Improving Performance and Security of Intel SGX

Marina Minkin

Improving Performance and Security of Intel SGX

Research Thesis

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science

Marina Minkin

Submitted to the Senate of
the Technion — Israel Institute of Technology
Tevet 5778 Haifa December 2018

The research thesis was done in the Computer Science department under the supervision of Prof. Mark Silberstein (Electrical Engineering department, Technion).

Parts of this thesis have been published as articles by the author and research collaborators during the course of the author's Master's research period, the most up-to-date versions of which being:

1. Van Bulck, J., Minkin, M., Weisse, O., Genkin, D., Kasikci, B., Piessens, F., Silberstein, M., Wenisch, T.F., Yarom, Y. and Strackx, R., 2018. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In 27th USENIX Security Symposium (USENIX Security 18) (pp. 991-1008).

2. Orenbach, M., Lifshits, P., Minkin, M. and Silberstein, M., 2017, April. Eleos: ExitLess OS services for SGX enclaves. In Proceedings of the Twelfth European Conference on Computer Systems (pp. 238-253). ACM.

Acknowledgments

I would like to express my special appreciation and thanks to my advisor Professor Mark Silberstein, you have been a tremendous mentor for me. You are one of the smartest people I know I would like to thank you for encouraging my research and for allowing me to grow as a research scientist. Your advice on my research as have been invaluable.

I would like to thank Boris Van-Sosin, Itai Dabran, Lina Maudlej, Boaz Sternfeld and all the residents of the 4th floor at Taub for all the support during my studies. I woul d also like to thank Tanya Brokhman, Vasilis Dimistas, Shai Bergman and all the other members of the ACG.

During my studies, I was also fortunate to have collaborated, and learned from many wonderful researchers: Pavel Lifshits, Meni Orenbach, Ofir Weisse, Daniel Genkin, Baris Kasikci, Thomas F. Wenisch Yuval Yarom, Jo Van Bulck, Frank Piessens and Raoul Strackx.

Finally, the Computer Science department, the Electrical Engineering department and the TCE at Technion provided me with a wonderful research environment, from which I greatly benefited and enjoyed.

The generous financial support of the Technion, the Technion Hiroshi Fujiwara cyber security research center and the Israel cyber bureau is gratefully acknowledged.

Contents

List of Figures

Abstract

Intel Software Guard Extensions (SGX) are a new set of CPU instructions which enable trusted and *isolated* execution of selected sections of application code in hardware containers called *enclaves*. An enclave acts as a reverse sandbox: its private memory and execution state are isolated from any software outside the enclave, including an OS and/or a hypervisor, yet the code running in the enclave may access *untrusted* memory of the owner process. While SGX provides the convenience of a standard `x86` execution environment inside the enclave, there are important differences in the way enclaves manage their private memory and interact with the host OS.

In this work, we try to understand better and improve the performance and security of SGX enclaves. First, we present Foreshadow, which is an attack on SGX that extracts full memory dumps of SGX enclaves thereby compromising SGX's integrity and confidentiality guarantees.

Next, previous work (Eleos, EuroSys'17) has shown that performance of SGX paging can be improved by using software user-level paging, called SUVM. In this thesis we show that SUVM for a single enclave is not effective for multi-enclave systems, and introduce Multi-SUVM. A system that supports SUVM for multi enclave environment with kernel space and in-enclave modifications. We show in Multi-SUVM up to 65% speedup, and up to $3.4\times$ higher throughput than SUVM.

Chapter 1

Introduction

Intel Software Guard eXtensions (SGX) is a new technology for `x86` processors that enables a user to execute sensitive code handling secret data on a remote `x86` machine. SGX protects the user's sensitive data from a (potentially malicious) remote host owner and *Operating System* (OS). In addition, SGX provides the user with tools to verify that the owner of a remote host cannot read the program's secrets or compromise the integrity of the computation that the program is trying to perform. For example, a content streaming company could use SGX to protect their *Digital rights management* (DRM), so the client would be able to use the content *only* within a specific application and for a pre-determined number of times (e.g., viewing a movie only once using an authorized player).

To support private and secure code execution, SGX provides isolated execution environments, called *enclaves*, which offer confidentiality and integrity guarantees to programs running inside them. While SGX's security properties strongly rely on the processor's hardware implementation, SGX removes all other system components (such as the memory hardware, the firmware, and the operating system) from the Trusted Computing Base (TCB). In particular, SGX ensures that the processor and memory states of an enclave are only accessible to the code running inside it and that they remain out of reach to all other enclaves and software, running at any privilege level, including a potentially malicious operating system (OS), and/or the hypervisor. Finally, SGX provides a remote attestation mechanism, which allows enclaves to prove to remote parties that they have been correctly initialized on a genuine (hence, presumed secure) Intel processor.

To achieve this protection, SGX encrypts and authenticates all data that exits the CPU die, thereby protecting making it impossible for malicious applications or even OS and Hypervisor cannot read or modify enclave's private code and data. All write attempts will be discarded, and all read attempts will return `0xFF`, regardless of the data's actual value.

Next, while ensuring integrity and confidentiality, enclaves are part of user-level applications, and as such are managed by the operating system. Thus, even though the OS is not allowed to read their content, it has tools to manage enclaves: The OS can stop and resume enclave execution at will in order to manage their scheduling and let them run alongside regular applications. The OS also manages SGX's virtual to physical memory mapping via dedicated interfaces, without compromising SGX's security guarantees.

1.1 Our Contrition

Unfortunately, like any new technology, SGX has some flaws. In this thesis we are trying to identify the problems it has, and try to fix them. Since deploying fixes for SGX in hardware is usually much harder than deploying software fixes, we tried to present software workaround to the issues we identified. At a high level, in this work we found two limitations of Intel SGX, and their potential to be fixed in software:

- We present a vulnerability in Intel SGX that allows breaking all the security guarantees of SGX. In particular, unlike previous work which extracted secret information from an SGX enclave by observing secret-dependent memory access-patterns [100], we show that it is possible to extract enclave secrets even from a *perfect* enclave, which does not contain any software vulnerabilities.
- Next, we present a performance limitation related to the SGX memory management mechanism, and show how to achieve a significant speedup using software techniques only.

This thesis is organized as follows. First, in chapter 3, we present Foreshadow, a microarchitectural side channel attack that breaks SGX's confidentiality, integrity and trust. First, we show how an attacker can read the entire memory contents of any victim enclave running on the target machine. Next, we demonstrate how to leverage this ability to violate the integrity and confidentiality of SGX's sealing mechanism, which protects enclaves' long term storage. Finally, we show how to recover the machine's attestation key and breach the SGX trust mechanism, allowing attackers to masqurade as any victim enclave. We empirically demonstrate our attack on several up-to-date SGX-enabled machines, including machines updated with countermeasures against previous speculative execution attacks, such as Spectre and Meltdown [53, 60].

The second result in this thesis will show that running memory-demanding multi-enclave server applications leads to significant performance degradation. Previous work [71, Eleos] discovered that the SGX memory management mechanism presents a major bottleneck to SGX execution time. To address this, Eleos presented a novel Secure User-managed Virtual Memory (SUVM) abstraction that implements application-level paging inside the enclave. Unfortunately, for the case of multiple enclaves running on the same physical hardware, the work of [71] does not result in any performance gains. Addressing this issue, in chapter 4, we present Multi-SUVM, which analyzes and solves the overhead of hardware paging for multiple enclave environments running on the same hardware. Analyzing the performance of our solution, we show up to 65% speedup over a naive SUVM implementation in the multi-enclave case.

1.2 Related work

System support for trusted execution. Intel SGX SDK [8, 20, 34, 61] introduces the `OCALL` interface to allow untrusted function calls from enclaves, which force the enclave to exit to perform such a call. Eleos replaces `OCALL` with a more efficient exit-less implementation.

The authors of Intel SGX 2 [62, 99] acknowledges the performance overheads of `OCALL`. However, they do not consider indirect costs.

Haven [12], Graphene [88] and PANOPLY [84] provide secure execution for legacy applications inside SGX enclaves without application code modifications, by providing a compatibility layer that deals with enclave execution. Our work is complementary, and can be used to improve applications performance, as we do with Graphene (§ 4.5). Finally, the integration of Eleos adds only a few hundred lines of code into the TCB.

VC3 [77] uses SGX to achieve confidentiality and integrity as part of the MapReduce framework. Ryoan [35] is a system used to execute enclaves in a sandbox and distributed environment. Ryoan proposed use cases include health analysis and image processing modules, which like VC3 are both I/O and memory-demanding. Thus, using Eleos with it might be beneficial.

Closest to our work, SCONE [10] leverages SGX to provide isolated execution for Linux containers [63]. SCONE employs an independently developed technique that is similar to Eleos's RPC mechanism. However, the authors do not analyze the costs of exits, as we do in this paper. Furthermore, Eleos enhances the RPC mechanism to reduce LLC pollution by using CAT. Finally, we extend the scope of exit-less services to virtual memory, and show significant performance benefits for workloads exceeding the size of PRM.

Asynchronous system calls. The authors of FlexSC [86] observe the need to reduce user/kernel transitions to optimize the system call performance, proposing asynchronous system call execution with batching. Eleos's RPC service is similar. Furthermore, our analysis of LLC pollution and TLB flushes was inspired by that of FlexSC.

System services for GPUs. This work adapts some of the ideas introduced earlier to provide system services on GPUs. Specifically, GPUfs [85] and GPUnet[51] are systems for efficient I/O abstractions for GPUs. Like them, Eleos uses an RPC infrastructure to reduce transition costs. ActivePointers [81] is a software address translation system for GPUs that provides support for memory mapped files. Eleos adopts this concept for *s*pointers but extends it by redesigning its paging system to support secure paging and optimizing it for execution on CPUs.

Virtual machine ballooning. Eleos applies the idea of coordinated memory management among virtual machines [93] to enclaves. Thus enclaves, like virtual machines, may evict pages according to their eviction policy. However, unlike VM ballooning, Eleos adds its own trusted swapping thread, and can directly modify the enclave's working set.

Distributed shared memory. Shasta [76] is a software based distributed shared memory system, which supports a shared address space across a cluster. To maintaining coherency in fine-grain granularity, Shasta instruments load and store instructions to test for memory state validity. Eleos, adapts this concept into *s*pointers, yet extends it to support full virtual memory management in a secure fashion.

Exit-less interrupts for optimized I/O in VMs. The concept of exit-less interrupt handling in Virtual Machines introduced in ELI [26] inspired us to consider techniques for eliminating costly exits in enclaves. ELI, however, focuses on interrupt handling in the context of optimized I/O performance, and does not consider avoiding exits due to page faults.

Exploiting Operating System Control. Excluding the OS from the TCB gives potential attackers powers that do not exist in more traditional attack models. The controlled channel attack [100] uses the OS's control over the enclave's memory mapping to leak information about the enclave's operation. Under this attack, the OS protects the enclave pages by manipulating virtual memory permissions, preventing access. When the enclave attempts to access a protected page, the processor traps to the OS, which records the access before enabling permission and allowing the enclave to continue. The attack can recover high-resolution data, including outlines of JPEG images [100] and cryptographic keys [83, 97].

In addition to page faults, the operating system can also monitor other effects of page table access, including whether a page is dirty or has been accessed, and the caching status of the page table or the translation lookaside buffer [91, 94]. This approach reduces the overhead and side effects of the controlled channel attack.

Microarchitectural Attacks. Microarchitectural side channel attacks are considered outside the scope for SGX [41, 42], hence it is not surprising that SGX is vulnerable to such attacks. However, operating system control gives attackers additional powers when deploying microarchitectural attacks.

One such power is the ability to interrupt the victim enclave frequently, allowing the attacker to monitor the cache [65, 66] or the branch predictor unit [21, 58] after almost every victim instruction. Attackers can also use the operating system control to reduce system activity and the noise it induces on microarchitectural attacks. For example, the operating system can block interrupts and ensure that the attacker thread and the enclave execute on the two hyperthreads of the same core [16]. Furthermore, the operating system has access to performance information that is not normally available to, or is very noisy when used from user processes [16, 27, 58].

Speculative Execution Attacks on Enclaves. Both O'Keeffe et al. [68] and Chen et al. [19] demonstrate that the Spectre attack [53] works on SGX enclaves. Both attacks are demonstrated only against specially crafted enclaves, and as such are more at the proof-of-concept stage rather than being practical attacks.

Both attacks rely on executing vulnerable code within the victim enclave. Our attack, in contrast, does not require any specific code in the victim enclave and can extract all of the memory contents of the enclave without executing any of the enclave code. While existing work shows vulnerable gadgets exist in the SGX SDK [68], these attacks can be mitigated by patching the SDK and removing these gadgets. Our attack does not rely on vulnerabilities in SDK.

Denial of Service Attacks. The use of encrypted and authenticated memory protects enclaves from subversion via attacks, such as Rowhammer [52], that modify the contents of memory. However, the pitfall of this protection is that Rowhammer attacks can now be used to mount a denial-of-service attack on the entire machine, as the unauthorized memory changes lock the processor, requiring a power cycle for execution to resume [30, 49].

Other Attacks on Enclaves. SGX offers only limited protection to vulnerabilities in the code running within enclaves, which can compromise enclave security. One example is Edger8r [46], a timing leakage in the Intel SGX SDK that allows attackers to retrieve some contents of the attacked enclave. Lee et al. [56] show that memory corruption vulnerabilities in SGX enclaves can be exploited and explain how to mount Return Oriented Programming [80] attacks on such vulnerabilities. Finally, AyncShock [96] shows how to exploit synchronization bugs to hijack the enclave control flow.

Attacks from Enclaves. Because the contents of enclaves cannot be observed by the operating system, malicious code may run undetectably in enclaves. SGX provides some protections against malicious enclaves. In particular, enclaves execute in user space, and thus cannot invoke privileged instructions. Furthermore, several non-privileged instructions are disabled in enclaves, for example, `IN` or `OUT`, which perform input/output operations and `SMSW`, which may leak kernel information.

However, this protection does not extend to microarchitectural attacks. Consequently, enclaves can leak information through cache attacks [78] and modify the contents of memory using the Rowhammer attack [30].

Defenses. Several mitigation techniques for SGX attacks have been proposed. T-SGX [82] uses a Transactional Synchronization Extensions transaction to catch interrupts, so the enclave can identify the page faults used in the controlled channel attack [100] and the timer interrupts used in other attacks [21, 58, 65]. Shinde et al. [83] defend against the controlled channel attack by forcing a deterministic access pattern to enclave pages. HyperRace [18] aims to protect against attacks that rely on hyperthreading, such as Brasser et al. [16]. HyperRace uses a data race to implement a co-location test with which an enclave thread verifies that it is co-located on the same core as a dedicated *shadow* thread, thereby ensuring that no attacker concurrently executes on the same core. Similar techniques are used in Varys[69]. SGX-Shield [79] randomizes enclaves' address space layout to protect against memory-based attacks, including the controlled channel attack. DR.SGX [15] protects from some cache-based attacks using fine-grained randomization of enclaves' data locations. SGXBounds [55] provides memory safety for SGX enclaves by tagging pointers with bounds information.

Chapter 2

Background

Trusted Execution Environments (TEEs) are a set of architectural extensions, recently introduced in commodity processors, which collectively provide strong security guarantees to software running in the presence of powerful adversaries. TEEs' promise to allow secure execution on adversary-controlled machines has spawned many new applications [11, 54, 75, 89, 103, 105], both in academia [10, 12, 13, 35, 57, 77, 87, 91, 100] and industry [2, 9, 22, 39]. Whereas several TEEs have been proposed (e.g., ARM's TrustZone [7] or AMD's Secure Encrypted Virtualization [1]), the currently prevailing TEE implementation is Intel's Secure Guard eXtensions (SGX) [8, 37].

To support private and secure code execution, SGX provides isolated execution environments, called *enclaves*, which offer confidentiality and integrity guarantees to programs running inside them. While SGX's security properties strongly rely on the processor's hardware implementation, SGX removes all other system components (such as the memory hardware, the firmware, and the operating system) from the Trusted Computing Base (TCB). In particular, SGX ensures that the processor and memory states of an enclave are only accessible to the code running inside it and that they remain out of reach to all other enclaves and software, running at any privilege level, including a potentially malicious operating system (OS), and/or hypervisor. At a high level, SGX achieves these strong security guarantees by encrypting enclave memory and protecting it with a secure authentication code, making the associated cryptographic keys inaccessible to software. Finally, SGX provides a *remote attestation* mechanism, which allows enclaves to prove to remote parties that they have been correctly initialized on a genuine (hence, presumed secure) Intel processor.

Notwithstanding its strong security guarantees, SGX does not protect against microarchitectural side channel attacks. Such side channel attacks exploit subtle timing variations resulting from contention on CPU microarchitectural resources to extract otherwise-unavailable secret information. Since their introduction over a decade ago [14, 72, 73, 90], microarchitectural attacks have been used to break numerous cryptographic implementations [25, 47, 102], track user behaviors [28, 59, 70], and create covert channels [29, 95]. Moreover, recent works combine microarchitectural attacks with speculative execution [38, 53, 60], allowing the attacker to read the entire address space of victim processes or of the operating system.

In terms of protection against side channel attacks, Intel acknowledges that "SGX does not defend against this adversary" [42, Page 115] arguing that "preventing side channel attacks is a matter for the enclave developer" [41]. Indeed, starting with the controlled channel attack [100], numerous works have demonstrated side channel attacks on or from SGX enclaves (See Section 1.2).

Crucially, all the previously published attacks on SGX exploit *existing* side channel vulnerabilities [100], coding bugs [55], or speculative execution gadgets [19, 68] in enclaves' code to leak sensitive data.

2.1 Memory Encryption

To protect enclaves' data from the operating system, the firmware of the machine reserves a range of memory called the *Processor Reserved Memory* (PRM), which contains a region encrypted using the Intel Memory Encryption Engine (MEE) [31, 32], as well as metadata used for SGX and for MEE.

The main aim of MEE is to protect against an adversary that has physical access to the memory of the host machine. To provide confidentiality of the data, MEE encrypts the data in the PRM. To protect the data integrity, MEE maintains a Merkle Tree [64] of stateful Message Authentication Codes (MACs), which ensure unauthorized modifications, including rollbacks, of the memory are detected. MEE operates between the LLC and the system's memory. Cache lines are encrypted when written to memory and decrypted and validated when read from memory.

2.2 EPC and PRM

An enclave may access an isolated *trusted* memory space called the *enclave page cache*, or *EPC*, which is accessible only from that enclave. Physical memory that stores the EPC contents is limited to the size of the PRM (128MB today). Therefore, EPC introduces an extra level of virtual memory with its own demand paging system. Under PRM pressure, EPC pages are securely evicted to untrusted memory and paged in on-demand by the SGX driver in response to EPC page faults.

2.3 Enclave Creation

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

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

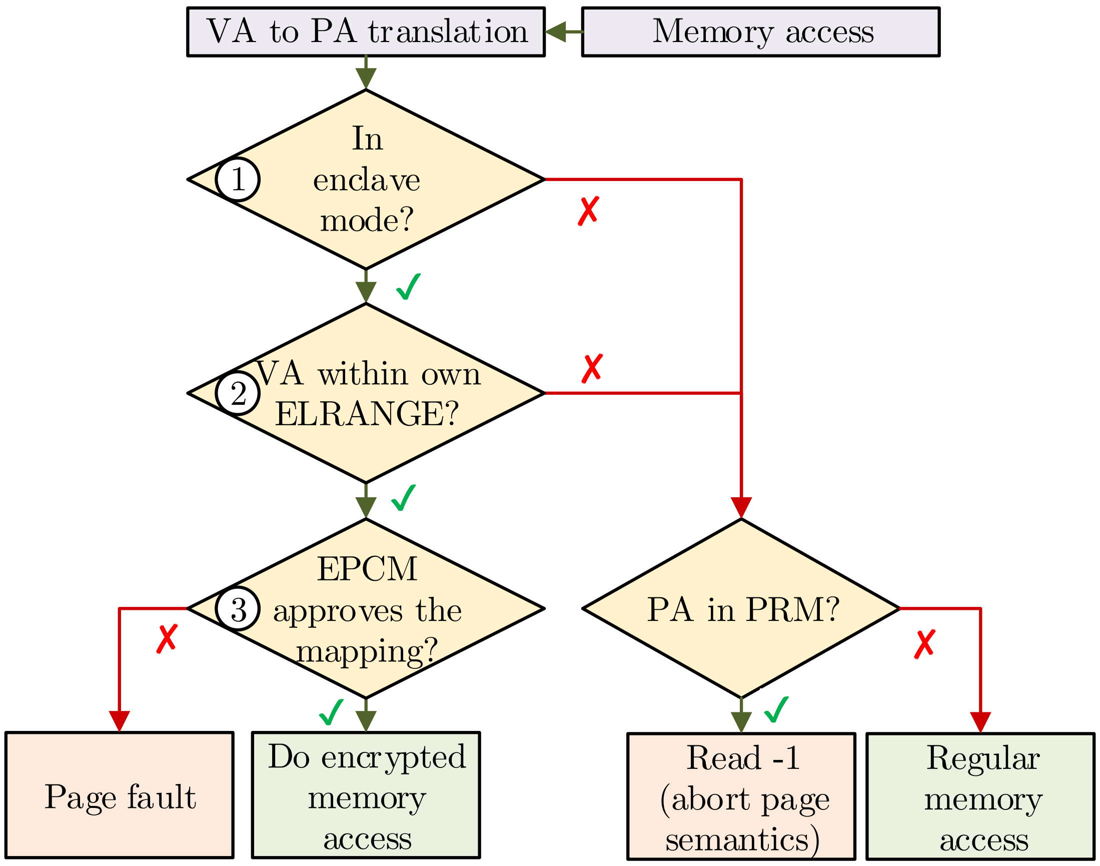

Figure 2.1: SGX's memory access flow, as described in [61].

2.4 Memory Management

To protect the contents of the ELRANGE, pages within this range must map to the PRM. More specifically, part of the PRM is used for the *Enclave Page Cache* (EPC), in which enclaves' pages are stored. Each page within the ELRANGE of an enclave can be either loaded in the EPC, where the enclave can use it, or (securely) stored outside the EPC, where the OS must load it into the enclave before it may be accessed. The operating system is given the primitives required for managing the EPC, without being able to observe or to modify the contents of the EPC pages. These mechanisms enable the operating system to implement a secure paging facility for enclaves whose footprint exceeds the available capacity of the EPC.

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

Because SGX enclaves execute within the virtual address space of a process, the translation of enclave addresses to physical addresses must be trusted. However, the operating system controls the mapping of virtual to physical addresses and can change this mapping at will. Instead of ensuring the correctness of the operating system's page table, SGX maintains an internal data structure called the

Enclave Page Cache Map (EPCM), which tracks the mapping and the identities of frames within the EPC. The EPCM provides the reverse mapping of the physical-to-virtual address mapping encoded in the page tables.

The purpose of the EPCM is to protect against attempts to bypass the SGX protection by mapping an EPC page at a different virtual address. This protection is achieved by adding several validation tests following a virtual to physical address translation. Figure 2.1 shows a flow chart of the validation process. First (Step ①), the processor checks whether the access is from within an enclave. Non-enclave code is blocked from any access to the PRM by providing *abort page semantics* for such accesses. That is, reads from the PRM return an all-one data (`0xFF`) and writes to the PRM are ignored.

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

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

Chapter 3

FORESHADOW: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution

3.1 Overview

Even though SGX has lots of benefits such as little performance overhead, support of existing x86 code, easy integration with existing OSes and a strong attacker threat model, little is known, about the side channel security of SGX enclaves that do not contain existing side channel vulnerabilities or other coding bugs. Thus, in this thesis, we ask:

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

Next, we observe that whereas SGX's confidentiality guarantees in the presence of side channels have been studied before [58, 91, 96, 100], SGX's integrity guarantees in the presence side channels have received much less research attention. Thus, we ask:

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

Finally, given the importance of SGX *remote attestation* [50] in establishing trust in the SGX ecosystem, we finally ask:

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

3.1.1 Our Contribution

We answer all three questions in the affirmative. We answer the first question by presenting several new attacks that compromise SGX's confidentiality guarantees. We then use our attacks on SGX's confidentiality properties to break SGX's integrity guarantees, thereby answering the second question. Finally, we use these attacks to recover the machine's private attestation keys, thereby breaking SGX's attestation protocol and answering the third question. *As such, until mitigated were out, our* *results fully compromised the basis of trust in the SGX ecosystem, both in terms of confidentiality and integrity.*

Breaking SGX's Confidentiality. Our first attack exploits the speculative execution features present in all SGX-enabled Intel CPUs to read the entire address space of victim enclaves. Crucially, unlike previous Spectre-style speculative execution attacks on SGX [19, 67], our attack *does not require* any code gadget or any other form of cooperation from the victim enclave. In fact, our attack reads all the secrets of the victim enclave without requiring that enclave to execute any instruction. In particular, our attack bypasses all currently proposed side-channel mitigations for SGX [18, 69, 79, 82], as well as proposed countermeasures for speculative execution attacks [38, 40].

At a high level, an attacker can maliciously retrieve memory contents of the victim enclave by remapping the victim memory into the address space of the attacker enclave, which allows bypassing SGX's default enclave-isolation mechanism of abort page semantics. The attacker then prefetches the victim's data into the L1 cache without the victim's involvement by leveraging the cache behavior of SGX paging instructions, thus dramatically improving the attack effectiveness. Finally, the attacker uses speculative execution to perform segmentation-fault-free access to the victim's memory. We refer the reader to Section 3.3 for additional details.

Breaking the Integrity of Sealed Data. Going beyond attacks on the SGX confidentiality properties, we show the first attack that compromises SGX's *long-term storage integrity* guarantees. More specifically, in addition to secure computation, SGX also aims to provide private and authenticated long-term storage, which is implemented via a special *sealing* Application Programming Interface (API) [8]. This storage mechanism allows enclaves to encrypt and verify data stored by the (untrusted) operating system.

As we show in Section 3.4, we can use our attack on SGX's confidentiality to extract the sealing key from a victim enclave that uses the SGX sealing mechanism. We note that extracting this key from the address space of an enclave is challenging as the SGX Software Development Kit (SDK) implementation of the sealing API [36] zeros out the sealing key from memory immediately after using it, thereby requiring our attack to intercept the key before it is destroyed. After recovering the sealing key, we use it to unseal and read the sealed information, then modify and reseal it. As SGX provides no means to detect such a change, the victim might now operate on data corrupted by the attacker.

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

As we show in Section 3.5, we can mount the aforementioned attacks on the SGX *Quoting Enclave*, dump its entire address space, and retrieve its sealing key. Besides being the first documented attack on a production enclave, this attack is particularity devastating as we use the sealing key to unseal the persistent storage of the Quoting Enclave, which contains the machine's private attestation key. With this key in hand, we can construct malicious SGX simulators that pass the entire attestation process, masquerading as enclaves that are allegedly running on genuine Intel processors with the SGX security guarantees. As the simulated enclaves do not offer any security guarantees, this attack undermines the trustworthiness of SGX's attestation mechanism.

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

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

3.1.2 Targeted Hardware and Current Status

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

3.1.3 Threat Model

The Attacker Controls the Operating System. We assume that the attacker controls the operating system and, in particular, can install kernel modules or otherwise execute code with supervisor (ring-0) privileges. The attacker is therefore capable of controlling the virtual-to-physical memory mapping of processes and execute SGX instructions. We note that while we assume a very strong adversary, such a malicious attacker is well within the SGX threat model. Specifically, an SGX enclave is designed to be "protected even when the BIOS, VMM, OS, and drivers are compromised" [44].

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

The Attacker Can Observe *When* a Secret is in Memory. We assume the attacker is capable of launching the victim enclave, and estimating *when* the enclave contains secrets. In a simple scenario, the observation is made by the attacker invoking specific enclave functions that generate a secret locally or request a secret from an external party. In section 3.4, we explain how an attacker can pause an enclave to retrieve a secret in memory before it is erased.

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

No Assumptions on the Victim's Code. Our attack makes no assumptions on the victim's code or on its layout in memory. Unlike Spectre [38, 53], our attack does not require any special code gadgets [68]. Even if the code is encrypted or Address Space Layout Randomization (ASLR) is used [79], our attack is capable of reading the contents of the enclave *after* it is decrypted and after ASLR code placement randomization is completed.

3.2 Background

3.2.1 The Flush+Reload Attack

Flush+Reload [33, 101] is a cache-based microarchitectural attack technique that detects access to a shared memory location. The technique consists of two main operations. The *flush* operation evicts the contents of a monitored memory location from the cache. Typically, this is done using a dedicated instruction, such as the `clflush` instruction on x86 architecture. The *reload* operation then measures the time it takes to access the monitored location. Based on this time, it determines whether the contents of the monitored location was cached prior to its execution.

In a typical attack scenario, an attacker flushes one or more monitored locations. It then either executes an operation it wants to analyze or waits until it is naturally executed by the victim. The attacker then reloads the monitored locations, while recording the amount of time required to perform the reload. As the analyzed operation accesses (and thereby caches) some of the monitored locations but not others, the attacker is then able to learn information about the memory access pattern of the analyzed operation. Finally, as memory access patterns often reveal information about the *inputs* of the analyzed operation, the attacker is often capable of completely recovering these inputs. Flush+Reload has been extensively used for side channel attacks [6, 23, 24, 28, 48, 53, 60, 74, 101, 104].

3.2.2 Speculative Execution

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

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

3.2.3 Spectre and Meltdown

When the CPU abandons speculatively executed instructions, it does not fully revert the side-effects these instructions have on its microarchitectural state. Spectre and Meltdown [53, 60] exploit these side-effects to leak information across protection domains. The attacks cause the CPU to speculatively execute a *gadget* that implements the transmitting side of a covert channel, sending information that would otherwise be unavailable to the receiver.

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

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

3.3 Reading Enclave Data

In this section, we describe our first attack, which allows us to read data located within the address space of some victim SGX enclave. At a high level, our attack is constructed to coerce Steps ①, ②, and ③ in Figure 2.1 to result in a page fault due to a failure of verification of an EPCM test. We then use a variant of the Meltdown attack to subvert the page fault and read the victim's data.

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

3.3.1 Establishing a Cache-Based Covert Channel

Similar to prior work [53, 67], we leverage the Flush+Reload covert channel. The channel consists of three operations, abstracted as functions in Figure 3.1.

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

Transmitting a Byte. The `send` function takes a one byte argument `data`, and transmits it via the covert channel by accessing a corresponding element from `probeArray` (Line 7), thereby bringing it to the CPU's cache. As we can only distinguish accesses at a cache-line granularity, we multiply (`data`) by 256 before accessing `probeArray`. With a cache-line size of 64 bytes,

```
1  prepare() {
2     for (i=0; i < 65536; i++)
3        flush(probeArray[i]);
4  }
5
6  send(data) {
7     t = probeArray[data * 256];
8  }
9
10 recieve() {
11    for (i = 0; i < 256; i++) {
12       // mix guess to avoid prefetching probeArray
13       guess = ((i * 167) + 13) % 256;
14       // compute address to probe
15       addr = &probeArray[guess*256];
16       t1 = rdtscp(); // read timer
17       temp = *addr;  // access probing array
18       t2 = rdtscp(); // read timer
19       if (t2-t1 <= CACHE_HIT_THRESHOLD){
20          results[guess]=1;
21       } else results[guess]=0;
22    }
23    return results;
24 }
```

Figure 3.1: Pseudocode of the Flush+Reload covert channel.

spreading indices by multiples of 256 ensures that different values of `data` index to different cache lines. We find that using lower multipliers increases noise level due to prefetcher activity.

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

We note here that while the Flush+Reload channel is usually clean, some measurement errors remain possible, typically due to unrelated system activity. Consequently, instead of returning a single guess for the transmitted data, `receive` returns a *vector* of results that indicates which values where possibly transmitted by `send`. In Section 3.3.5 we show how to avoid these errors and correctly recover the transmitted data. Finally, to avoid the CPU's prefetcher modifying the cache state, `receive` does not access the indices of `probeArray` sequentially and follows the approach of previous works [53, 67] by applying a simple linear permutation to the order of accesses (Line 13).

3.3.2 The Malicious Hosting Process

The attacker launches the victim enclave in a process and identifies the virtual address range (ELRANGE) of the victim enclave inside the process's address space. The range can be located either via a malicious driver or by inspecting the contents of `/proc/pid/maps`. At this point, a naive attacker might attempt to read directly the enclave address space. However, the virtual addresses of the victim enclave are mapped to physical addresses residing in the EPC, which is part of the PRM. Consequently, as Figure 2.1 shows, such access would result in abort page semantics, i.e., the read returns a value with all bits set, regardless of the actual memory content. Alternatively, the attacker may attempt to read the enclave's data by mounting a speculative execution attack (e.g., Meltdown). However, we found that such attempts also result in abort page semantics.

3.3.3 The Attacker Enclave

To cause a page fault, we first need to pass Step ① of the memory access flow (Figure 2.1). We achieve this by spawning a malicious *attacker enclave*, which performs the read access. We note that there is no need for the attacker enclave to be vetted by Intel, as we can execute the attack enclave in SGX debug mode. However, a naive use of an enclave is not sufficient to coerce a page fault, because the ELRANGE of the attack enclave is different than that of the victim enclave. Consequently, the validation in Step ② of Figure 2.1 fails and the access still results in abort page semantics.

3.3.4 Malicious SGX Driver

To overcome the ELRANGE check deeming the victim's memory address outside of the attacker enclave (Step ② in Figure 2.1), we use a malicious SGX driver that introduces an incorrect and malicious mapping from the *virtual* address space of the attacker's enclave to the *physical* EPC page of the victim enclave (see Figure 3.2). With the malicious mapping in place, the attacker enclave can pass the ELRANGE check when attempting to read EPC pages located in physical memory and belonging to the victim enclave.

However, as mentioned in Section 2.4, SGX prevents operating systems from maliciously manipulating the virtual to physical mapping of enclave pages by keeping the reverse mapping inside the EPCM (Figure 3.2). The EPCM contains the reverse mapping from the physical addresses in the EPC to the virtual addresses in the enclave's ELRANGE. When memory is accessed from inside an enclave, the CPU checks the OS-controlled page-table against the EPCM to verify that the OS-managed virtual to physical mapping matches the page's EPCM entry. As shown in Figure 2.1, memory is only accessed in case both mappings match, and an EPCM page fault occurs if a mismatch is detected.

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

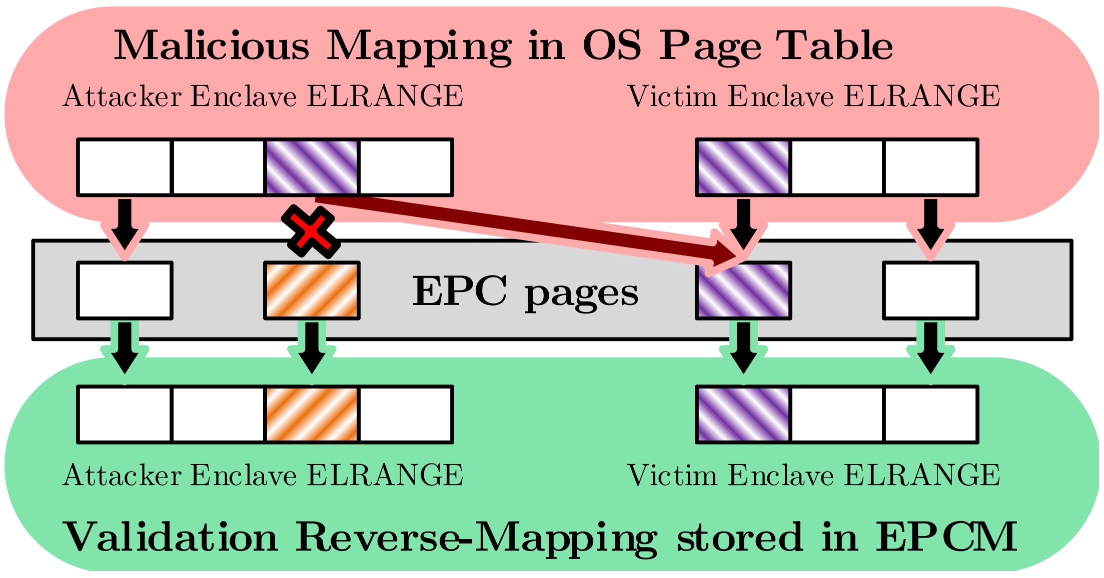

Figure 3.2: **Malicious mapping.** At creation time, the OS assigns the enclave a contiguous virtual address range (ELRANGE). The Virtual Address of an enclave page may be assigned a physical address of a page in the EPC. In SGX-ray, the attacker maps a virtual page within *its own* virtual range to the victim's physical EPC page.

3.3.5 Reading Cached Enclave Data

We begin our attack by assuming that the address space of the victim enclave contains some secret that the attacker wants to read and that this secret happens to reside in the CPU's L1 cache. Later, in Section 3.3.7, we show how to remove this assumption, allowing the attacker to read the entire address space of the victim enclave. Our attack thus proceeds as follows.

A Cross-Enclave Speculative Execution Attack. We begin our attack by setting up a malicious hosting process (Section 3.3.2) which initializes both the victim enclave and the attacker enclave (Section 3.3.3). The process then uses the malicious driver to set up a malicious mapping of the victim enclave's page that contains the data we want to read (Section 3.3.4). Finally, as explained above, in this section we assume that the data that the attacker wants to read resides in the CPU's L1 cache.

Next, our attacker enclave exploits the CPU's branch predictor in order to mount a speculative execution attack on the victim enclave. Unlike Spectre attacks [53], which exploit the branch predictor to read (and subsequently leak) data from within the same address space, our attack exploits the branch predictor to leak information across enclave boundaries while eluding the page fault issued for the illegal access. At a high level, our speculative execution attack consists of three phases, which we now describe.

Step 1: Branch Predictor Training. Consider the pseudocode presented in Figure 3.3, which is executed by the *attacker enclave* using some `dummyValue` that is provided by the malicious hosting process. During the first five iterations of the for loop in line 3, the selected address is `dummyAddress` (which is the address of the variable `dummyValue`) and it is the case that the branch in Line 7 evaluates to true. Next, as it is the case that `selectedAddress` equals `dummyAddress`, Lines 9 and 12 actually send the attacker-provided `dummyValue` through the

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

Figure 3.3: Pseudo code of the attacker enclave. The function `ct_select` outputs addressToRead when $i = 0$ and dummyAddress otherwise.

Flush+Reload cache covert channel. We note that while this does not provide any additional information to the attacker, it does train the CPU's branch predictor that the branch in line 7 typically evaluates to true.

Step 2: Attack Phase. Consider the final iteration $(i = 0)$ of the loop in line 3. As $i = 0$, the selected address in line 5 is the address from which to read in the victim enclave, as provided by the hosting process. Moreover, as line 6 flushes the value of `dummyAddress`, it is impossible to evaluate the branch in line 7 until `dummyAddress` is fetched from memory. Rather than waiting, the CPU consults the branch predictor and speculatively executes the branch in line 7, predicting the condition to be true. Next, as `selectedAddress` equals `addressToRead` (since $i = 0$), both values actually point (via the malicious mapping described in Section 3.3.4) into a physical page belonging to the victim enclave. As the data located in `addressToRead` already resides in the L1 cache, the CPU proceeds to speculatively execute line 9, setting `value` to be the value located in `addressToRead`. Finally, while `dummyAddress` is still fetched from memory, the CPU also proceeds to speculatively execute line 12, thereby leaking `value` through a cache-based Flush+Reload covert channel.

Step 3: Fault Suppression. We note that executing line 9 when $i = 0$ actually performs a memory access to `addressToRead`, which points (via the malicious memory mapping) to a physical page of the victim enclave. Next, as discussed in Section 3.3.4, SGX holds a redundant copy of relevant page table entries in the EPCM, to verify the virtual to physical mapping, as managed by the (potentially malicious) OS. Thus, as shown in Figure 2.1, executing line 9 for the case of $i = 0$ where the EPCM does not match the page table entry should have resulted in an EPCM page fault. However, recall that our attack actually executes line 9 speculatively, while waiting for `dummyAddress` to arrive from memory (after being flushed in line 6). When the value of `dummyAddress` eventually does arrive from memory, the CPU realizes the branch in line 7 was mispredicted, for the case of $i = 0$, and rolls back the execution of Lines 9—12 without emitting an EPCM page fault. However, as the cache state is not rolled back, the host process can receive the value obtained from the victim enclave, sent through the cache-based covert channel.

Relation to Meltdown and Spectre. Our attack is, in fact, a hybrid between the techniques proposed in the Spectre [53] and Meltdown [60] papers. More specifically, it uses a technique similar to Meltdown where the attacker abuses speculative execution to dereference a pointer to a privileged address space, and subsequently leaking a result through a cache-based side channel. On the other hand, our attack also uses the fault suppression technique from Spectre, as it mistrains the CPU's branch predictor regarding line 7 of Figure 3.3. While such a combination was theoretically considered in the Meltdown paper [60], to the best of our knowledge, this paper is the first to provide an explicit implementation and empirical evaluation of this technique.

3.3.6 Recovering the Leaked Data

So far we have focused on describing how our attack reads data from the victim enclave and transmits it using a cache-based covert channel. In this section, we focus on describing the receiving side of our attack, which recovers the data of the victim enclave from the cache-based covert channel. First, we notice that the code in Figure 3.3 actually transmits two values using the Flush+Reload based covert channel. Indeed, during the first five iterations of the loop in line 3 of Figure 3.3, the code sends the value of `dummyValue` (as provided by the attacker). Next, during the final iteration ($i = 0$), the code in Figure 3.3 sends the value located in `addressToRead`. Thus, to learn the value located in `addressToRead`, the attacker must somehow distinguish the transmission of `dummyValue` from the transmission of other values (presumably obtained during the last iteration of the loop in line 3). The problem is further compounded by the existence of sporadic channel noise that sometimes corrupts some of the transmissions.

At a high level, we solve both issues using an approach similar to that of [53]. That is, we transmit the value located in `addressToRead` multiple times, each time providing a different `dummyValue` to be transmitted along with it. We then aggregate the results across all the multiple transmission attempts, returning the most common value as the value located in `addressToRead`. As the correct value located in `addressToRead` remains the same while `dummyValue` is constantly changed, we expect that the most common value transmitted via the cache-based covert channel will be the value located in `addressToRead`.

More specifically, after creating the incorrect memory mapping between the attacker's enclave and the victim enclave (as described in Section 3.3.4), the attacker proceeds to execute the pseudocode presented in Figure 3.4, setting `addressToRead` to be a virtual address mapped to the victim enclave (via the incorrect mapping). At a high level, the pseudocode presented in Figure 3.4 performs the following for each try index $i = \mathsf{maxTries}, \cdots, 1$.

- **Step 1: Preparing the Covert Channel.** The attacker starts every attempt to read the data of the victim enclave by preparing a cache-based covert channel. This is achieved in line 6 of Figure 3.4 by calling the `prepare()` function of Figure 3.1.
- **Step 2: Leaking the Victim's Memory via Speculative Execution.** The attacker then mounts a speculative execution attack on the victim's enclave by invoking (line 11) the code presented in Figure 3.3, supplying it with $\mathsf{addressToRead}$ and using the try index i as the dummy value.
- **Step 3: Receiving Data via the Covert Channel.** After mounting the speculative execution

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

Figure 3.4: Pseudocode of reading a single byte.

attack, the attacker calls the receive function (line 15) of the cache-based covert channel to receive the data transmitted during the speculative execution attack. As the cache-based covert channel has been prepared, we expect the receive function (line 15) to return high scores for only two specific values: i % 256 (which is used as `dummyValue` and sent during the branch predictor training phase in Section 3.3.5) and the value located at `addressToRead` (which is sent during the attack phase in Section 3.3.5). The attacker then sums the scores for each value over all tries into the `byteScores` buffer, where `byteScores[j]` corresponds to the score of a j-valued byte.

Recovering the Byte's Value. With the above steps performed for each try index $i = \mathsf{maxTries}, \cdots, 1$, the attacker has collected statistics for each try index about which values were received via the covert channel. As mentioned above, for every try index $i = \mathsf{maxTries}, \cdots, 1$ we increase the score of `bytesScores[i]` and `bytesScores[value]` where value is the data located at `addressToRead`. Thus, after performing the above steps for all $i = \mathtt{maxTries}, \cdots, 1$, we expect that `bytesScores[value]` will equal `maxTries` while all other values in `bytesScores` are significantly lower. Thus, returning `argmax(byteScores)` successfully recovers value (line 22 of Figure 3.4).

3.3.7 Reading Uncached Enclave Data

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

Managing the Enclave Page Cache (EPC). Although SGX assumes an untrusted OS, SGX nevertheless *does* rely on the host's operating system for managing the limited space allocated for the EPC in the machine's physical memory. As explained in Section 2.4, the operating system uses the `EWB` and `ELDU` leaf instructions to securely copy enclave pages out of and back into the EPC. We observe that while decrypting and verifying an encrypted enclave page, the `ELDU` instruction loads the page's contents into the CPU's L1 cache. Crucially, `ELDU` never evicts the page from the L1 cache, leaving the page's contents cached after the instruction terminates.

Exploiting ELDU. Thus, our attack performs the following. Going over the pages of the victim enclave, the malicious SGX driver described in Section 3.3.4 first uses `EWB` to evict the page from the EPC only to immediately load it to the EPC using the `ELDU` instruction. As the `ELDU` instruction loads the page into the L1 cache and does not evict it afterwards, we can use the attack presented in Section 3.3.5 to extract its content. Finally, the entire attack process is repeated for the next page of the victim enclave.

3.3.8 Overall Attack Performance

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

Reading Specific Memory Areas. Using the setup described in Section 3.1.2, we begin our evaluation with the assumption that the attacker desires to recover the contents of a specific memory region inside the victim enclave. For this experiment, we launch our own victim enclave and filled 128 consecutive pages (512 KiB) of its memory with random data. Initially, using the attack described in Section 3.3.7, we can read all of these pages, achieving a reading speed of 1.63 KiB/sec and 56.6% accuracy.

Reducing Prefetching and Cache Noise. Next, to minimize cache pollution and prevent eviction of cache lines containing secret data, we disabled several memory prefetchers [92]. While this operation requires elevated privileges, recall that the SGX threat model assumes a malicious OS, thereby giving the attacker elevated privileges on the target machine. Lastly, to improve the attack accuracy, we gradually increased the value of `maxTries` in line 3 of Figure 3.4 if the extracted value is 0x00 or 0xFF. Employing these two optimizations improves the read accuracy to 85.68%, with a read speed of 1.58 KiB/sec.

Eliminating Noise Caused by Entering and Leaving the Enclave. Inspecting the errors in the recovered data, we identify that these typically occur in bursts of cache line size (i.e., 64 bytes). Observing that entering an enclave and exiting it are complex operations, we hypothesize that these operations generate sufficient noise in the cache to evict the victim enclave data during the call to `speculative_read` in line 11 of Figure 3.4. We thus follow the approach proposed by [71, 98] of having a thread continuously running inside the enclave calling `speculative_read`, avoiding the enclave enter and exit operations. Using this optimization we were able to successfully read 99.93% of the bytes at 0.88 KiB/Sec.

Reading the Entire Enclave Contents. To read the contents of an entire victim enclave, without knowing the specific virtual address of interest, we inspect the contents of `/proc/pid/maps` to find the physical addresses that match each of the enclave pages. While the entire range of the tested victim enclave is 16 MiB, only 4 MiB are allocated. Due to our optimization of dynamically adjusting the number of repetitions in Figure 3.4 if the extracted value is 0x00 or 0xFF, reading pages containing only zeros is significantly slower, yielding a read speed of 13.5 bytes/sec, compared with 880 bytes/sec for other pages. Overall, reading the entire enclave took 3:42 hours with 99.77% of the bytes successfully read.

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

3.4 Attacking SGX's Sealing Mechanism

The attack described in Section 3.3 is capable of breaching SGX's confidentiality guarantees, by reading the virtual address space dedicated to any SGX enclave available on the target machine. It cannot, however, breach SGX's integrity guarantees as it is unable to modify the contents of enclave memory.

In this section, we show an attack against SGX's sealing mechanism, which is a mechanism designed to provide enclaves with encrypted and authenticated persistent storage. In a nutshell, we use our attack from Section 3.3 to recover the sealing keys from within the address space of the victim enclave. We then use the recovered sealing keys to unseal the victim's persistent storage and replace its content. Finally, we use the recovered sealing keys again, this time to seal the new contents, by encrypting it and calculating the new authentication information. The victim enclave can now successfully unseal our (malicious) sealed data and, since the authentication information was correctly computed, believes the data is genuine and has not been tampered with. Before presenting our attack, we now provide further background information about the SGX sealing mechanism.

3.4.1 SGX's Sealing Mechanism

SGX provides enclaves with a mechanism for an encrypted and authenticated persistent storage. During CPU production, a randomly generated *Root Seal Key*, which is not kept in Intel's records, is fused in every SGX-enabled CPU. Using this key, the CPU can derive a sealing key, which can be used to encrypt and authenticate information from within the enclave's address space. Data that is *sealed* with this key, i.e., encrypted and authenticated, can be safely passed to the operating system for long-term storage, for example, on the computer's disk. SGX provides two types of sealing mechanisms, which we now describe (See [8, 45] for additional details).

Sealing Using the Enclave's Identity. As described in Section 2.3, each enclave has a unique field, called MRENCLAVE, which is a cryptographic hash of the contents used to initialize the enclave code as well as some additional properties. Using the values of the Root Seal Key, MRENCLAVE, and the CPU Security Version Number (SVN) an enclave can use the `EGETKEY` instruction to derive a unique sealing key for sealing data before passing it to the operating system for long term storage. Note that as a consequence of this approach, for the same Root Sealing Key (i.e., on the same CPU), different software versions of the same enclave have different sealing keys. This prohibits both data migration between different versions of enclaves as well as using these sealed keys for intra-enclave communication.

Sealing Using the Developer's Identity. An alternative option to the one discussed above is to generate the sealing key using the Root Seal Key, the SVN, and MRSIGNER (instead of MRENCLAVE). As we explain in Section 2.3, MRSIGNER is a cryptographic hash of the public RSA key of the enclave developer and is the same for all enclaves developed by the same vendor. Thus, data sealed in this way is accessible by different versions of the same enclave, as well as by different enclaves belonging to the same vendor.

3.4.2 Extracting SGX Sealing Keys

Key derivation using `EGETKEY` leaves the sealing key in the memory of the victim enclave. Thus, in principle, it is possible to read this key using our attack, described in Section 3.3 above. However, immediately after using it to encrypt or decrypt the sealed data, the implementation of SGX's sealing API erases the sealing key from memory. Hence, to extract the key we need a method for launching the attack described in Section 3.3 during the data sealing or unsealing process, before these keys are erased from the memory.

A Control Channel Attack. We time our attack using a variant of the controlled channel attack [100], inducing an Asynchronous Enclave Exit (AEX) when the enclave calls the encryption function. More specifically, we first examine the shared object file of the victim enclave and find the virtual addresses of the sealing and encryption functions inside the address space of the victim enclave. Next, we use our malicious driver (subsection 3.3.4) to evict from the EPC the page(s) containing the encryption and sealing functions. This induces a page fault and an asynchronous enclave exit whenever the victim enclave attempts to call the encryption functions. However, as the pages containing the monitored functions also contain code of other functions, the controlled channel attack will trigger an enclave exit on accesses to these other functions and does not reveal *exactly* which function caused the exit.

Determining the Precise Function Called. We note that during an AEX, the contents of all registers are saved in a dedicated State Save Area (SSA), including the instruction pointer register, which points to the next instruction to be executed by the enclave upon return from the AEX. The SSA is located in the enclave's virtual address space, and therefore its contents can be extracted using the attack from Section 3.3. To find the SSA's address, we follow the pointer to it found in a special enclave page called Thread Control Structure (TCS). While the TCS is not readable even to the enclave, it is readable to the attack described in Section 3.3.

Our attack thus proceeds as follows. Upon a page fault due to an access to the *target page* containing the code for the SGX sealing and encryption functions, we use the attack from Section 3.3 to read the contents of the TCS and SSA and recover the value of the instruction pointer. In the case that the instruction pointer points to a function other than the SGX encryption or decryption functions, we load the evicted *target page* back into the EPC, evict a *benign page* we anticipate will be accessed next, and resume normal enclave execution. On the next page fault, caused by access to the *benign page*, we evict the *target page* again.

However, if the instruction pointer points to the beginning of the encrypt or decrypt functions, we know that the victim's seal key is present in the victim's memory, at an address pointed by the `RDI` register, which is used by the compiler for passing function parameters. Our attack then proceeds to extract the contents of `RDI` from the SSA. The victim's seal key is then recovered from the address pointed by `RDI` using the attack from Section 3.3.

3.4.3 Empirical Evaluation

We empirically evaluate the attack presented in this section, using the experimental setup described in Section 3.1.2. We implemented a victim enclave which seals and unseals data. We successfully launch our attack as described in this section and extract the sealing key from within the victim enclave. Next, to validate we have the correct key, we implemented custom seal and unseal functions that operate on a seal key, instead of calling SGX key derivation instruction (`EGETKEY`). Using these functions, we can unseal the data sealed by the victim enclave as well as to seal new (malicious) data, outside the victim's enclave. Running the victim enclave again, the new data was successfully unsealed without errors.

3.5 Attacking SGX Attestation

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

While the attacks described in Section 3.3 and 3.4 are capable of violating the confidentiality of the entire address space of the victim enclave and the integrity of its sealed data, they cannot make the victim violate program semantics, designed by the enclave writer.

In this section, we show that the attack described in Section 3.3 and 3.4, which violate the confidentiality of the enclave address space and sealed inputs actually have devastating consequences for the soundness property of SGX's attestation protocol. More specifically, by mounting the attack described above on the SGX's Quoting Enclave, we are able to recover the private attestation keys, used by the target machine for proving its genuineness. With these keys at hand, we are able to construct a malicious simulator which passes attestation as if it was an SGX enclave running on a genuine Intel processor, while executing the simulator code outside of an actual enclave. As the private attestation keys are all that distinguish genuine Intel hardware from potentially malicious simulators, the remote verifying party has no way of distinguishing between the two and thus cannot trust the computation's output to be correct.

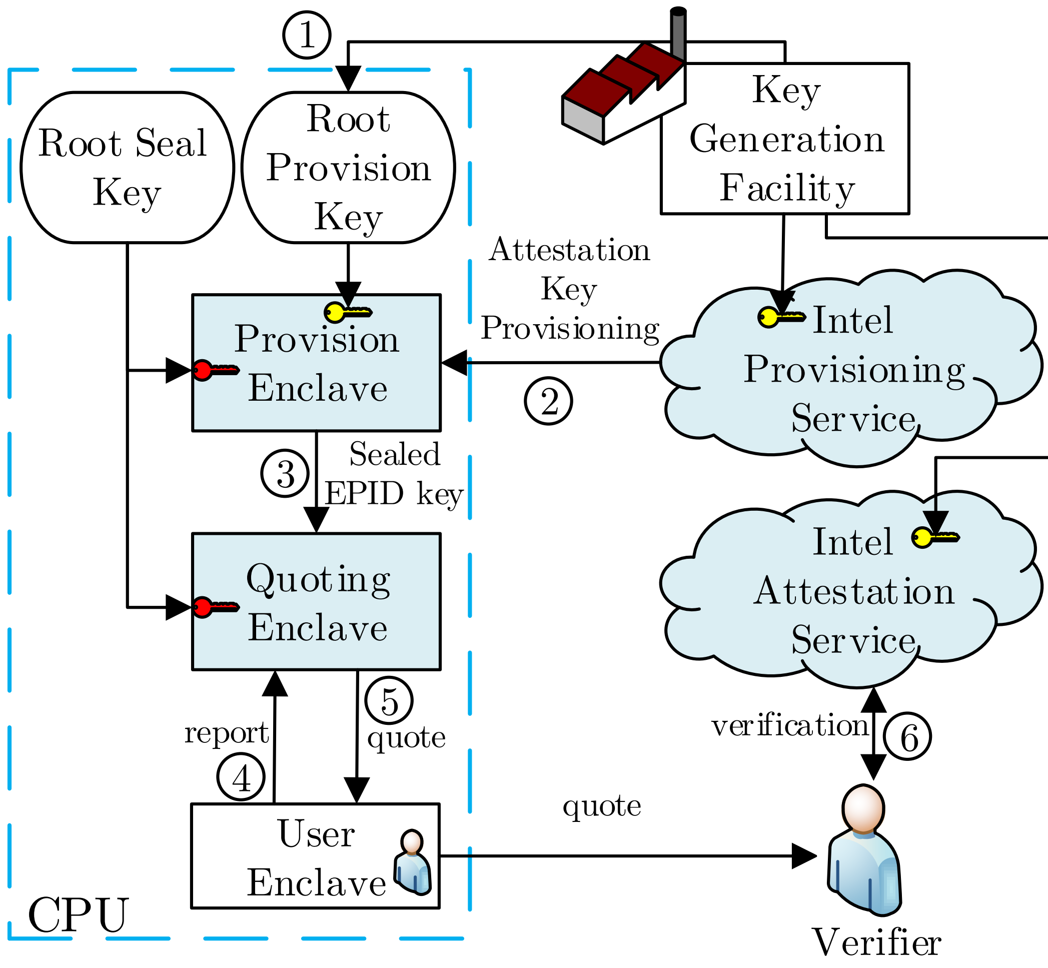

Figure 3.5: SGX's Attestation Process.

3.5.1 SGX Remote Attestation

The remote attestation process allows a *remote* verifying party to verify that a *specific* software is correctly initialized and executes within an enclave, on a genuine Intel CPU. At a high level, this is performed as follows (see [50] for an extended discussion).

In addition to the Root Sealing Key (Section 3.4.1), every SGX-enabled CPU is also shipped with a randomly-generated Root Provisioning Key (Step 1, Figure 3.5). However, unlike the Root Sealing key, Intel does retain a copy of the Root Provisioning Key, as it acts as a shared secret between Intel and every individual CPU. Next, Intel provides two special enclaves, called the *Quoting Enclave* and the *Provisioning Enclave* which are used in the attestation process.

Attestation Key Provisioning. The initialization phase of the SGX attestation protocol consists of the Provisioning Enclave contacting Intel's provisioning server, transmitting the CPU's provisioning ID, and claimed security version (SVN). As the provisioning ID uniquely identifies a specific CPU, it is only accessible to the Intel-signed Provisioning Enclave and is sent encrypted to the provisioning server under Intel's public key. After recovering the root provisioning key, corresponding to the CPU's provisioning ID, the provisioning server and Provisioning Enclave proceed to execute the *Join* phase of Intel's Enhanced Privacy ID (EPID) protocol [17], using the root provisioning key as a shared secret (Step 2, Figure 3.5).

At a high level, Intel's EPID protocol is a type of group signature that allows a CPU to sign messages (using its private signing keys) without uniquely disclosing its identity. All that an external observer (e.g., Intel) can do is to verify the signature (thereby becoming convinced that it was signed by a genuine Intel CPU belonging to the group), without being able to link it to any specific Intel CPU or to other signatures it previously signed. See [17] for additional discussion.

Sealing the EPID Key. The *Join* phase of the EPID protocol results in the Provisioning Enclave obtaining a private EPID signing key, which is not known to Intel. The Provisioning Enclave then generates a sealing key for sealing the EPID signing key, using the CPU's Root Sealing key, its SVN and the MRSIGNER value of the Provisioning Enclave. It then seals the private EPID key using this sealing key and outputs it to the OS for long term storage (Step 3, Figure 3.5). Notice that as the Provisioning Enclave is provided and signed by Intel, its MRSIGNER value is a hash of Intel's public key. Consequently, any Intel-signed enclave can unseal the CPU's private EPID key by regenerating the sealing key used to seal it. While this design feature is indeed useful, as it allows the Quoting Enclave (also signed by Intel) to unseal the private EPID key, it is also dangerous as the OS actually has an encrypted copy of the CPU's private EPID keys.

Local Attestation. When an enclave wants to prove to a remote verifier that it is running on genuine Intel hardware with a specific security version, it first needs to prove its identity to the *Quoting Enclave*, which is another special enclave provided and signed by Intel, via a processes referred to by Intel as *local attestation* [8, 45]. At a high level, this is done by having the proving enclave use the EREPORT instruction, which prepares a report containing the MRENCLAVE and MRSIGNER values of the proving enclave. The report is also signed using a key that is only accessible to the Quoting Enclave. The proving enclave then passes the report to the Quoting Enclave, which proceeds with the remote attestation process (Step 4, Figure 3.5).

Remote Attestation. Upon receiving the report from the proving enclave, the Quoting Enclave performs the remote attestation process, which we now describe. Indeed, after verifying that the report was correctly signed by the EREPORT instruction, the Quoting Enclave proceeds with unsealing the EPID private key that was originally sealed by the Provisioning Enclave. Recall that the EPID private key was sealed using a sealing key derived from the CPU's SVN version, Root Sealing Key, and the MRSIGNER value of the Provisioning Enclave. As both the Quoting Enclave and the Provisioning Enclave are signed by Intel (and thus have the same MRSIGNER value), the Quoting Enclave can regenerate this sealing key and subsequently unseal the private EPID key. Next, using the unsealed private EPID signing key, the Quoting Enclave executes the *Sign* phase of the EPID protocol and signs the report given to it by the proving enclave, creating an *attestation quote*. Finally, the Quoting Enclave returns the quote to the proving enclave, which in turn forwards it to the remote verifying party (Step 5, Figure 3.5).

Attestation Verification. After the proving enclave sends the signed quote to the remote verifying party, the remote party interacts with Intel's Attestation Server (IAS [43]) and provides it with the *quote* it obtained from the Quoting Enclave (Step 6, Figure 3.5). Next, IAS performs the *Verify* phase of the EPID protocol while ensuring that the signer's private EPID key has not been revoked by Intel. Intel's server completes the attestation by sending its response (OK, SIGNATURE_INVALID, etc.) to the remote party. The server's response also contains the quote itself and is signed with Intel's signing key, generating a signed attestation transcript which can later be verified by any party that trusts Intel's public key.

3.5.2 Extracting SGX Attestation Keys

In this Section, we describe our attack on SGX's attestation protocol. As explained above, the Quoting Enclave, which can access the EPID signing keys, will not sign a local attestation report without first verifying it. Moreover, as mentioned in Section 3.5.1 above, the operating system actually has a copy of the EPID private keys, which are sealed by a key derived from the CPU's Root Sealing Key. Our attack thus proceeds as follows.

Step 1: Recovering the Sealing Keys. Using the attack described in Section 3.4 on the Quoting Enclave, our attack recovers the sealing keys used for sealing the EPID signing keys.

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

Step 3: A Malicious Quote Enclave. Using the source code of Intel's Quoting Enclave [36], we have constructed a malicious Quoting Enclave that signs *any* local attestation report with the EPID keys, obtained in Step 2 above, without first verifying it.

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

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

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

3.5.3 Empirical Evaluation

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

Extracting EPID Keys. Using the setup from Section 3.1.2, we have successfully extracted the EPID sealing keys from a genuine SGX Quoting Enclave and subsequently unsealed the machine's private EPID keys.

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

Quote Verification. Verifying the validity of our quote, we have contacted Intel’s Attestation Server (IAS) and provided it with the above generated quote. As explained in [17, 50], the attestation server will only approve the quote if it can verify that the quote’s EPID signature is correct. Since we have correctly extracted a non-revoked EPID private signing key, the attestation server deemed our quote as correct and replied with “isvEnclaveQuoteStatus:OK”, signing its response with Intel’s private key and accompanied it with the appropriate certificate chain leading to Intel’s CA certificate.

Chapter 4

Multi-SUVM: Better Support for Multi-Enclaves

4.1 Background

The *Secure User-managed Virtual Memory (SUVM)* [71] abstraction provides a user-space mechanism for managing secure memory on top of the enclave's EPC, eliminating costly EPC hardware page faults and the associated enclave exits.

SUVM is designed as an additional level of virtual memory (VM) on top of hardware VM. The enclave program allocates memory by calling `suvm_malloc()`, which returns a special pointer we call a *secure active pointer*, or *spointer*. *S*pointers implement the same semantics as regular pointers, but they encapsulate the address translation mechanism which steers respective memory accesses to use SUVM.

SUVM implements a full-fledged paging system, with its own page table and a page cache in the EPC, and a backing store in untrusted memory of the enclave's owner process (see Figure 4.1). The SUVM page cache, *EPC++*, caches the contents of the backing store in the trusted memory. The SUVM page table maintains the mapping between EPC++ pages and the location of the cached content in the backing store. When an application accesses pages not resident in EPC++, an equivalent of a page fault occurs, but it is triggered by and handled in software. Specifically, the enclave handles the page fault by transferring the page content from the backing store into EPC++. SUVM software paging does not require exiting the enclave. SUVM mimics the behavior of the original SGX paging, maintaining privacy, integrity, and freshness of the evicted pages.

Secure backing store. A secure backing store is allocated in untrusted memory of the process that owns the enclave. The data is initialized inside the enclave, but is written back to the backing store when a page is evicted from EPC++. Upon eviction, the page is first encrypted with a random per-page nonce and signed using a random per-application key stored in the EPC. When the page is paged in, its integrity is checked to avoid replay and data manipulation attacks. The nonce and the page signature are stored in the page table inside the enclave. The encryption, signing, and validation operations use AES-GCM just like the `EWB` SGX instruction, as described in the SGX manual [3].

Periodic page eviction. In vanilla SGX, EPC page eviction is the responsibility of the SGX driver.

In Eleos, the eviction logic runs inside the enclave. Eviction may occur in three cases: (1) when EPC++ is full and a new page has to be paged in due to a page fault, (2) when an *EPC++ swapper thread*, which is periodically invoked by the untrusted runtime, removes some pages to maintain enough pages in the EPC++ free memory pool, and (3) when the swapper thread removes pages to reduce the size of EPC++ upon request of the SGX driver, e.g., when another enclave is started, as we explain in §4.3.1.

The eviction logic runs inside the enclave and is *trusted*, obviating the need to use costly *untrusted* EPC page table manipulation instructions such as `EWB`, `EBLOCK` and `ETRACK`.

4.2 Our Contribution

PRM is shared by all enclaves. Under PRM pressure, e.g., due to new enclave invocation, the SGX driver may evict part of the SUVM page cache, undermining its performance benefits. Therefore, Multi-SUVM coordinates the size of its page cache with the SGX driver to avoid thrashing, similarly to ballooning in virtual machines [93]. As a result, performance of multi-enclave execution improves by up to $3.5\times$ (§4.5).

4.3 Challenges

The SGX driver does not store any information about the currently executing enclaves, because management of such meta-structures complicates the synchronization of the driver.

In order for us to be able to serve the in-enclave Multi-SUVM controller efficiently, we decided to add our dedicated locks to the Multi-SUVM kernel component, thus maintaining the correctness of both our code and the SGX driver.

4.3.1 Multi-SUVM

Design. While designing Multi-SUVM, we had these design goals in mind:

- **Performance.** We seek to reduce the cost of running memory-demanding server applications in multi-enclave environments.
- **Small TCB.** Multi-SUVM adds relatively little code into the enclave's TCB.
- **Ease-of-use.** Multi-SUVM is transparent to application developers, so no special effort is required for usage in systems that support SUVM.
- **Security.** Multi-SUVM does not change the original SGX security guarantees, nor does it improve or weaken them.

Design overview. Figure 4.1 shows the high-level design. There are three components:

1. **Multi-SUVM controller.** the trusted runtime, which provides the SUVM inside the enclave
2. **Multi-SUVM coordinator.** an untrusted runtime running in a separate application thread to handle RPC requests and to interact with the SGX driver

3. **Multi-SUVM kernel component** the SGX driver module, which exposes the interface for coordinating SUVM memory allocation across enclaves

We now describe each component in detail.

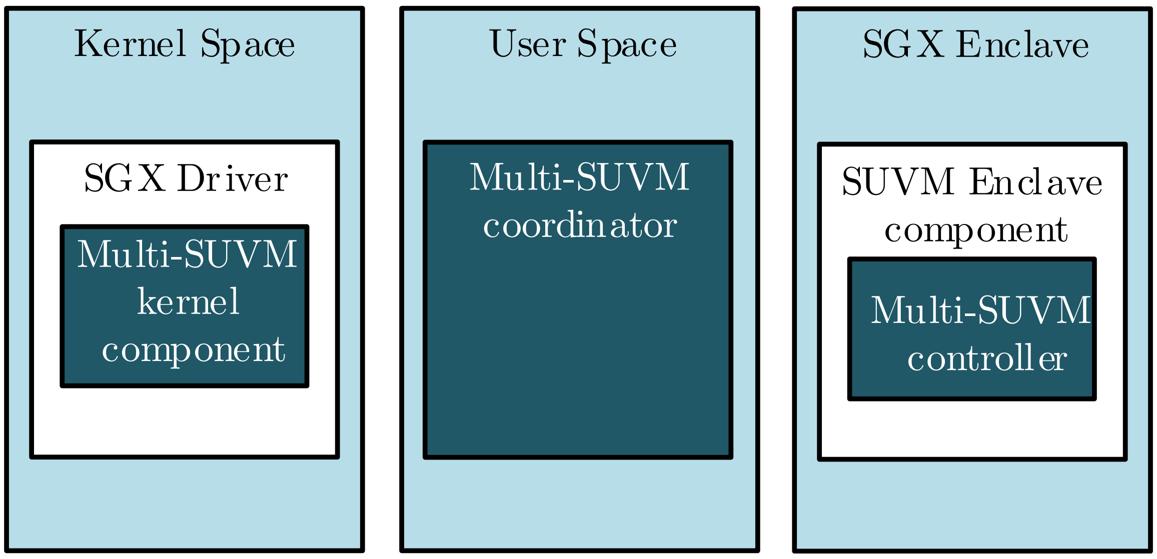

Figure 4.1: Multi-SUVM high-level design

In this work, we present Multi-enclave memory allocation - Multi-SUVM

Any EPC page, and in particular EPC++ memory, may be evicted from the EPC by the SGX driver. Unfortunately, this renders the SUVM exit-less paging mechanism useless, because accessing evicted SUVM pages would still result in a hardware page fault, and even incur additional SUVM translation penalty. This problem becomes severe with multiple enclaves, because the driver redistributes the secure memory dynamically as enclaves get invoked, without notifying the enclaves of the EPC allocation changes.

We, therefore, extend the SGX driver to coordinate EPC page eviction and allocation with the untrusted user-space runtime. Specifically, the runtime periodically queries the SGX driver to determine the secure memory space available to the enclave, adjusting the EPC++ allocation accordingly. To adjust the allocation of running enclaves the runtime invokes the SUVM swapper thread, which enters the enclave and frees or adds pages to its EPC++.

Similarity to ballooning. We note that the basic idea of collaborative management of EPC++ across enclaves is similar to that of memory ballooning [93], used by a hypervisor to manage memory allocation among virtual machines. However, whereas a hypervisor has no direct control of the OS memory consumption, the Eleos trusted runtime can directly modify the enclave's working set by evicting or adding new pages to its EPC++.

Key Differences between Multi-SUVM and ballooning. We also note that in memory ballooning for traditional virtualization, there hypervisor must work under the assumption that the VM could be malicious, and enforce the ballooning even if the VM does not cooperate.

In the SGX model, since only authorized enclaves that were signed by Intel are allowed to run in production systems, it is safe to assume there are no malicious enclaves in the system, thus allowing the OS to delegate the responsibility of actually resizing the EPC++ to the enclaves.

This way, the OS never has to restrict the enclave's EPC usage without coordinating with the enclave.

4.4 Implementation and Limitations

We implement the Multi-SUVM prototype for Linux on Skylake SGX-enabled CPUs.

SGX driver modifications. We add an `ioctl()` to query the amount of PRM available for a given enclave. Today's driver splits the PRM evenly among the enclaves, and therefore our implementation returns the number of active enclaves as a simple heuristic.

SUVM Modifications. In SUVM the size of the EPC was hard-coded and set in compile time. We moved it into a dynamic variable, and added synchronization primitives to handle resizing of the EPC++ while being accessed.

EPC++ resizing. Our implementation currently lacks working support for dynamic EPC++ resizing and thus we resort to initialization-time configuration.

Enclave cooperation. The driver code trusts the enclave code to poll the driver and free unused EPC++. We believe that this assumption is reasonable, because production enclaves must be signed by Intel, which ensures no malicious code inside them.

Blocking debug enclaves. The current SGX driver allows executing both debug and release enclaves. There is no way for us to assure that debug enclaves would implement our policy properly. This is why Multi-SUVM does not support them.

However, since we do not have the ability to write production enclaves, all the experiments in this chapter were conducted on debug enclaves.

4.5 Evaluation

We evaluate Eleos for multi-enclave environments using microbenchmarks.

Setup. We use a Dell OptiPlex 7040 machine, with Intel Skylake i7-6700 4-core CPU with 8MB LLC, 128 MB PRM (about 93MB available for applications), 16 GB RAM, and 256 GB SSD drive. The machine runs Ubuntu Linux 14.04 64-bit, Linux 4.2.0-36, and the latest Intel SGX driver [5], SDK and platform software (PSW) [4] with our modifications for performance measurements. We use GCC 4.8.4, and compile using the SGX SDK *Prerelease* configuration, which has the same performance as production enclaves [3].

Measurement methodology. Measuring in-enclave performance is a non-trivial challenge: hardware performance counters cannot be used inside the enclave because reading them (including `RDTSC`) from enclave is not supported [1]. In addition, profilers that use them rely on interrupts for sampling, which in turn induce enclave exits and distort the actual values of the counters.

Instead, we use a measurement thread outside the enclave to sample the timer. The thread is signaled from the enclave via a shared flag to measure in-enclave execution time. The measurement error is about 200 cycles, which is an order of magnitude smaller than the values we measure in the experiments.

[1] Attempts to issue `RDTSC` or `RDPMC` on our platform resulted in #UD.

Unless specified, we measure end-to-end performance, run each experiment 60 times, with the first ten invocations as warm-up, and report the average of the rest. The standard deviation is within 5% across all the experiments and is not reported.

Coordinated allocation of EPC++ across enclaves. Each experiment measures the throughput of 4K random reads for three different sizes of arrays. We test three configurations: native SGX, SUVM with correctly configured EPC++ =30MB (fitting in PRM with two enclaves), and SUVM with incorrectly configured EPC++ =50MB (which causes thrashing with two enclaves). Figure 4.2 confirms that the EPC++ size has to be adjusted in response to PRM pressure. Running two enclaves with incorrectly configured EPC++ causes both SUVM and SGX faults, which results in up to $3.4\times$ lower throughput compared to correctly configured EPC++.

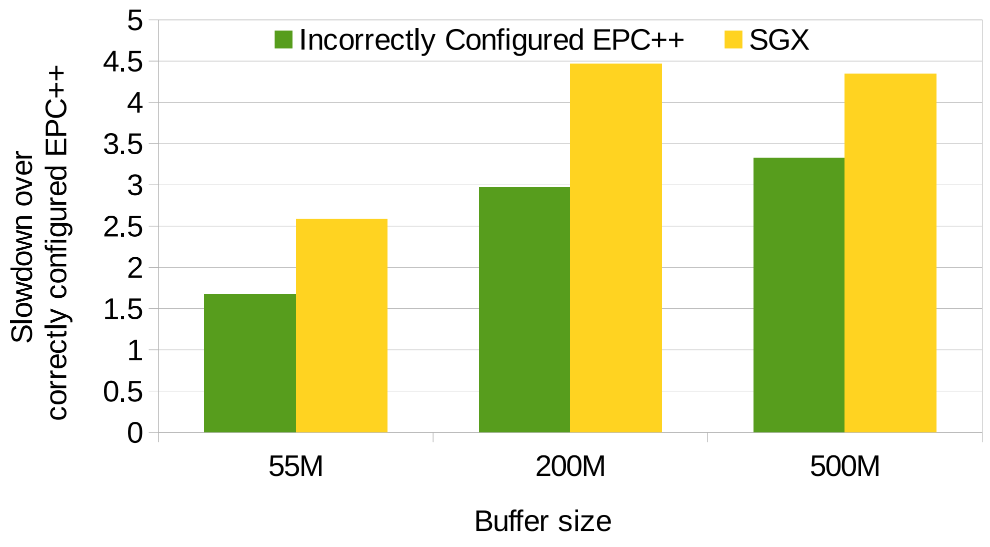

Figure 4.2: EPC++ resizing: slowdown of running two enclaves with SGX and incorrectly configured EPC++ over two enclaves with Multi-SUVM. Lower is better.

Chapter 5

Conclusions

In this work, we present novel security and performance issues regarding Intel SGX. Since SGX is implemented in hardware which is hard which is relatively hard to update and re-distribute, we tackled the question of which issues can and cannot be resolved only in software.

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

Our attack exposes the fragility of the SGX ecosystem, where a single vulnerability can result in cascading compromises that erode the security and trust properties of SGX. Intel have developed mitigations for the specific vulnerability we exploit. Nevertheless, we cannot rule out a future confidentiality breach to the SGX ecosystem that may again lead to cascading vulnerabilities that erode trust. We thus urge the community to research defense in depth for TEE technologies as well as provide inspectable open source TEE designs in the hopes of making future vulnerabilities easier to identify, mitigate, and recover. We hope that this will prevent future downfalls of a TEE ecosystem due to one breach. We also prove that mitigating our attack in software only, is impossible.

Next, we show that the performance of SGX paging can be improved with our software implementation of Multi-SUVM. Multi-SUVM is essentially different from previous work of memory ballooning by [93], because the trust model is completely different. In [93], the authors must take into design consideration a misbehaving virtual machine, even in the cost of losing performance. It is not the case with SGX, where all the production enclaves are trusted. We evaluate our system, and present a 65% and $3.4\times$ improvement in speedup and throughput, respectively.

Bibliography

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

[2] *Microsoft: Introducing Azure confidential computing*. https://azure.microsoft.com/en-us/blog/introducing-azure-confidential-computing/.

[3] Intel Software Guard Extensions Programming Reference. https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf, 2016. Accessed: 2016-10.

[4] Intel Software Guard Extensions SDK for Linux OS Developer Reference. http://download.01.org/intel-sgx/linux-1.6/docs/Intel_SGX_SDK_Developer_Reference_Linux_1.6_Open_Source.pdf, 2016. Accessed: 2016-10.

[5] Intel SGX Linux Driver. https://github.com/01org/linux-sgx-driver, 2017. Accessed: 2017-01.

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

[7] Tiago Alves and Don Felton. Trustzone: Integrated hardware and software security-enabling trusted computing in embedded systems, 2014.

[8] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. Innovative technology for CPU based attestation and sealing. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, volume 13, 2013.

[9] *Secure your keys and applications In the cloud*. Anjuna. https://www.anjuna.io/solutions/.

[10] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Daniel O'Keeffe, Mark L Stillwell, et al. SCONE: Secure Linux Containers with Intel SGX. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016.

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

[12] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding applications from an untrusted cloud with haven. *ACM Transactions on Computer Systems (TOCS)*, 33(3):8, 2015.

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

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

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

[16] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software grand exposure: SGX cache attacks are practical. In *11th USENIX Workshop on Offensive Technologies (WOOT 17)*, Vancouver, BC, 2017. USENIX Association. URL https://www.usenix.org/conference/woot17/workshop-program/presentation/brasser.

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

[18] Guoxing Chen, Wenhao Wang, Tianyu Chen, Sanchuan Chen, Yinqian Zhang, XiaoFeng Wang, Ten-Hwang Lai, and Dongdai Lin. Racing in hyperspace: Closing hyper-threading side channels on sgx with contrived data races. In *Racing in Hyperspace: Closing Hyper-Threading Side Channels on SGX with Contrived Data Races*, page 0. IEEE.

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

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

[21] Dmitry Evtyushkin, Ryan Riley, Nael Abu-Ghazaleh, and Dmitry Ponomarev. BranchScope: A new side-channel attack on directional branch predictor. In *ASPLOS*, 2018.

[22] *Fortanix - Runtime Encryption with Intel SGX*. Fortanix. https://www.fortanix.com/.

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

[24] Cesar Pereida García, Billy Bob Brumley, and Yuval Yarom. “make sure DSA signing exponentiations really are constant-time”. In *ACM Conference on Computer and Communications Security*, pages 1639–1650. ACM, 2016.

[25] Daniel Genkin, Lev Pachmanov, Itamar Pipman, Eran Tromer, and Yuval Yarom. Ecdsa key extraction from mobile devices via nonintrusive physical side channels. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1626–1638. ACM, 2016.

[26] Abel Gordon, Nadav Amit, Nadav Har’El, Muli Ben-Yehuda, Alex Landau, Assaf Schuster, and Dan Tsafrir. ELI: bare-metal performance for I/O virtualization. *ACM SIGPLAN Notices*, 47(4):411–422, 2012.

[27] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. Cache attacks on intel SGX. In *EUROSEC*, pages 2:1–2:6. ACM, 2017.

[28] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache template attacks: Automating attacks on inclusive last-level caches. In *USENIX Security Symposium*, pages 897–912. USENIX Association, 2015.

[29] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+ flush: a fast and stealthy cache attack. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 279–299. Springer, 2016.

[30] Daniel Gruss, Moritz Lipp, Michael Schwarz, Daniel Genkin, Jonas Juffinger, Sioli O’Connell, Wolfgang Schoechl, and Yuval Yarom. Another flip in the wall of Rowhammer defenses. In *IEEESP*, 2018.

[31] Shay Gueron. A memory encryption engine suitable for general purpose processors. *IACR Cryptology ePrint Archive*, 2016:204, 2016.

[32] Shay Gueron. Memory encryption for general-purpose processors. *IEEE Security & Privacy*, 14(6):54–62, 2016.

[33] David Gullasch, Endre Bangerter, and Stephan Krenn. Cache games – bringing access-based cache attacks on AES to practice. In *IEEESP*, pages 490–505, May 2011.

[34] Matthew Hoekstra, Reshma Lal, Pradeep Pappachan, Vinay Phegade, and Juan Del Cuvillo. Using innovative instructions to create trustworthy software solutions. In *HASP@ ISCA*, page 11, 2013.

[35] Tyler Hunt, Zhiting Zhu, Yuanzhong Xu, Simon Peter, and Emmett Witchel. Ryoan: A distributed sandbox for untrusted computation on secret data. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association.

[36] *Intel(R) Software Guard Extensions for Linux* OS*. Intel, . https://github.com/01org/linux-sgx.

[37] *Software Developer Manual*. Intel, . http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf.

[38] *Intel Analysis of Speculative Execution Side Channels*. Intel, . https://newsroom.intel.com/wp-content/uploads/sites/11/2018/01/Intel-Analysis-of-Speculative-Execution-Side-Channels.pdf.

[39] *A More Protected Cloud Environment: IBM Announces Cloud Data Guard Featuring Intel SGX*. Intel, . https://itpeernetwork.intel.com/ibm-cloud-data-guard-intel-sgx/.

[40] *Retpoline: A Branch Target Injection Mitigation*. Intel, . https://software.intel.com/sites/default/files/managed/1d/46/Retpoline-A-Branch-Target-Injection-Mitigation.pdf.

[41] *Intel SGX and Side-Channels*. Intel, . https://software.intel.com/en-us/articles/intel-sgx-and-side-channels.

[42] *Intel Software Guard Extensions*. Intel, . https://software.intel.com/sites/default/files/332680-001.pdf.

[43] *Software Guard Extensions: EPID Provisioning and Attestation Services*. Intel, . https://software.intel.com/sites/default/files/managed/3d/c8/IAS_1_0_API_spec_1_1_Final.pdf.

[44] *Intel Software Guard Extensions*. Intel, . https://software.intel.com/en-us/sgx.

[45] *Intel Software Guard Extensions SDK for Linux OS*. Intel, . https://01.org/sites/default/files/documentation/intel_sgx_sdk_developer_reference_for_linux_os_pdf.pdf.

[46] *Intel Software Guard Extensions (SGX) SW Development Guidance for Potential Edger8r Generated Code Side Channel Exploits*. Intel, March 2018. https://software.intel.com/sites/default/files/managed/e1/ec/180309_SGX_SDK_Developer_Guidance_Edger8r.pdf.

[47] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. S$a: a shared cache attack that works across cores and defies vm sandboxing–and its application to aes. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 591–604. IEEE, 2015.

[48] Gorka Irazoqui Apecechea, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. Wait a minute! A fast, cross-VM attack on AES. In *RAID*, volume 8688 of *Lecture Notes in Computer Science*, pages 299–319. Springer, 2014.

[49] Yeongjin Jang, Jaehyuk Lee, Sangho Lee, and Taesoo Kim. Sgx-bomb: Locking down the processor via Rowhammer attack. In *SysTEX'17*, pages 5:1–5:6. ACM, 2017.

[50] Simon Johnson, Vinnie Scarlata, Carlos Rozas, Ernie Brickell, and Frank Mckeen. Intel® software guard extensions: Epid provisioning and attestation services. *White Paper*, 1:1–10, 2016.

[51] Sangman Kim, Seonggu Huh, Xinya Zhang, Yige Hu, Amir Wated, Emmett Witchel, and Mark Silberstein. GPUnet: Networking abstractions for GPU programs. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 201–216, 2014.

[52] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of dram disturbance errors. In *ACM SIGARCH Computer Architecture News*, volume 42, pages 361–372. IEEE Press, 2014.

[53] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. *arXiv preprint arXiv:1801.01203*, 2018.

[54] Robert Krahn, Bohdan Trach, Anjo Vahldiek-Oberwagner, Thomas Knauth, Pramod Bhatotia, and Christof Fetzer. Pesos: Policy enhanced secure object store. In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys)*, 2018.

[55] Dmitrii Kuvaiskii, Oleksii Oleksenko, Sergei Arnautov, Bohdan Trach, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. Sgxbounds: Memory safety for shielded execution. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 205–221. ACM, 2017.

[56] Jaehyuk Lee, Jinsoo Jang, Yeongjin Jang, Nohyun Kwak, Yeseul Choi, Changho Choi, Taesoo Kim, Marcus Peinado, and Brent B Kang. Hacking in darkness: Return-oriented programming against secure enclaves. In *USENIX Security*, pages 523–539, 2017.

[57] Jaehyuk Lee, Jinsoo Jang, Yeongjin Jang, Nohyun Kwak, Yeseul Choi, Changho Choi, Taesoo Kim, Marcus Peinado, and Brent ByungHoon Kang. Hacking in darkness: Return-oriented programming against secure enclaves. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 523–539, Vancouver, BC, 2017. USENIX Association. ISBN 978-1-931971-40-9. URL https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/lee-jaehyuk.

[58] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring fine-grained control flow inside sgx enclaves with branch shadowing. *arXiv preprint arXiv:1611.06952*, 2016.

[59] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. Armageddon: Cache attacks on mobile devices. In *USENIX Security Symposium*, pages 549–564, 2016.

[60] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown. *arXiv preprint arXiv:1801.01207*, 2018.

[61] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R Savagaonkar. Innovative instructions and software model for isolated execution. In *HASP@ ISCA*, page 10, 2013.

[62] Frank McKeen, Ilya Alexandrovich, Ittai Anati, Dror Caspi, Simon Johnson, Rebekah Leslie-Hurd, and Carlos Rozas. Intel Software Guard Extensions (Intel SGX) Support for Dynamic Memory Management Inside an Enclave. In *Proceedings of the Hardware and Architectural Support for Security and Privacy 2016*. ACM, 2016.

[63] Dirk Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal*, 2014(239), 2014.

[64] Ralph C. Merkle. A digital signature based on a conventional encryption function. In *CRYPTO*, pages 369–378, 1987.

[65] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. Cachezoom: How SGX amplifies the power of cache attacks. In *CHES*, volume 10529 of *Lecture Notes in Computer Science*, pages 69–90. Springer, 2017.

[66] Ahmad Moghimi, Thomas Eisenbarth, and Berk Sunar. Memjam: A false dependency attack against constant-time crypto implementations in SGX. In *CT-RSA*, volume 10808 of *Lecture Notes in Computer Science*, pages 21–44. Springer, 2018.

[67] Dan O'Keeffe, Divya Muthukumaran, Pierre-Louis Aublin, Florian Kelbert, Christian Priebe, Josh Lind, Huanzhou Zhu, and Peter Pietzuch. Spectre attack against SGX enclave. https://github.com/lsds/spectre-attack-sgx, 2018.

[68] Dan O'Keeffe, Divya Muthukumaran, Pierre-Louis Aublin, Florian Kelbert, Christian Priebe, Josh Lind, Huanzhou Zhu, and Peter Pietzuch. SGXSpectre. https://github.com/lsds/spectre-attack-sgx, 2018.

[69] Oleksi Oleksenko, Bohdan Trach, Robert Krahn, Andre Martin, Mark Silberstein, and Christof Fetzer. VARYS: Protecting sgx enclaves from practical side-channel attacks. In *USENIX Annual Technical Conference*, 2018.

[70] Yossef Oren, Vasileios P Kemerlis, Simha Sethumadhavan, and Angelos D Keromytis. The spy in the sandbox: Practical cache attacks in javascript and their implications. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 1406–1418. ACM, 2015.

[71] Meni Orenbach, Pavel Lifshits, Marina Minkin, and Mark Silberstein. Eleos: Exitless os services for sgx enclaves. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 238–253. ACM, 2017.

[72] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of aes. In *Cryptographers' Track at the RSA Conference*, pages 1–20. Springer, 2006.

[73] Colin Percival. Cache missing for fun and profit, 2005.

[74] Peter Pessl, Leon Groot Bruinderink, and Yuval Yarom. To BLISS-B or not to be: Attacking strongswan's implementation of post-quantum signatures. In *CCS*, pages 1843–1855. ACM, 2017.

[75] Christian Priebe, Kapil Vaswani, and Manuel Costa. Enclavedb - a secure database using sgx. IEEE, May 2018. URL `https://www.microsoft.com/en-us/research/publication/enclavedb-a-secure-database-using-sgx/`.

[76] Daniel J Scales and Kourosh Gharachorloo. Design and performance of the Shasta distributed shared memory protocol. In *Proceedings of the 11th international conference on Supercomputing*, pages 245–252. ACM, 1997.

[77] Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. Vc3: Trustworthy data analytics in the cloud using sgx. In *2015 IEEE Symposium on Security and Privacy*, pages 38–54. IEEE, 2015.

[78] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Malware guard extension: Using SGX to conceal cache attacks. In *DIMVA*, pages 3–24, 2017.

[79] Jaebaek Seo, Byounyoung Lee, Seongmin Kim, Ming-Wei Shih, Insik Shin, Dongsu Han, and Taesoo Kim. Sgx-shield: Enabling address space layout randomization for sgx programs. In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA*, 2017.

[80] Hovav Shacham. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *ACM Conference on Computer and Communications Security*, pages 552–561. ACM, 2007.

[81] Sagi Shahar, Shai Bergman, and Mark Silberstein. Activepointers: a case for software address translation on GPUs. In *Proceedings of the 43rd International Symposium on Computer Architecture*, pages 596–608. IEEE Press, 2016.

[82] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. T-sgx: Eradicating controlled-channel attacks against enclave programs. In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA*, 2017.

[83] Shweta Shinde, Zheng Leong Chua, Viswesh Narayanan, and Prateek Saxena. Preventing page faults from telling your secrets. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, pages 317–328. ACM, 2016.

[84] Shweta Shinde, Dat Le Tien, Shruti Tople, and Prateek Saxena. PANOPLY: Low-TCB Linux Applications with SGX Enclaves. *National University of Singapore, Tech. Rep*, 2016.

[85] Mark Silberstein, Bryan Ford, Idit Keidar, and Emmett Witchel. GPUfs: integrating a file system with GPUs. In *ACM SIGPLAN Notices*, volume 48, pages 485–498. ACM, 2013.

[86] Livio Soares and Michael Stumm. FlexSC: flexible system call scheduling with exception-less system calls. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, pages 33–46. USENIX Association, 2010.

[87] Florian Tramer, Fan Zhang, Huang Lin, Jean-Pierre Hubaux, Ari Juels, and Elaine Shi. Sealed-glass proofs: Using transparent enclaves to prove and sell knowledge. In *Security and Privacy (EuroS&P), 2017 IEEE European Symposium on*, pages 19–34. IEEE, 2017.

[88] Chia-Che Tsai, Kumar Saurabh Arora, Nehal Bandi, Bhushan Jain, William Jannen, Jitin John, Harry A Kalodner, Vrushali Kulkarni, Daniela Oliveira, and Donald E Porter. Cooperation and security isolation of library OSes for multi-process applications. In *Proceedings of the Ninth European Conference on Computer Systems*. ACM, 2014.

[89] Chia-Che Tsai, Donald E Porter, and Mona Vij. Graphene-sgx: A practical library os for unmodified applications on sgx. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, page 8, 2017.

[90] Yukiyasu Tsunoo, Etsuko Tsujihara, Kazuhiko Minematsu, and Hiroshi Hiyauchi. Cryptanalysis of block ciphers implemented on computers with cache. In *International Symposium on Information Theory and Its Applications*, Xi'an, CN, October 2002.

[91] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution. In *Proceedings of the 26th USENIX Security Symposium*. USENIX Association, 2017.

[92] Vish Viswanathan. Disclosure of H/W prefetcher control on some Intel processors. https://software.intel.com/en-us/articles/disclosure-of-hw-prefetcher-control-on-some-intel-processors, September 2014.

[93] Carl A Waldspurger. Memory resource management in VMware ESX server. *ACM SIGOPS Operating Systems Review*, 36(SI):181–194, 2002.

[94] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A. Gunter. Leaky cauldron on the dark land: Understanding memory side-channel hazards in SGX. In *CCS*, pages 2421–2434. ACM, 2017.

[95] Zhenghong Wang and Ruby B Lee. Covert and side channels due to processor architecture. In *Computer Security Applications Conference, 2006. ACSAC'06. 22nd Annual*, pages 473–482. IEEE, 2006.

[96] Nico Weichbrodt, Anil Kurmus, Peter Pietzuch, and Rüdiger Kapitza. Asyncshock: Exploiting synchronisation bugs in intel sgx enclaves. In *European Symposium on Research in Computer Security*, pages 440–457. Springer, 2016.

[97] Samuel Weiser, Raphael Spreitzer, and Lukas Bodner. Single trace attack against RSA key generation in Intel SGX SSL. In *AsiaCCS*, 2018.

[98] Ofir Weisse, Valeria Bertacco, and Todd Austin. Regaining lost cycles with hotcalls: A fast interface for sgx secure enclaves. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pages 81–93. ACM, 2017.

[99] Bin Cedric Xing, Mark Shanahan, and Rebekah Leslie-Hurd. Intel Software Guard Extensions (Intel SGX) Software Support for Dynamic Memory Allocation inside an Enclave. In *Proceedings of the Hardware and Architectural Support for Security and Privacy 2016*. ACM, 2016.

[100] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 640–656. IEEE, 2015.

[101] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: a high resolution, low noise, L3 cache side-channel attack. In *USENIX Security*, pages 719–732, 2014.

[102] Yuval Yarom, Daniel Genkin, and Nadia Heninger. Cachebleed: a timing attack on openssl constant-time rsa. *Journal of Cryptographic Engineering*, 7(2):99–112, 2017.

[103] Fan Zhang, Ethan Cecchetti, Kyle Croman, Ari Juels, and Elaine Shi. Town crier: An authenticated data feed for smart contracts. In *Proceedings of the 2016 aCM sIGSAC conference on computer and communications security*, pages 270–282. ACM, 2016.

[104] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-tenant side-channel attacks in paas clouds. In *ACM Conference on Computer and Communications Security*, pages 990–1003. ACM, 2014.

[105] Wenting Zheng, Ankur Dave, Jethro Beekman, Raluca Ada Popa, Joseph Gonzalez, and Ion Stoica. Opaque: An oblivious and encrypted distributed analytics platform. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, Boston, MA, 2017. USENIX Association. URL https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/zheng.

ל SGX ישנה תכונה חשובה מאוד של עדות מרוחקת, שבעזרתה מובלעות של SGX יכולות להוכיח למשתמש המרוחק שהן מריצות את הקוד הנכון ושהן רצות על חומרה אמיתית של אינטל ולא על סימולטור.

בעבודה זו, אנו ניסינו להבין טוב יותר ולשפר את הביצועים ואת האבטחה של SGX. מכיוון שה SGX ממומש בחומרה, אנחנו ניתחנו וניסינו להבין אילו שיפורים ניתן לבצע בתוכנה, ואילו שיפורים דורשים שינויים יקרים בחומרה. ההפרדה הזאת היא מאוד חשובה, מכיוו שהפצת עדכוני חומרה לכל המעבדים היא פעולה הרבה יותר יקרה ומורכבת מאשר עדכון תוכנה.

בחלק הראשון של עבודה זו, אנו מציגים את Foreshadow, מתקפה מיקרו ארכיטקטונית על מובלעות SGX, שמאפשרת למערכת ההפעלה לקרוא את הזיכרון הפרטי של המובלעת, בתנאי והמידע נמצא במטמון. בהמשך העבודה אנחנו מציגים כיצד בעזרת Foreshadow, אנחנו יכולים לשבור את כל מערכת העדות המרוחקת, ושבעזרת פריצת מכונה אחת שלנו, אנחנו מסוגלים למוטט את כל האמון במנגנון העדות המרוחקת של SGX.

לבסוף, אנחנו מוכיחים שאת המתקפה הזאת לא ניתן למגר בתוכנה, ושעדכוני חומרה הם הכרחיים לצורך כך. אנחנו מוכיחים זאת בכך שאנחנו מנצלים את מנגנון הדפדוף כדי לפענח מידע מוצפן של המובלעת ולטעון אותו לתוך המטמון. מכיוון שקוד של המובלעת לא צריך לרוץ לשם כך, אין אף גישה תוכנתית שתמנע את המתקפה שלנו.

בחלק השני של עבודה זו, אנו מתמקדים בשיפור האבטחה של מובלעות של SGX. בעבודות קודמות ראינו שמכיוון שכל פעולת דפדוף מלווה ביציאה איטית מהמובלעת, ניתן לשפר את פקודות הדפדוף של SGX באופן משמעותי, באמצעות מנגנון תוכנה שנקרא SUVM.

המנגנון מחקה את מנגנון הדפדוף החומרתי של SGX, אבל מכיוון שהוא מתבצע בתוכנה, ולא בחומרה, נחסכות היציאות היקרות מהמובלעת, מה שמוביל לשיפור ביצועים משמעותי.

בעבודה הזאת, אנחנו מציגים את Multi-SUVM- מערכת תוכנתית שמרחיבה את SUVM לעבוד בסביבות מרובות-מובלעות. אנחנו גילינו ש SUVM לא יעיל בסביבות מרובות-מובלעות, מכיוון שלא קיים שם מנגנון שיתוף משאבים. ב Multi-SUVM, אנחנו מציגים שינוי למערכת הפעלה שמנהלת את המובלעות שמוביל לשיפור בביצועים. אנחנו מצליחים לשפר את זמן הריצה ב 65%, ואת הספיקה של המערכת פי 3.4.

תקציר

SGX היא טכנולוגית חומרה חדשה המרחיבה את סט הפקודות של מעבדי אינטל. בכדי לשפר את האבטחה של תוכניות הרצות על גבי המעבד, היא מאפשרת לאפליקציות לייצר מובלעות המהוות סביבות חישוב מבודדות ובטוחות בתוך המעבד.׳. הזיכרון הפרטי ותוכן האוגרים של המובלעת נסתרים מהתוכנית המארחת אותה ומכל התוכניות האחרות, כולל מערכת ההפעלה. למובלעת, לעומת זאת יש גישה לזיכרון של האפליקציה המארחת אותה.

ברמת הארכיטקטורה הנחשפת לכותב האפליקציה, למרות שמובלעות של SGX נכתבות בשפת המכונה הוותיקה של המעבד של אינטל, ישנן פקודות דפדוף חדשות ומיוחדות שנועדו לנהל את הזיכרון של המובלעת.

ברמת החומרה, המובלעת משתמשת בזיכרון הדפים של המובלעת. זהו זיכרון שמוקצה בעת עליית המערכת. כל הזיכרון הפרטי של המובלעת נשמר בתוך מטמון הדפים של המובלעת. זהו זיכרון המוצפן בעזרת מפתח שנגיש אך ורק למעבד, כך שאפילו תוקף בעל גישה פיזית למכונה, שיכול לקרוא את כל הזיכרון, אינו מסוגל לקרוא או לשנות את הסודות בתוך מטמון הדפים של המובלעת. כאשר המובלעת רוצה לקרוא מידע מזיכרון הדפים של המובלעת, המעבד מפענח את המידע בעבורה, וטוען אותו בצורה קריאה אל תוך המטמון של המעבד.

כאשר המובלעות במערכת דורשות יותר זיכרון מאשר זמין בתוך מטמון הדפים של המובלעת, מערכת ההפעלה יכולה להשתמש בפקודות מיוחדות על מנת לדפדף דפים של המובלעת אל תוך הזיכרון הלא מוצפן, וחזרה. כדי לתמוך הפעולה הזאת מבלי לתת למערכת ההפעלה אפשרות לקרוא את הזיכרון הפרטי של המובלעת, הארכיטקטורה מספקת פקודות שמצפינות ומפענחות דפים עבור מערכת ההפעלה בעזרת מפתחות הנגישים רק לחומרה.

תודות

אני רוצה להביע את הערכתי המיוחדת ולהודות לפרופסור מרק זילברשטיין, היית מנחה מעולה עבורי. אתה אחד האנשים הכי חכמים שאני מכירה ואני רוצה להודות לך על כך שעודדת את המחקר שלי ואפשרת לי לגדול כחוקרת. התרומה שלך למחקר שלי לא תסולא בפז.

אני רוצה להודות לבוריס ואן סוסין, איתי דברן, לינה מאדלוג', בעז שטרנפלד ולכל הדיירים של קומה 4 בטאוב על התמיכה הרבה שהם סיפקו לי במהלך לימודי. בנוסף אני רוצה להודות לטניה ברוכמן, וסיליס דימיסטאס, שי ברגמן ולכל שאר קבוצת ה ACG.

במהלך לימודי, התמזל מזלי לעבוד בשיתוף פעולה ולמוד מהרבה אנשים נפלאים: פבל ליפשיץ, מני אורנבך, אופיר וויס, דניאל גנקין, תומאס וניש, יובל ירום, יו ואן בולק, פרנק פיסנס וראול סטראקס.

לבסוף, אני רוצה להוסות לפקולטה למדעי במחשב, לפקולטה להנדסת חשמל ול TCE בטכניון, על שסיפקו לי סביבה מחקרית נהדרת שנהניתי ממנה רבות.

אני מודה לטכניון, למרכז הסייבר של הטכניון על שם Hiroshi Fujiwara ולמערך הסייבר הלאומי על התמיכה הנדיבה במשך השתלמותי.

המחקר בוצע בפקולטה למדעי המחשב בטכניון בהנחייתו של פרופ. מרק זילברשטיין (מהפקולטה להנדסת חשמל).

חלק מהתוצאות בחיבור זה פורסמו כמאמרים מאת המחברת ושותפיה למחקר בכנסים במהלך תקופת מחקר המאסטר של המחברת, אשר גרסאותיהם העדכניות ביותר הינן:

1. Van Bulck, J., Minkin, M., Weisse, O., Genkin, D., Kasikci, B., Piessens, F., Silberstein, M., Wenisch, T.F., Yarom, Y. and Strackx, R., 2018. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In 27th USENIX Security Symposium (USENIX Security 18), pp. 991-1008.
2. Orenbach, M., Lifshits, P., Minkin, M. and Silberstein, M., 2017, April. Eleos: ExitLess OS services for SGX enclaves. In Proceedings of the Twelfth European Conference on Computer Systems (EuroSys 17), pp. 238-253.

שיפור ביצועים ואבטחה של SGX

חיבור על מחקר

לשם מילוי חלקי של הדרישות לקבלת התואר
מגיסטר למדעים במדעי המחשב

מרינה מינקין

הוגש לסנט הטכניון – מכון טכנולוגי לישראל
טבת התשע"ט חיפה דצמבר 2018

שיפור ביצועים ואבטחה של SGX

מרינה מינקין