/security-advisories`.

[243] Texas Instruments. *MSP430x1xx Family: User’s guide*. 2006.

[244] R. M. Tomasulo. “An Efficient Algorithm for Exploiting Multiple Arithmetic Units”. In: *IBM Journal of research and Development* 11.1 (1967), pp. 25–33.

[245] F. Tramer, F. Zhang, H. Lin, J.-P. Hubaux, A. Juels, and E. Shi. “Sealed-Glass Proofs: Using Transparent Enclaves to Prove and Sell Knowledge”. In: *2nd IEEE European Symposium on Security and Privacy (Euro S&P)*. 2017.

[246] C.-C. Tsai, D. E. Porter, and M. Vij. “Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX”. In: *USENIX Annual Technical Conference (ATC)*. 2017.

[247] J. Van Bulck. *ecc: Store EdDSA Session Key in Secure Memory*. Jan. 2017. URL: `https://github.com/gpg/libgcrypt/commit/5a22de9`.

[248] J. Van Bulck. “Secure Resource Sharing for Embedded Protected Module Architectures”. MA thesis. KU Leuven, 2015.

[249] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx. “Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution”. In: *27th USENIX Security Symposium*. Aug. 2018, pp. 991–1008.

[250] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx. “Breaking Virtual Memory Protection and the SGX Ecosystem with Foreshadow”. In: *IEEE Micro Top Picks from the 2018 Computer Architecture Conferences* 39.3 (2019), pp. 66–74.

[251] J. Van Bulck, D. Moghimi, M. Schwarz, M. Lipp, M. Minkin, D. Genkin, Y. Yuval, B. Sunar, D. Gruss, and F. Piessens. "LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection". In: *41st IEEE Symposium on Security and Privacy (S&P).* May 2020, pp. 54–72.

[252] J. Van Bulck, J. T. Mühlberg, and F. Piessens. "VulCAN: Efficient Component Authentication and Software Isolation for Automotive Control Networks". In: *33rd Annual Computer Security Applications Conference (ACSAC).* Dec. 2017, pp. 225–237.

[253] J. Van Bulck, J. Noorman, J. T. Mühlberg, and F. Piessens. "Towards Availability and Real-Time Guarantees for Protected Module Architectures". In: *Companion Proceedings of the 15th International Conference on Modularity (MASS).* Mar. 2016, pp. 146–151.

[254] J. Van Bulck, D. Oswald, E. Marin, A. Aldoseri, F. D. Garcia, and F. Piessens. "A Tale of Two Worlds: Assessing the Vulnerability of Enclave Shielding Runtimes". In: *26th ACM Conference on Computer and Communications Security (CCS).* Nov. 2019, pp. 1741–1758.

[255] J. Van Bulck and F. Piessens. "Tutorial: Uncovering and Mitigating Side-Channel Leakage in Intel SGX Enclaves". In: *8th International Conference on Security, Privacy, and Applied Cryptography Engineering (SPACE).* Dec. 2018, pp. 20–24.

[256] J. Van Bulck, F. Piessens, and R. Strackx. "Nemesis: Studying Microarchitectural Timing Leaks in Rudimentary CPU Interrupt Logic". In: *25th ACM Conference on Computer and Communications Security (CCS).* Oct. 2018, pp. 178–195.

[257] J. Van Bulck, F. Piessens, and R. Strackx. "SGX-Step: A Practical Attack Framework for Precise Enclave Execution Control". In: *2nd Workshop on System Software for Trusted Execution (SysTEX).* ACM, Oct. 2017, 4:1–4:6.

[258] J. Van Bulck, N. Weichbrodt, R. Kapitza, F. Piessens, and R. Strackx. "Telling your Secrets Without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution". In: *26th USENIX Security Symposium.* Aug. 2017, pp. 1041–1056.

[259] S. Van Schaik, C. Giuffrida, H. Bos, and K. Razavi. "Malicious Management Unit: Why Stopping Cache Attacks in Software is Harder Than You Think". In: *27th USENIX Security Symposium.* 2018, pp. 937–954.

[260] M. Völp, J. Decouchant, C. Lambert, M. Fernandes, and P. Esteves-Verissimo. “Enclave-Based Privacy-Preserving Alignment of Raw Genomic Information: Information Leakage and Countermeasures”. In: *2nd Workshop on System Software for Trusted Execution (SysTEX)*. ACM, 2017, 7:1–7:6.

[261] J. Wampler, I. Martiny, and E. Wustrow. “ExSpectre: Hiding Malware in Speculative Execution.” In: *26th Annual Network and Distributed System Security Symposium (NDSS)*. 2019.

[262] K. Wang and M. Franklin. “Highly Accurate Data Value Prediction Using Hybrid Predictors”. In: *30th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1997.

[263] W. Wang, G. Chen, X. Pan, Y. Zhang, X. Wang, V. Bindschaedler, H. Tang, and C. A. Gunter. “Leaky Cauldron on the Dark Land: Understanding Memory Side-Channel Hazards in SGX”. In: *24th ACM Conference on Computer and Communications Security (CCS)*. 2017, pp. 2421–2434.

[264] R. N. Watson, J. Woodruff, M. Roe, S. W. Moore, and P. G. Neumann. *Capability Hardware Enhanced RISC Instructions (CHERI): Notes on the Meltdown and Spectre Attacks*. Tech. rep. University of Cambridge, Computer Laboratory, 2018.

[265] N. Weichbrodt, P.-L. Aublin, and R. Kapitza. “sgx-perf: A Performance Analysis Tool for Intel SGX Enclaves”. In: *19th International Middleware Conference*. 2018, pp. 201–213.

[266] N. Weichbrodt, A. Kurmus, P. Pietzuch, and R. Kapitza. “AsyncShock: Exploiting Synchronisation Bugs in Intel SGX Enclaves”. In: *European Symposium on Research in Computer Security (ESORICS)*. 2016.

[267] S. Weiser, M. Werner, F. Brasser, M. Malenko, S. Mangard, and A.-R. Sadeghi. “TIMBER-V: Tag-Isolated Memory Bringing Fine-Grained Enclaves to RISC-V”. In: *26th Annual Network and Distributed System Security Symposium (NDSS)*. Feb. 2019.

[268] S. Weiser, L. Mayr, M. Schwarz, and D. Gruss. “SGXJail: Defeating Enclave Malware via Confinement”. In: *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*. 2019, pp. 353–366.

[269] S. Weiser, D. Schrammel, L. Bodner, and R. Spreitzer. “Big Numbers–Big Troubles: Systematically Analyzing Nonce Leakage in (EC) DSA Implementations”. In: *29th USENIX Security Symposium*. 2019.

[270] S. Weiser, R. Spreitzer, and L. Bodner. “Single Trace Attack Against RSA Key Generation in Intel SGX SSL”. In: *13th ACM Asia Conference on Computer and Communications Security (AsiaCCS)*. 2018, pp. 575–586.

[271] O. Weisse, J. Van Bulck, M. Minkin, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, R. Strackx, T. F. Wenisch, and Y. Yarom. “Foreshadow-NG: Breaking the Virtual Memory Abstraction with Transient Out-of-Order Execution”. In: *Technical Report* (Aug. 2018).

[272] J. Werner, J. Mason, M. Antonakakis, M. Polychronakis, and F. Monrose. “The SEVerESt Of Them All: Inference Attacks Against Secure Virtual Enclaves”. In: *14th ACM Asia Conference on Computer and Communications Security (AsiaCCS).* 2019, pp. 73–85.

[273] M. Werner, T. Unterluggauer, L. Giner, M. Schwarz, D. Gruss, and S. Mangard. “ScatterCache: Thwarting Cache Attacks via Cache Set Randomization”. In: *28th USENIX Security Symposium.* 2019, pp. 675–692.

[274] L. Wilke, J. Wichelmann, M. Morbitzer, and T. Eisenbarth. “SEVurity: No Security Without Integrity – Breaking Integrity-Free Memory Encryption with Minimal Assumptions”. In: *41st IEEE Symposium on Security and Privacy (S&P).* May 2020.

[275] J. Woodruff, R. N. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton, and M. Roe. “The CHERI Capability Model: Revisiting RISC in an Age of Risk”. In: *41st International Symposium on Computer Architecture (ISCA).* 2014, pp. 457–468.

[276] M. Xu, C. Qian, K. Lu, M. Backes, and T. Kim. “Precise and Scalable Detection of Double-Fetch Bugs in OS Kernels”. In: *39th IEEE Symposium on Security and Privacy (S&P).* 2018, pp. 661–678.

[277] Y. Xu, W. Cui, and M. Peinado. “Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems”. In: *36th IEEE Symposium on Security and Privacy (S&P).* 2015, pp. 640–656.

[278] M. Yan, J. Choi, D. Skarlatos, A. Morrison, C. W. Fletcher, and J. Torrellas. “InvisiSpec: Making Speculative Execution Invisible in the Cache Hierarchy”. In: *51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO).* 2018.

[279] Y. Yarom and N. Benger. “Recovering OpenSSL ECDSA Nonces Using the Flush+Reload Cache Side-Channel Attack”. In: *IACR Cryptology ePrint Archive* (2014), p. 140.

[280] Y. Yarom and K. Falkner. “Flush+Reload: A High Resolution, Low Noise, L3 Cache Side-Channel Attack”. In: *23rd USENIX Security Symposium.* 2014, pp. 719–732.

[281] Z. Zhang, Y. Cheng, D. Liu, S. Nepal, Z. Wang, and Y. Yarom. “PThammer: Cross-User-Kernel-Boundary Rowhammer through Implicit Accesses”. In: *arXiv preprint arXiv:2007.08707* (2020).

List of publications

Peer-reviewed international conference papers

- D. Moghimi, J. Van Bulck, N. Heninger, F. Piessens, and B. Sunar. “CopyCat: Controlled Instruction-Level Attacks on Enclaves”. In: *29th USENIX Security Symposium.* Aug. 2020, pp. 469–486.

- M. Busi, J. Noorman, J. Van Bulck, L. Galletta, P. Degano, J. T. Mühlberg, and F. Piessens. “Provably Secure Isolation for Interruptible Enclaved Execution on Small Microprocessors”. In: *33rd IEEE Computer Security Foundations Symposium (CSF).* June 2020, pp. 262–276.

- K. Murdock, D. Oswald, F. D. Garcia, J. Van Bulck, D. Gruss, and F. Piessens. “Plundervolt: Software-Based Fault Injection Attacks Against Intel SGX”. In: *41st IEEE Symposium on Security and Privacy (S&P).* May 2020, pp. 1466–1482.

- J. Van Bulck, D. Moghimi, M. Schwarz, M. Lipp, M. Minkin, D. Genkin, Y. Yuval, B. Sunar, D. Gruss, and F. Piessens. “LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection”. In: *41st IEEE Symposium on Security and Privacy (S&P).* May 2020, pp. 54–72.

- C. Canella, D. Genkin, L. Giner, D. Gruss, M. Lipp, M. Minkin, D. Moghimi, F. Piessens, M. Schwarz, B. Sunar, J. Van Bulck, and Y. Yarom. “Fallout: Leaking Data on Meltdown-Resistant CPUs”. In: *26th ACM Conference on Computer and Communications Security (CCS).* Nov. 2019, pp. 769–784.

- M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss. “ZombieLoad: Cross-Privilege-Boundary Data Sampling”. In: *26th ACM Conference on Computer and Communications Security (CCS).* Nov. 2019, pp. 753–768.

- J. Van Bulck, D. Oswald, E. Marin, A. Aldoseri, F. D. Garcia, and F. Piessens. “A Tale of Two Worlds: Assessing the Vulnerability of Enclave Shielding Runtimes”. In: *26th ACM Conference on Computer and Communications Security (CCS)*. Nov. 2019, pp. 1741–1758.
- C. Canella, J. Van Bulck, M. Schwarz, M. Lipp, B. von Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss. “A Systematic Evaluation of Transient Execution Attacks and Defenses”. In: *28th USENIX Security Symposium*. Aug. 2019, pp. 249–266.
- J. Van Bulck, F. Piessens, and R. Strackx. “Nemesis: Studying Microarchitectural Timing Leaks in Rudimentary CPU Interrupt Logic”. In: *25th ACM Conference on Computer and Communications Security (CCS)*. Oct. 2018, pp. 178–195.
- J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx. “Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution”. In: *27th USENIX Security Symposium*. Aug. 2018, pp. 991–1008.
- J. Gyselinck, J. Van Bulck, F. Piessens, and R. Strackx. “Off-limits: Abusing Legacy x86 Memory Segmentation to Spy on Enclaved Execution”. In: *International Symposium on Engineering Secure Software and Systems (ESSoS)*. June 2018, pp. 44–60.
- J. Van Bulck, J. T. Mühlberg, and F. Piessens. “VulCAN: Efficient Component Authentication and Software Isolation for Automotive Control Networks”. In: *33rd Annual Computer Security Applications Conference (ACSAC)*. Dec. 2017, pp. 225–237.
- J. Van Bulck, F. Piessens, and R. Strackx. “SGX-Step: A Practical Attack Framework for Precise Enclave Execution Control”. In: *2nd Workshop on System Software for Trusted Execution (SysTEX)*. ACM, Oct. 2017, 4:1–4:6.
- J. Van Bulck, N. Weichbrodt, R. Kapitza, F. Piessens, and R. Strackx. “Telling your Secrets Without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution”. In: *26th USENIX Security Symposium*. Aug. 2017, pp. 1041–1056.
- J. T. Mühlberg, S. Cleemput, A. M. Mustafa, J. Van Bulck, B. Preneel, and F. Piessens. “Implementation of a High Assurance Smart Meter using Protected Module Architectures”. In: *10th WISTP International Conference on Information Security Theory and Practice (WISTP)*. Aug. 2016, pp. 53–69.

- J. Van Bulck, J. Noorman, J. T. Mühlberg, and F. Piessens. “Towards Availability and Real-Time Guarantees for Protected Module Architectures”. In: *Companion Proceedings of the 15th International Conference on Modularity (MASS).* Mar. 2016, pp. 146–151.
- J. Van Bulck, J. Noorman, J. T. Mühlberg, and F. Piessens. “Secure Resource Sharing for Embedded Protected Module Architectures”. In: *9th WISTP International Conference on Information Security Theory and Practice (WISTP).* Aug. 2015, pp. 71–87.

Peer-reviewed international journal articles

- K. Murdock, D. Oswald, F. D. Garcia, J. Van Bulck, D. Gruss, and F. Piessens. “Plundervolt: How a Little Bit of Undervolting Can Create a Lot of Trouble”. In: *IEEE Security & Privacy Magazine Special Issue on Hardware-Assisted Security* (2020).
- J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx. “Breaking Virtual Memory Protection and the SGX Ecosystem with Foreshadow”. In: *IEEE Micro Top Picks from the 2018 Computer Architecture Conferences* 39.3 (2019), pp. 66–74.
- J. T. Mühlberg and J. Van Bulck. “Reflections on Post-Meltdown Trusted Computing: A Case for Open Security Processors”. In: *;login: the USENIX magazine* 43.3 (2018), pp. 6–9.
- J. Noorman, J. Van Bulck, J. T. Mühlberg, F. Piessens, P. Maene, B. Preneel, I. Verbauwhede, J. Götzfried, T. Müller, and F. Freiling. “Sancus 2.0: A Low-Cost Security Architecture for IoT Devices”. In: *ACM Transactions on Privacy and Security* 20.3 (2017), pp. 1–33.

Tutorials at international conferences, with abstracts published in proceedings

- J. Van Bulck and F. Piessens. “Tutorial: Uncovering and Mitigating Side-Channel Leakage in Intel SGX Enclaves”. In: *8th International Conference on Security, Privacy, and Applied Cryptography Engineering (SPACE).* Dec. 2018, pp. 20–24.

- J. T. Mühlberg and J. Van Bulck. “Tutorial: Building Distributed Enclave Applications with Sancus and SGX”. In: *48th International Conference on Dependable Systems and Networks (DSN).* June 2018.

Technical reports

- M. Busi, J. Noorman, J. Van Bulck, L. Galletta, P. Degano, J. T. Mühlberg, and F. Piessens. “Provably Secure Isolation for Interruptible Enclaved Execution on Small Microprocessors: Extended Version”. In: (Jan. 2020).
- O. Weisse, J. Van Bulck, M. Minkin, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, R. Strackx, T. F. Wenisch, and Y. Yarom. “Foreshadow-NG: Breaking the Virtual Memory Abstraction with Transient Out-of-Order Execution”. In: *Technical Report* (Aug. 2018).

FACULTY OF ENGINEERING SCIENCE
DEPARTMENT OF COMPUTER SCIENCE
IMEC-DISTRINET
Celestijnenlaan 200A box 2402
B-3001 Leuven
jo.vanbulck@cs.kuleuven.be
https://www.distrinet.cs.kuleuven.be