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

Jo Van Bulck, *imec-DistriNet, KU Leuven;* **Marina Minkin**, *Technion;* **Ofir Weisse, Daniel Genkin, and Baris Kasikci**, *University of Michigan;* **Frank Piessens**, *imec-DistriNet, KU Leuven;* **Mark Silberstein**, *Technion;* **Thomas F. Wenisch**, *University of Michigan;* **Yuval Yarom**, *University of Adelaide and Data61;* **Raoul Strackx**, *imec-DistriNet, KU Leuven*

https://www.usenix.org/conference/usenixsecurity18/presentation/bulck

This paper is included in the Proceedings of the 27th USENIX Security Symposium.

August 15–17, 2018 • Baltimore, MD, USA

ISBN 978-1-939133-04-5

Open access to the Proceedings of the 27th USENIX Security Symposium is sponsored by USENIX.

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

Jo Van Bulck[1], Marina Minkin[2], Ofir Weisse[3], Daniel Genkin[3], Baris Kasikci[3], Frank Piessens[1], Mark Silberstein[2], Thomas F. Wenisch[3], Yuval Yarom[4], and Raoul Strackx[1]

[1]*imec-DistriNet, KU Leuven,* [2]*Technion,* [3]*University of Michigan,* [4]*University of Adelaide and Data61*

Abstract

Trusted execution environments, and particularly the Software Guard eXtensions (SGX) included in recent Intel x86 processors, gained significant traction in recent years. A long track of research papers, and increasingly also real-world industry applications, take advantage of the strong hardware-enforced confidentiality and integrity guarantees provided by Intel SGX. Ultimately, enclaved execution holds the compelling potential of securely offloading sensitive computations to untrusted remote platforms.

We present Foreshadow, a practical software-only microarchitectural attack that decisively dismantles the security objectives of current SGX implementations. Crucially, unlike previous SGX attacks, we do not make any assumptions on the victim enclave's code and do not necessarily require kernel-level access. At its core, Foreshadow abuses a speculative execution bug in modern Intel processors, on top of which we develop a novel exploitation methodology to reliably leak plaintext enclave secrets from the CPU cache. We demonstrate our attacks by extracting full cryptographic keys from Intel's vetted architectural enclaves, and validate their correctness by launching rogue production enclaves and forging arbitrary local and remote attestation responses. The extracted remote attestation keys affect millions of devices.

1 Introduction

It becomes inherently difficult to place trust in modern, widely used operating systems and applications whose sizes can easily reach millions of lines of code, and where a single vulnerability can often lead to a complete collapse of all security guarantees. In response to these challenges, recent research [11,41,48] and industry efforts [1,2,35,43] developed Trusted Execution Environments (TEEs) that feature an alternative, non-hierarchical protection model for isolated application compartments called *enclaves*. TEEs enforce the confidentiality and integrity of mutually distrusting enclaves with a minimal Trusted Computing Base (TCB) that includes only the processor package and microcode. Enclave-private CPU and memory state is exclusively accessible to the code running inside it, and remains explicitly out of reach of all other enclaves and software running at any privilege level, including a potentially malicious operating system and/or hypervisor. Besides strong memory isolation, TEEs typically offer an *attestation* primitive that allows local or remote stakeholders to cryptographically verify at runtime that a specific enclave has been loaded on a genuine (and hence presumed to be secure) TEE processor.

With the announcement of Intel's Software Guard eXtensions (SGX) [2, 27, 43] in 2013, hardware-enforced TEE isolation and attestation guarantees are now available on off-the-shelf x86 processors. In light of the strong security guarantees promised by Intel SGX, industry actors are increasingly adopting this technology in a wide variety of applications featuring secure execution on adversary-controlled machines. Open Whisper Systems [50] relies on SGX for privacy-friendly contact discovery in its Signal network. Both Microsoft and IBM recently announced support for SGX in their cloud infrastructure. Various off-the-shelf Blu-ray players and initially also the 4K Netflix client furthermore use SGX to enforce Digital Rights Management (DRM) for high-resolution video streams. Emerging cryptocurrencies [44] and innovative blockchain technologies [25] rely even more critically on the correctness of Intel SGX.

Our Contribution. This paper shows, however, that current SGX implementations cannot meet their security objectives. We present the Foreshadow attack, which leverages a speculative execution bug in recent Intel x86 processors to reliably leak plaintext enclave secrets from the CPU cache. At its core, Foreshadow abuses the same processor vulnerability as the recently announced Meltdown [40] attack, i.e., a delicate race condition in the CPU's access control logic that allows an attacker to use

the results of unauthorized memory accesses in transient out-of-order instructions before they are rolled back. Importantly, however, whereas Meltdown targets traditional hierarchical protection domains, Foreshadow considers a very different attacker model where the adversary's goal is not to read kernel memory from user space, but to compromise state-of-the-art *intra-address space* enclave protection domains that are not covered by recently deployed kernel page table isolation defenses [19]. We explain how Foreshadow necessitates a novel exploitation methodology, and we show that our basic attack can be entirely mounted by an unprivileged adversary without root access to the victim machine. Given SGX's unique privileged attacker model, however, we additionally contribute a set of *optional* kernel-level optimization techniques to further reduce noise for root adversaries. Our findings have far-reaching consequences for the security model pursued by Intel SGX in that, in the absence of a microcode patch, current SGX processors cannot guarantee the confidentiality of enclaved data nor attest the integrity of enclaved execution, including for Intel's own architectural enclaves. Moreover, despite SGX's ambition to defend against strong kernel-level adversaries, present SGX processors cannot even safeguard enclave secrets in the presence of unprivileged user space attackers.

All previously known attacks against Intel SGX rely on application-specific information leakage from either side-channels [30, 39, 45, 51, 57, 58, 60] or software vulnerabilities [38, 59]. It was generally believed that well-written enclaves could prevent information leakage by adhering to good coding practices, such as never branching on secrets, prompting Intel to state that "in general, these research papers do not demonstrate anything new or unexpected about the Intel SGX architecture. Preventing side channel attacks is a matter for the enclave developer" [33]. Foreshadow defeats this argument, however, as it relies solely on elementary Intel x86 CPU behavior and does *not* exploit any software vulnerability, or even require knowledge of the victim enclave's source code. We demonstrate this point by being the first to actually extract long-term platform launch and attestation keys from Intel's critical and thoroughly vetted architectural launch and quoting enclaves, decisively dismantling SGX's security objectives. In summary, our contributions are:

- We advance the understanding of Meltdown-type transient execution CPU vulnerabilities by showing that they also apply to intra-address space isolation and SGX's non-terminating abort page semantics.
- We present novel exploitation methodologies that allow an unprivileged software-only attacker to reliably extract enclave secrets residing in either protected memory locations or CPU registers.
- We evaluate the effectiveness and bandwidth of the Foreshadow attack through controlled experiments.
- We extract full cryptographic keys from Intel's architectural enclaves, and demonstrate how to (*i*) bypass enclave launch control; and (*ii*) forge local and remote attestations to completely break confidentiality plus integrity guarantees for remote computations.

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

Intel confirmed that microcode patches are underway and should be deployed concurrently to the public release of our results. As of this writing, however, we have not been provided with substantial technical information about these mitigations. We discuss defense strategies in Section 6, and provide further guidelines on the impact of our findings at `https://foreshadowattack.eu/`.

Disclosure. Foreshadow was independently and concurrently discovered by two teams. The KU Leuven authors discovered the vulnerability, independently developed the attack, and first notified Intel on January 3, 2018. Their work was done independently from and concurrently to other recent x86 speculative execution vulnerabilities, notably Meltdown and Spectre [36, 40]. The authors from Technion, University of Michigan, and the University of Adelaide independently discovered and reported the vulnerability to Intel during the embargo period on January 23, 2018.

2 Background

We first overview Intel SGX [2, 10, 27, 43] and refine the attacker model. Thereafter, we introduce the relevant parts of the x86 microarchitecture, and discuss previous research results on speculative execution vulnerabilities.

2.1 Intel SGX

Memory Isolation. SGX enclaves live in the virtual address space of a conventional user mode process, but

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

SGX furthermore protects enclaves against motivated adversaries that exploit Rowhammer DRAM bugs, or resort to physical cold boot attacks. A hardware-level Memory Encryption Engine (MEE) [21] transparently safeguards the integrity, confidentiality, and freshness of enclaved code and data while residing outside of the processor package. That is, any access to main memory is first authenticated and decrypted before being brought as plaintext into the CPU cache.

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

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

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

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

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

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

2.2 Attack Model and Objectives

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

Crucially, in contrast to all previously published SGX side-channel attacks [17,39,45,51,57,58,60] and existing Spectre-style speculative execution attacks [7,49] against SGX enclaves, Foreshadow does *not* require any side-channel vulnerabilities, code gadgets, or even knowledge of the victim enclave's code. In particular, our attack is immune to all currently proposed side-channel mitigations for SGX [8,9,18,52–54], as well as countermeasures for speculative execution attacks [31,32]. In fact, as long as secrets reside in the enclave's address space, our attack does not even require the victim enclave's execution.

Breaking SGX Confidentiality. The Intel SGX documentation unequivocally states that "enclave memory cannot be read or written from outside the enclave regardless of current privilege level and CPU mode (ring3/user-mode, ring0/kernel- mode, SMM, VMM, or another enclave)" [28]. As Foreshadow compromises confidentiality of production enclave memory, this security objective of Intel SGX is clearly broken.

Our basic attack requires enclave secrets to be residing in the L1 data cache. We show how unprivileged adversaries can preemptively or concurrently extract secrets as they are brought into the L1 data cache when executing the victim enclave. For root adversaries, we furthermore contribute an innovative technique that leverages SGX's paging instructions to prefetch arbitrary enclave memory into the L1 data cache without even requiring the victim enclave's cooperation. When combined with a state-of-the-art enclave execution control framework, such as SGX-Step [57], our root attack can essentially dump the entire memory and register contents of a victim enclave at any point in its execution.

Breaking SGX Sealing and Attestation. The SGX design allows enclaves to "request a secure assertion from the platform of the enclave's identity [and] bind enclave ephemeral data to the assertion" [2]. While we cannot break integrity of enclaved data directly, we do leverage Foreshadow to extract enclave sealing and report keys. The former compromises the confidentiality and integrity of sealed secrets directly, whereas the latter can be used to forge false local attestation reports. Our attack on Intel's trusted quoting enclave for remote attestation furthermore completely collapses confidentiality plus integrity guarantees for remote computations and secret provisioning.

2.3 Microarchitectural x86 Organization

Instruction Pipeline. For a complex instruction set, such as Intel x86 [10,27], individual instructions are first split into smaller micro-operations (μops) during the decode stage. Micro-operation decoding simplifies processor design: only actual μops need to be implemented in hardware, not the entire rich instruction set. In addition it enables hardware vendors to patch processors when a flaw is found. In case of Intel SGX, this may lead to an increased CPU security version number.

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

Out-of-Order and Speculative Execution. As an important optimization technique, the processor may choose to not execute sequential micro-operations as provided by the in-order instruction stream. Instead, μops are executed *out-of-order*, as soon as the required execution unit plus any source operands become available. Following Tomasulo's algorithm [55], μops are dynamically scheduled, e.g., using reservation stations, and await the availability of their input operands before they are executed. After completing μop execution, intermediate results are buffered, e.g., in a Reorder Buffer (ROB), and committed to architectural state only upon instruction retirement.

To yield correct architectural behavior, however, the processor should ensure that μops are retired according to the intended in-order instruction stream. Out-of-order execution therefore necessitates a roll-back mechanism that flushes the pipeline and ROB to discard uncommitted μop results. Generally, such *speculatively* executed μops are to be dropped by the CPU in two different scenarios. First, after realizing an execution path has been mispredicted by the branch predictor, the processor flushes μop results from the incorrect path and starts executing the correct execution path. Second, hardware exceptions and interrupts are guaranteed to be "always taken in the 'in-order' instruction stream" [27], which implies that all transient μop results originating from out-of-order instructions following the faulting instruction should be rolled-back as well.

CPU Cache Organization. To speed up repeated code and data memory accesses, modern Intel processors [27] feature a dedicated L1 and L2 cache per physical CPU (shared among logical HyperThreading cores), plus a single last-level L3 cache shared among all physical cores. The unit of cache organization is called a *cache line* and measures 64 bytes. In multi-way, set-associative caches, a cache line is located by first using the lower bits of the (physical) memory address to locate the corresponding *cache set*, and thereafter using a tag to uniquely identify the desired cache line within that set.

Since CPU caches introduce a measurable timing difference for DRAM memory accesses, they have been studied extensively in side-channel analysis research [16].

2.4 Transient Execution Attacks

The aforementioned in-order instruction retirement ensures functional correctness: the CPU's architectural state (memory and register file contents) shall be consistent with the intended program behavior. Nevertheless, the CPU's *microarchitectural* state (e.g., internal caches) can still be affected by μops that were speculatively executed and afterwards discarded. Recent concurrent research [15, 24, 36, 40, 42] on *transient execution attacks* shows how an adversary can abuse such subtle microarchitectural side-effects to breach memory isolation barriers.

A first type of Spectre [36] attacks exploit the CPU's branch prediction machinery to trick a victim protection domain into speculatively executing instructions out of its intended execution path. By "poisoning" the shared branch predictor resource, an attacker is able to steer the victim program's execution into transient instruction sequences that dereference memory locations the victim is authorized to access but the attacker not. A second type of attacks, including Meltdown [40] and Foreshadow, exploit a more crucial flaw in modern Intel processors. Namely, that there exists a small time window in which the results of unauthorized memory accesses are available to the out-of-order execution, before the processor issues a fault and rolls back any speculatively executed μops. As such, Meltdown represents a critical race condition inside the CPU, which enables an attacker to transiently execute instructions that access unauthorized memory locations.

Essentially, transient execution allows an attacker to perform secret-dependent computations whose direct architectural effects are later discarded. In order to actually extract secrets, a "covert channel" should therefore be established to bring information into the architectural state. That is, the transient instructions have to deliberately alter the shared microarchitectural state so as to transfer/leak secret values. The CPU cache constitutes one such reliable covert channel; Meltdown-type vulnerabilities have therefore also been dubbed "rogue data cache loads" [24].

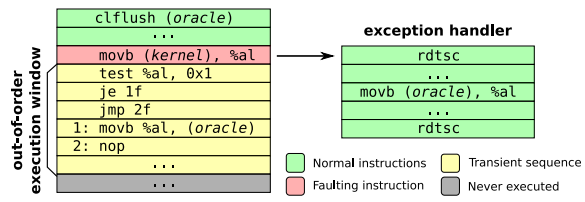


Figure 1: Rogue data cache loads can be leveraged to leak sensitive data from more privileged security layers.

Figure 1 illustrates a toy example scenario where an attacker extracts one bit of information across privilege levels. In the first step, an attacker attempts to read data from a more privileged protection layer, eventually causing a fault to be issued and the execution of an exception handler. But, a small attack window exists where attackers can execute instructions based on the actual data read, and encode secrets in the CPU cache. The example uses a reliable FLUSH+RELOAD [61] covert channel, where the transient instruction sequence loads a predetermined "oracle" memory location into the cache, dependent on the least significant bit of the kernel data just read. When the processor catches up and eventually issues the fault, a previously registered user-level exception handler is called. This marks the beginning of the second step, where the adversary receives the secret bit by carefully measuring the amount of time it takes to reload the oracle memory slot.

3 The Foreshadow Attack

In contrast to Meltdown [40], Foreshadow targets enclaves operating within an untrusted context. As such, adversaries have many more possibilities to execute the attack. However, as explained below and further explored in Appendix A, targeting enclaved execution also presents substantial challenges, for SGX's modified memory access and non-terminating fault semantics reflect extensive microarchitectural changes that affect transient execution.

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

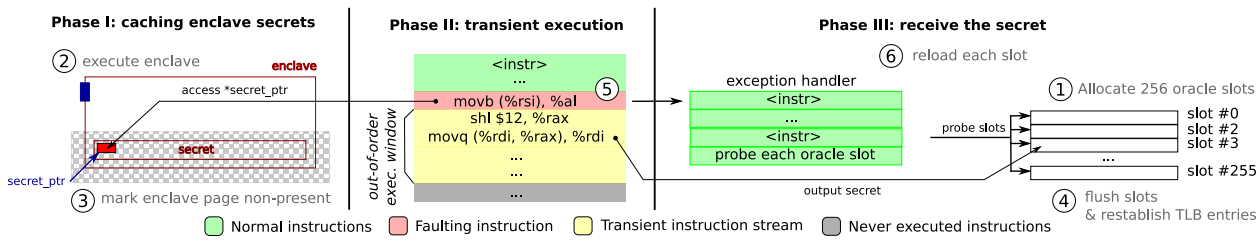


Figure 2: Basic overview of the Foreshadow attack to extract a single byte from an SGX enclave.

3.1 The Basic Foreshadow Attack

The basic Foreshadow attack extracts a single byte from an SGX enclave in three distinct phases, visualized in Fig. 2. As part of the attack preparation, the untrusted enclave host application should first allocate an "oracle buffer" ① of 256 slots, each measuring 4 KiB in size (in order to avoid false positives from unintentionally activating the processor's cache line prefetcher [26, 40]). In Phase I of the attack, plaintext enclave data is brought into the CPU cache. Next, Phase II dereferences the enclave secret and speculatively executes the transient instruction sequence, which loads a secret-dependent oracle buffer entry into the cache. Finally, Phase III acts as the receiving end of the FLUSH+RELOAD covert channel and reloads the oracle buffer slots to establish the secret byte.

Phase I: Caching Enclave Secrets. In contrast to previous research [15, 24, 40] on exploiting Meltdown-type vulnerabilities to read kernel memory, we found consistently that enclave secrets never reach the transient out-of-order execution stage in Phase II when they are not already residing in the L1 cache. A prerequisite for any successful transient extraction therefore is to bring enclave secrets into the L1 cache. As we noticed that the untrusted application cannot simply `prefetch` [20] enclave memory directly, the first phase of the basic Foreshadow attack executes the victim enclave ② in order to cache plaintext secrets. For now, we assume the secret we wish to extract resides in the L1 cache after the enclaved execution. We elaborate on this assumption in Sections 3.3 and 3.4 for interrupt-driven and HyperThreading-based attacks respectively. Section 3.5 thereafter explains how root adversaries can bring secrets in the L1 cache without even executing the victim enclave.

Note that, while Meltdown has reportedly been successfully applied to read uncached kernel data directly from DRAM, Intel's official analysis report clarifies that "on some implementations such a speculative operation will only pass data on to subsequent operations if the data is resident in the lowest level data cache (L1)" [29]. We suspect that SGX's modified memory access semantics bring about fundamental differences at the microarchitectural level, such that the CPU's access control logic does not pass the results of unauthorized enclave memory loads unless they can be served from the L1 cache. Intel confirmed this hypothesis, officially referring to Foreshadow as an "L1 Terminal Fault" attack. We furthermore provide experimental evidence in Appendix A, showing that Foreshadow can indeed transiently compute on kernel data in the L2 cache, but decisively *not* on enclave secrets residing in the L2 cache.

Regarding Intel SGX's hardware-level memory encryption [21], it should be noted that the MEE security perimeter encompasses the processor package, including the entire CPU cache hierarchy. That is, enclave secrets always reside as plaintext inside the caches and are only encrypted/decrypted as they move to/from DRAM. Practically, this means that transient instructions can in principle compute on plaintext enclave secrets as long as they are cached. As such, the MEE hardware unit does not impose any fundamental limitations on the Foreshadow attack, and is assuredly not the cause for the observation that we cannot read enclave secrets residing in the L2 cache.

Phase II: Transient Execution. In the second phase, we dereference `secret_ptr` and execute the transient instruction sequence. In contrast to previous transient execution attacks [15, 24, 29, 40] that result in a page fault after accessing kernel space, however, dereferencing unauthorized enclave memory does *not* produce a page fault. Instead, abort page semantics [28] apply and the data read is silently replaced with the dummy value -1. As such, in the absence of an exception, the race condition does not apply and any (transient) instructions following the rogue data fetch will never see the actual enclave secret, but rather the abort page value.

Foreshadow overcomes this challenge by taking advantage of previous research results on page table-based enclaved execution attacks [58, 60]. Intel SGX implements an additional layer of hardware-enforced isolation on top of the legacy page table-based virtual memory protection mechanism. That is, abort page semantics apply only *after* the legacy page table permission check succeeded without issuing a page fault.[1] This property effectively

[1] Alternatively, as a result of SGX's additional EPCM checks [27], rogue virtual-to-physical mappings also result in page fault behavior

enables the unprivileged host process to impose strictly more restrictive permissions on enclave memory. In our running example, we proceed by revoking ③ all access permissions to the enclave page we wish to read:

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

We verified that the above `mprotect` system call simply clears the "present" bit in the corresponding page table entry, such that any access to this page now (eventually) leads to a fault. This observation yields an important side result, in that previous Meltdown attacks [15, 24, 29, 40] focussed exclusively on reading kernel memory pages. Intel's analysis of speculative execution vulnerabilities hence explicitly mentions that rogue data cache loads only apply "to regions of memory designated supervisor-only by the page tables; not memory designated as not present" [29]. This is not in agreement with our findings.

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

Finally, we execute ⑤ the transient instruction sequence displayed in Listing 1. We provide a line-per-line translation to the equivalent C code in Listing 2. When called with a pointer to the oracle buffer and `secret_ptr`, the secret value is read at Line 5. As we made sure to mark the enclave page as not present, SGX's abort page semantics no longer apply and a fault will eventually be issued. However, the transient instructions at Lines 6–7 will still be executed and compute the secret-dependent location of a slot v in the oracle buffer before fetching it from memory.

Phase III: Receiving the Secret. Finally when the processor determines that it should not have speculatively executed the transient instructions, uncommitted register changes are discarded and a page fault is issued. After the fault is caught by the operating system, the attacker's user-level exception handler is called. Here, she carefully measures ⑥ the timings to reload each oracle slot to establish the secret enclave byte. If the transient instruction

after passing the address translation process. We experimentally verified that such faults can be successfully exploited by an attacker enclave that transiently dereferences a victim enclave's pages via a malicious memory mapping. Future mitigations (Section 6) should therefore decisively also take this microarchitectural exploitation path into account.

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

Listing 1: x86 assembly.

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

Listing 2: C code.

sequence reached the execution at Line 7, the oracle slot at the secret index now resides in the CPU cache and will experience a significantly shorter access time.

3.2 Reading Full Cache Lines

The basic Foreshadow attack of the previous section leaks sensitive information while only leveraging the capabilities of a conventional user space attacker. But as SGX also aims to defend against kernel-level attackers, this section presents various optimization techniques, some of which assume root access (when indicated). In Section 4 we will show that these optimizations increase the bandwidth plus reliability of our attack, enabling us to extract complete cache lines from a single enclaved execution.

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

Page Aliasing (Root). When untrusted code accesses enclave memory, abort page semantics apply and secrets do not reach the transient execution. The basic Foreshadow attack avoids this behavior by revoking all access rights from the enclave page through the `mprotect` interface. However, as enclaved execution also abides by page table-based access restrictions [58, 60], these privi-

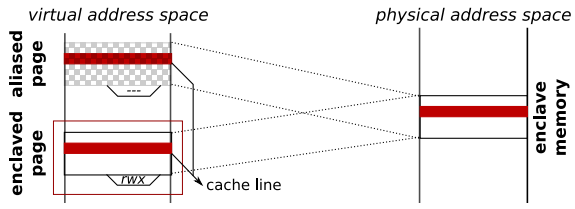


Figure 3: The physical enclave secret is mapped to an inaccessible virtual address for transient dereference.

leges can only be revoked *after* the enclave call returned. Unfortunately, we found that the `mprotect` system call exerts pressure on the processor's cache and may cause the enclave secret to be evicted from the L1 cache.

We propose an inventive "page aliasing" technique to avoid `mprotect` cache pollution for root adversaries. Figure 3 shows how our malicious kernel driver establishes an additional virtual-to-physical mapping for the physical enclave location holding the secret. As caches on modern Intel CPUs are physically tagged [27], memory accesses via the original or alias pages end up in the exact same cache lines. That is, the aliased page behaves similarly to the original enclaved page; only an additional page table walk is required for address translation. We evade abort page semantics for the alias page in the same way as in the basic Foreshadow attack, by calling `mprotect` to clear the present bit in the page table. Importantly, however, we can now issue `mprotect` once in Phase I of the attack, *before* entering the enclave. For the aliased memory mapping is never referenced by the enclave itself.

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

We leverage the Transactional Synchronization eXtensions (TSX) included in modern Intel processors [27] to silently handle exceptions *within* the unprivileged attacker process. Previous research [9, 40, 53, 54] has exploited an interesting feature of Intel TSX. Namely, a page fault during transactional execution immediately calls the user-level transaction abort handler, without first signalling the fault to the operating system. We abuse this property to avoid unnecessary kernel context switches between Phases II and III of the Foreshadow attack by wrapping the entire transient instruction sequence of Listing 1 in a TSX transaction. While the transaction's write set is discarded, we did not notice any difference in the read set. That is, accessed oracle slots remain in the L1 cache.

Note that, while readily available on many processors, TSX is by no means the only fault suppression mechanism that attackers could leverage. Alternatively, as previously suggested [24,40], the instruction dereferencing the secret could also be speculatively executed itself, behind a high-latency mispredicted branch. As a true hybrid between Spectre [36] and Meltdown [40], such a technique would deliberately mistrain the CPU's branch predictor to ensure that none of the instructions in Listing 1 are committed to the architecture, and hence no faults are raised.

Keeping Secrets Warm (Root). Context switches to kernel space are not the only sources of cache pollution. In Phase III of the attack the access time to each oracle slot is carefully measured. As each slot is loaded into the cache, enclave secrets might get evicted from the L1 cache. To make matters worse, oracle slots are placed 4 KiB apart to avoid false positives from the cache line prefetcher [26]. All 256 oracle slots thus share the same L1 cache index and map to the same cache set.

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

Isolating Cores (Root). We found overall system load to be another significant source of cache pollution. Intel architectures typically feature an inclusive cache hierarchy: data residing in the L1 cache shall also be present in the L2 and L3 caches [27]. Unfortunately, maintaining this invariant may lead to unexpected cache evictions. When an enclaved cache line is evicted from the shared last-level L3 cache by another resource-intensive process for instance, the processor is forced to also evict the enclave secret from the L1 cache. Likewise, since L1 and L2 caches are shared among logical processors, cache activity on one core might unintentionally evict enclave secrets on its sibling core.

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

Dealing with Zero Bias. Consistent with concurrent work on Meltdown-type vulnerabilities [15, 24, 40, 42], we found that the processor zeroes out the result of unauthorized memory reads upon encountering an exception. When this nulling happens before the transient out-of-order instructions in Phase II can operate on the real secret, the attacker loses the internal race condition from the CPU's access control logic. This will show up as reading an all-zeroes value in Phase III. To counteract this zero bias, Foreshadow retries the transient execution Phase II multiple times when receiving `0x00` in Phase III, before decisively concluding the secret byte was indeed zero.

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

3.3 Preemptively Extracting Secrets

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

To improve Foreshadow's temporal resolution, we therefore asynchronously exit the enclave after a secret in memory was brought into the L1 cache, and before it is later overwritten/evicted. We first explain how root adversaries can combine Foreshadow with the state-of-the-art SGX-Step [57] enclave execution control framework to achieve a maximal temporal resolution: memory operands leak after every single instruction. Next, we re-iterate that even unprivileged adversaries can pause enclaves at a coarser-grained 4 KiB page fault granularity [59, 60] through the `mprotect` system call interface. Using this capability, we contribute a novel technique that allows unprivileged Foreshadow attackers to reliably inspect private CPU register contents of a preempted victim enclave.

Single-Stepping Enclaved Execution (Root). SGX prohibits obvious interference with production enclaves. Specifically, the processor ignores advanced x86 debug features, such as hardware breakpoints or the single-step trap flag (RFLAGS.TF) [27]. We therefore rely on the recently published open-source SGX-Step [57] framework to interrupt the victim enclave instruction per instruction.

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

Dumping Enclaved CPU Registers. Section 2.1 explained how SGX securely stores the interrupted enclave's register contents in a preallocated SSA frame as part of the AEX microcode procedure. By targeting SSA enclave memory, a Foreshadow attacker can thus extract private CPU register contents. For this to work, however, the SSA frame data of interest should reside in the processor's L1 cache. The entire SSA frame measures multiple cache lines, with the general purpose register area alone already occupying 144 bytes (2.25 cache lines). These SSA cache lines could be unintentionally evicted as part of the kernel context switches needed to handle interrupts, or during Foreshadow's transient extraction Phases II and III.

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

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

3.4 Concurrently Extracting Secrets

In modern Intel processors with HyperThreading technology, the L1 cache is shared among multiple logical processors [27]. This property has recently been abused to mount stealthy SGX PRIME+PROBE L1 cache side-

channel attacks entirely from a co-resident logical processor, without interrupting the victim enclave [6, 17, 51].

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

This approach has considerable disadvantages as compared to the above interrupt-driven attack variants. Specifically, we found that the bandwidth for concurrently extracting secrets is severely restricted, since each Foreshadow round needs 256 time-consuming FLUSH+RELOAD measurements in order to transfer one byte from the microarchitectural state (Phase II) to the architectural state (Phase III). As the enclave now continues to execute during the measurement process, secrets are more likely to be overwritten or evicted before being read by the attacker. Nonetheless, this stealthy Foreshadow attack variant should decidedly be taken into account when considering possible defense strategies in Section 6.

3.5 Reading Uncached Secrets

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

Managing the Enclave Page Cache (EPC). The SGX design [27, 43] explicitly relies on untrusted system software for oversubscribing the limited protected physical memory EPC resource. For this, untrusted operating systems can make use of the privileged `ewb` and `eldu` SGX instructions that respectively copy encrypted and integrity-protected 4 KiB enclave pages out of, and back into EPC.

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

Dumping the Entire Enclave Contents (Root). We proceed as follows to extract the entire victim memory space. Going over all enclave pages (e.g., by inspecting `/proc/pid/maps`), our malicious kernel driver first uses `ewb` to evict the page from the EPC, only to immediately load it back using the `eldu` instruction. As `eldu` loads the page into the L1 cache and does not evict it afterwards, the basic Foreshadow attack described in Section 3.1 can reliably extract its content. Finally, the attack process is repeated for the next page of the victim enclave.

The above `eldu` technique dumps the entire address space of a victim enclave without requiring its cooperation. Since the initial memory contents is known to the adversary at enclave creation time, however, secrets are typically generated or brought in at runtime (e.g., through sealing or remote secret provisioning). As such, in practice, the victim should still be executed at least once, and the attacker could rely on a single-stepping primitive, such as SGX-Step [57], to precisely pause the enclave when it contains secrets, and before they are erased again.

Crucially, however, our `eldu` technique allows to extract secrets that are never brought into the L1 cache by the enclave code itself. As further discussed in Section 6, this attacker capability effectively rules out software-only mitigation strategies that force data to be directly stored in memory while deliberately evading the CPU cache hierarchy. For instance by relying on explicit non-temporal write `movnti` instructions [5, 27].

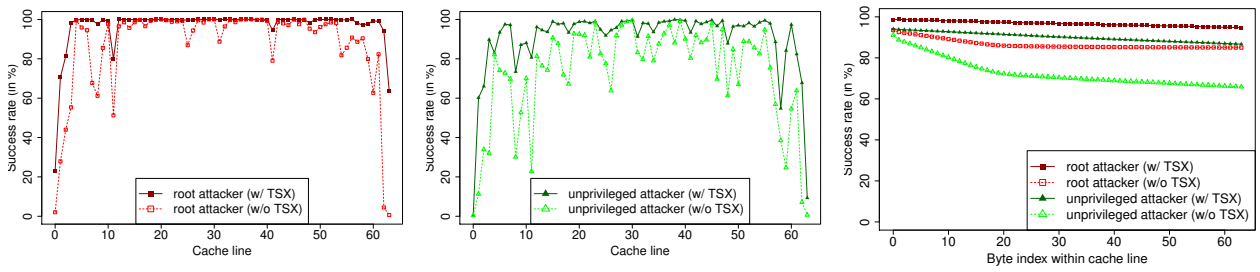
4 Microbenchmark Evaluation

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

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

Experimental Setup. For benchmarks, we consider the capabilities of both root and unprivileged attackers, conformant to our threat model in Section 2.2. The *root adversary* has full access to the targeted system. She for example aims to attack DRM technology enforced by an enclave running on her own device. This enables her to use all the attack optimization techniques described in Section 3.2. In addition, she may reduce cache pollution by pinning the victim thread to as specific logical core and offloading peripheral device interrupts to another core.

The *unprivileged adversary*, on the other hand, is much more constrained and represents an attacker targeting a remote server. She gained code execution on the device, and targets an enclave running in the same address space,



(a) Root attacker cache line extraction. (b) Unprivileged cache line extraction. (c) Intra-cache line degradation.

Figure 4: Success rates of the Foreshadow attack per cache line (4a and 4b) and per byte within a cache line (4c).

but did not manage to gain kernel-level privileges. Some attack optimizations, such as page aliasing or isolating workloads, can therefore not be applied.

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

Success Rates. Figure 4a displays the success rate for each cache line in the root attacker model. Overall, we reached an outstanding median success rate of 99.92% (with TSX). As not every SGX-capable machine supports TSX, we executed the same benchmark without relying on TSX features. This resulted in a moderate median success rate drop of 2.59 percentage points (97.32%).

Interestingly, the cache lines storing data at the beginning/end of the targeted page (i.e., cache lines `#0` and `#63`) manifest a distinctly lower average success rate: respectively 23.25/2.03% and 63.78/0.63% with and without TSX. We attribute this effect to unintended L1 cache line evictions from (*i*) the remaining enclaved execution after loading the secret into the cache (e.g., `eexit`); and (*ii*) our own attack measurement code (e.g., probing of the oracle buffer in Phase III). Specifically, upon closer inspection, we found that recent interrupt-driven SGX cache attacks [23, 46] explicitly report similar lowered success rates for the first and last cache lines, attributed to asynchronous enclave exit and kernel context switches. Note that we consider the increased cache pressure on the first/last cache lines only a nonessential limitation of our current attack framework, however, and decisively *not* an avenue to defend against improved Foreshadow attacks.

Figure 4b displays the result of the same benchmark for an unprivileged attacker. As expected, the median success rate drops reasonably to 96.82% and 81.14% with and without TSX respectively. While these success rates are somewhat lower, they distinctly show that even much more restrained user-level adversaries can successfully attack SGX enclaves with an impressive success rate.

It is crucial for the Foreshadow attack to succeed that the cache line holding the secret remains in the L1 cache. We found that the likelihood of inadvertently evicting secrets from the L1 cache increases with each byte extracted within a cache line. Figure 4c quantifies this *intra-cache line degradation* behavior. For the root adversary, the probability of successfully extracting the first byte within a cache line is 98.61%. By the time the last last byte of the cache line is extracted, however, the success rate has degraded to 94.75%. Especially the use of TSX shows to play a large role here. An unprivileged TSX attacker can limit intra-cache line degradation from 94.05% to 86.68%. This outperforms even all other optimization mechanisms for the root adversary without TSX (93.53% - 84.99%).

5 Attacking Intel Architectural Enclaves

While SGX is largely realized in hardware and microcode, Intel implemented certain critical functionality in software through dedicated "architectural enclaves". These enclaves are part of the TCB, and were written by experts with detailed knowledge of the security architecture. No obvious security flaws [38, 59] have ever been found, and Intel's architectural enclaves additionally implement various defense in-depth mechanisms. For example, even though private memory should never leak from enclaves, sensitive data gets overwritten as soon as possible.

To the best of our knowledge, we are the first to present full key extraction attacks against Intel's vetted architectural enclaves. To date only one subtle side-channel vulnerability [12] has been identified in Intel's quoting

enclave, which only affects secondary privacy concerns and assuredly does not invalidate remote attestation guarantees. This shows that Foreshadow is substantially more powerful than previous enclaved execution attacks that rely on either side-channels or memory safety bugs.

Note that, for maximum reliability, both our attacks against Intel's architectural launch and quoting enclaves assume the root adversary model, and apply all of the optimization techniques described in Section 3.2. Since our final exploits do *not* need to resort to the single-stepping or `eldu` prefetching root-only techniques of Sections 3.3 and 3.5, however, we expect they could be further improved to run entirely with user space privileges.

5.1 Attacking the Intel Launch Enclave

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

As with local attestation (Section 2.1), the security of this scheme ultimately relies on a processor-level secret accessible to both LE and `einit`. We refer to this secret as the platform *launch key*. The `einit` instruction derives the 128-bit launch key to verify the correctness of the provided EINITTOKEN, and takes care to only initialize enclaves whose identities and attributes match the ones in the token. In order to bootstrap initialization for the LE itself, Intel's MRSIGNER value is hard-coded in the processor and used by `einit` to skip the EINITTOKEN check and grant access to the launch key. This ensures that only an Intel-signed LE can invoke `egetkey` to derive the launch key needed to compute valid MACs.

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

Attack and Exploitation. Our goal is to extract a full 128-bit launch key from a single LE execution. This is necessary, for each `egetkey` derivation (Section 2.1) includes a random 256-bit KEYID, which is securely generated inside the enclave, such that each LE invocation

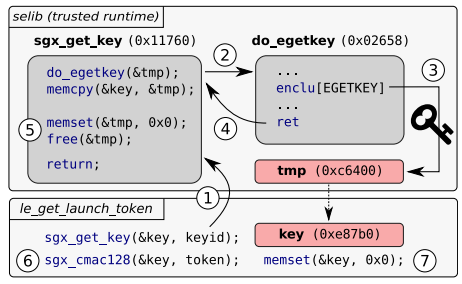


Figure 5: Key derivation in the SGX Launch Enclave.

uses a different launch key. We can therefore *not* correlate partial key recoveries from repeated launch enclave executions to extract a full key, as is common practice in side-channel research [6, 17, 39, 46, 51].

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

An attacker can get hold of the launch key by targeting either the short-lived `tmp` buffer, or the longer-lived `key` buffer. Our exploit targets the more challenging `tmp` buffer to demonstrate Foreshadow's strength in combination with state-of-the-art enclave execution control frameworks [57, 60]. In the exploratory (offline) phase of the attack, we single-step LE and dump register content (see Section 3.3) so as to easily establish the deterministic `tmp` address, plus any code locations of interest.[3] Next, in the online phase of the attack, we interrupt the victim enclave between steps ③ and ④ above, and instruct Foreshadow to extract the cache line containing the 128-bit key. We rely on page fault sequences [60] here to avoid any noise from timing-based interrupts, and to minimize the number of AEXs induced by our exploit. Specifically, we constructed a small finite state machine that alternately revokes access to either the `sgx_get_key` or `do_egetkey` code page. Merely counting page faults now suffices to deterministically locate the return instruction ④ in `do_egetkey`. At this point, the launch key resides in the L1 cache and can thus be reliably extracted by Fore-

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

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

shadow. We observed a 100% success rate in practice; that is, our final (online) exploit extracts the full 128-bit key without noise, from a single LE run with only 13 page faults in total — without resorting to the single-stepping or `eldu` prefetching techniques of Sections 3.3 and 3.5.

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

Impact. Bypassing Intel's controversial [10] launch control policy allows one to create arbitrary production enclaves without going through a license agreement process. Removing control over which enclaves can be run is a clear breach of Intel's licensing interests, but by itself has limited impact on SGX's security objectives. We are *not* able to fabricate enclaves. Any properly implemented key derivation in an enclave will depend on either the MRENCLAVE or MRSIGNER values (Section 2.1). Neither can be forged as they rely on cryptographic properties of SHA-256 and the signer's private key respectively. The ability to create rogue production enclaves could be abused for hiding malware [51], but does not provide an enclave writer with any substantial advantage.

There is one notable exception, related to CPU tracking privacy concerns [10]. Specifically, an attacker can now create enclaves with the ability to derive a "provisioning key" that remains constant as a processor changes owners. LE should make sure that only Intel-signed enclaves can derive such keys, needed for securing long-term remote attestation keys (Section 5.2). All other `egetkey` derivations include an internal OWNEREPOCH register, which can be re-randomized when a user sells her platform. This ensures that any remaining secrets are approvedly destroyed when a computer changes owners [2]. Note that provisioning key derivations do include MRSIGNER, however, such that we cannot derive Intel's provisioning key without access to Intel's private enclave signing key.

5.2 Attacking the Intel Quoting Enclave

Background. Section 2.1 introduced local, intra-platform attestation through the `ereport` instruction. Such tagged local attestation reports are useless to a remote stakeholder, however, as they can only be verified by a target enclave executing on the *same* platform. The Intel SGX design therefore includes a trusted Quoting Enclave (QE) [2, 10] to validate local attestation reports, and sign them with an asymmetric private key. The resulting

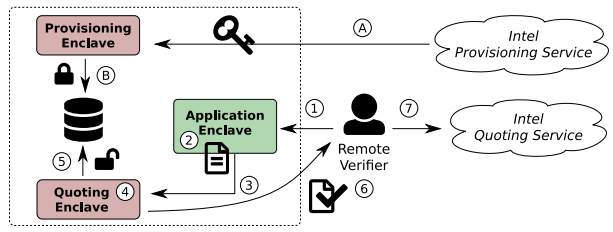


Figure 6: SGX Quoting Enclave for remote attestation.

signed attestation report, or *quote*, can now be verified by a remote party via the corresponding public key.

Intel imposes itself as a trusted third party in the attestation process. To address privacy concerns, QE implements Intel's Enhanced Privacy Identifier (EPID) [34] group signature scheme. An EPID group covers millions of CPUs of the same type (e.g., core i3, i5, i7) and security version number. In fully anonymous mode, the cryptosystem ensures that remote parties can verify quotes from genuine SGX-enabled platforms, without being able to track individual CPUs within a group or recognize previously verified platforms. In pseudonymous mode, on the other hand, remote verifiers can link different quotes from the same platform.

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

Attack and Exploitation. Remote attestation, as implemented by the SGX Quoting Enclave[4], relies on two pillars. First, QE relies on the infallibility of SGX's local attestation mechanism. An attacker getting hold of QE's report key can make QE sign arbitrary enclave measurements, effectively turning QE into a signing oracle.

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

Second, QE relies on SGX's sealing mechanism to securely store the asymmetric attestation key. Should the platform provisioning seal key leak, an attacker can get hold of the long-term attestation key and directly sign rogue enclave reports herself. We exploited both options to show how Foreshadow can adaptively dismantle different SGX primitives.

As with the LE attack, illustrated in Fig. 5, both our QE key extraction exploits target the `sgx_get_key` trusted runtime function. We again constructed a carefully crafted page fault state machine to deterministically preempt the QE execution between the `egetkey` invocation and the key buffer being overwritten. As with the LE exploit, our final attack does *not* rely on advanced single-stepping or `eldu` prefetching techniques, and achieves a 100% success rate in practice. That is, our exploit reliably extracts the full 128-bit report and provisioning seal keys from a single QE run suffering 14 page faults in total.

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

Impact. The ability to spoof remote attestation responses has profound consequences. Attestation is typically the first step to establish a secure communication channel, e.g., via an authenticated Diffie-Hellman key exchange [2]. Using our rogue quoting service, a network-level adversary (e.g., the untrusted host application) can trivially establish a man-in-the-middle position to read plus modify all traffic between a victim enclave and a remote party. All remotely provisioned secrets can now be intercepted, without even executing the victim enclave or requiring detailed knowledge of its internals — effectively rendering SGX-based DRM or privacy-preserving analytics [44, 50] applications useless. Apart from such confidentiality concerns, adversaries can furthermore fabricate arbitrary remote SGX computation results. This observation rules out transparent, integrity-only enclaved execution paradigms [56], and directly threatens an emerging ecosystem of untrusted cloud environments [4] and innovative blockchain technologies [25].

Intel's EPID [34] group signature scheme implemented by QE makes matters even worse. That is, in fully anonymous mode, obtaining a single EPID private key suffices to forge signatures for the *entire* group containing millions of SGX-capable Intel CPUs. Alarmingly, this allows us to use the platform attestation key extracted from our lab machine to forge anonymous attestations for enclaves running on remote platforms we don't even have code execution on. This does fortunately not hold for the officially recommended [34] pseudonymous mode, however, as remote stakeholders would recognize our fabricated quotes as coming from a different platform.

6 Discussion and Mitigations

Impact of Our Findings. Concurrent research on transient execution attacks [15, 24, 36, 40, 42] revealed fundamental flaws in the way current CPUs implement speculative out-of-order execution. So far, the focus of these attacks has been on breaching traditional kernel-level memory isolation barriers from an unprivileged user space process. Our work shows, however, that Meltdown-type CPU vulnerabilities also apply to non-hierarchical intra-address space isolation, as provided by modern Intel x86 SGX technology. This finding has profound consequences for the development of adequate defenses. The widely-deployed software-only KAISER [19] defense falls short of protecting enclave programs against Foreshadow adversaries. Indeed, page table isolation mitigations are ruled out, for SGX explicitly distrusts the operating system kernel, and enclaves live *within* the address space of an untrusted host process.

We want to emphasize that Foreshadow exploits a microarchitectural *implementation* bug, and does not in any way undermine the architectural *design* of Intel SGX and TEEs in general. We strongly believe that the non-hierarchical protection model supported by these architectures is still as valuable as it was before. An important lesson from the recent wave of transient execution attacks including Spectre, Meltdown, and Foreshadow, however, is that current processors exceed our levels of understanding [3, 47]. We therefore want to urge the research community to develop alternative hardware-software co-designs [11, 14], as well as inspectable open-source [47, 48] TEEs in the hopes of making future vulnerabilities easier to identify, mitigate, and recover from.

Mitigation Strategies. State-of-the-art enclave side-channel hardening techniques [8, 9, 18, 52–54] offer little protection only and cannot address the root causes of the Foreshadow attack. These defenses commonly rely on hardware transactional memory (TSX) support to detect suspicious page fault and interrupt rates in enclave mode, which only marginally increases the bar for Foreshadow attackers. First, not all SGX-capable processors are also shipped with TSX extensions, ruling out TSX-based hardening techniques for Intel's critical Launch and Quoting Enclaves. Second, since the `egetkey` instruction is *not* allowed within a TSX transaction [27], adversaries can always interrupt a victim enclave unnoticed after key

derivation to leak secrets (similar to Fig. 5). Furthermore, while the high interrupt rates generated by SGX-Step would be easily recognized, stealthy exploits can limit the number of enclave preemptions, or HyperThreading-based Foreshadow variants can be executed concurrently from another logical core. Finally, we showed how to abuse SGX's `eldu` instruction to extract enclaved memory secrets without even executing the victim enclave, effectively rendering any software-only defense strategy inherently insufficient.

Only Intel is placed in a unique position to patch hardware-level CPU vulnerabilities. They recently announced "silicon-based changes to future products that will directly address the Spectre and Meltdown threats in hardware [...] later this year." [37] Likewise, we expect Foreshadow to be directly addressed with silicon-based changes in future Intel processors. The SGX design [2] includes a notion of TCB recovery by including the CPU security version number in all measurements (Section 2.1). As such, future microcode updates could in principle mitigate Foreshadow on existing SGX-capable processors. In this respect, beta microcode updates [32] have recently been distributed to mitigate Spectre, but, at the time of this writing, no microcode patches have been released addressing Meltdown nor Foreshadow. Given the fundamental nature of out-of-order CPU pipeline optimizations, we expect it may not be feasible to directly address the Foreshadow/Meltdown access control race condition in microcode. Alternatively, based on our findings (see Appendix A) that Foreshadow requires enclave data to reside in the L1 cache, we envisage a hardware-software co-design mitigation strategy. Foreshadow-resistant enclaves should be guaranteed that (*i*) both logical cores in a HyperThreading setting execute within the same enclave [8,18,54], and (*ii*) the L1 cache is flushed upon each enclave exiting event [11].

7 Related Work

Several recent studies investigate attack surface for SGX enclaves. Existing attacks either exploit low-level memory safety vulnerabilities [38, 59], or abuse application-specific information leakage from side-channels. Importantly, in contrast to Foreshadow, all known attacks explicitly fall out-of-scope of Intel SGX's threat model [28, 33], and can be effectively avoided by rewriting the victim enclave's code to exclude such vulnerabilities.

Conventional microarchitectural side-channels [16] are, however, considerably amplified in the context of SGX's strengthened attacker model. This point has been repeatedly demonstrated in the form of a steady stream of high-resolution PRIME+PROBE CPU cache [6,12,17,23,46,51] and branch prediction [13, 39] attacks against SGX enclaves. The additional capabilities of a root-level attacker have furthermore been leveraged to construct instruction-granular enclave interrupt primitives [57], and to exploit side-channel leakage from x86 memory paging [58, 60] and segmentation [22]. Unexpected side-channels can also arise at the application level. We for example recently reported [30] a side-channel vulnerability in auto-generated `edger8r` code of the official Intel SGX SDK.

Concurrent research [7, 49] has demonstrated proof-of-concept Spectre-type speculation attacks against specially crafted SGX enclaves. Both attacks rely on executing vulnerable code within the victim enclave. Our attack, in contrast, does not require any specific code in the victim enclave, and can even extract memory contents without ever executing the victim enclave. While existing work shows vulnerable gadgets exist in the SGX SDK [7], such Spectre-type attack surface can be mitigated by patching the SDK. Recent Intel microcode updates furthermore address Spectre-type attacks against SGX enclaves directly at the hardware level, by cleansing the CPU's branch target buffer on every enclave entry/exit [7].

8 Conclusion

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

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

Acknowledgments. This research was partially supported by the Research Fund KU Leuven, the Technion Hiroshi Fujiwara cyber security research center, the Israel cyber bureau, by NSF awards `#1514261` and `#1652259`, the financial assistance award `70NANB15H328` from the U.S. Department of Commerce, National Institute of Standards and Technology, the 2017-2018 Rothschild Postdoctoral Fellowship, and the Defense Advanced Research Project Agency (DARPA) under Contract `#FA8650-16-C-7622`. Jo Van Bulck and Raoul Strackx are supported by a grant of the Research Foundation - Flanders (FWO).

References

[1] ALVES, T., AND FELTON, D. Trustzone: Integrated hardware and software security. *ARM white paper 3*, 4 (2004), 18–24.

[2] ANATI, I., GUERON, S., JOHNSON, S., AND SCARLATA, V. Innovative technology for CPU based attestation and sealing. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy* (2013), ACM.

[3] BAUMANN, A. Hardware is the new software. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems* (2017), ACM, pp. 132–137.

[4] BAUMANN, A., PEINADO, M., AND HUNT, G. Shielding applications from an untrusted cloud with Haven. In *USENIX Symposium on Operating Systems Design and Implementation* (2014).

[5] BRASSER, F., CAPKUN, S., DMITRIENKO, A., FRASSETTO, T., KOSTIAINEN, K., MÜLLER, U., AND SADEGHI, A.-R. Dr. sgx: Hardening sgx enclaves against cache attacks with data location randomization. *arXiv preprint arXiv:1709.09917* (2017).

[6] BRASSER, F., MÜLLER, U., DMITRIENKO, A., KOSTIAINEN, K., CAPKUN, S., AND SADEGHI, A.-R. Software grand exposure: SGX cache attacks are practical. In *11th USENIX Workshop on Offensive Technologies* (2017), WOOT '17, USENIX Association.

[7] CHEN, G., CHEN, S., XIAO, Y., ZHANG, Y., LIN, Z., AND LAI, T. H. Sgxpectre attacks: Leaking enclave secrets via speculative execution. *arXiv preprint arXiv:1802.09085* (2018).

[8] CHEN, G., WANG, W., CHEN, T., CHEN, S., ZHANG, Y., WANG, X., LAI, T.-H., AND LIN, D. Racing in hyperspace: Closing hyper-threading side channels on sgx with contrived data races. In *Proceedings of the IEEE Symposium on Security and Privacy* (2018), IEEE.

[9] CHEN, S., ZHANG, X., REITER, M. K., AND ZHANG, Y. Detecting privileged side-channel attacks in shielded execution with déjà vu. In *Proceedings of the 2017 Asia Conference on Computer and Communications Security* (2017), Asia CCS '17, ACM, pp. 7–18.

[10] COSTAN, V., AND DEVADAS, S. Intel SGX explained. Cryptology ePrint Archive, Report 2016/086, 2016.

[11] COSTAN, V., LEBEDEV, I., AND DEVADAS, S. Sanctum: Minimal hardware extensions for strong software isolation. In *Proceedings of the 25th USENIX Security Symposium* (2016), USENIX Association.

[12] DALL, F., DE MICHELI, G., EISENBARTH, T., GENKIN, D., HENINGER, N., MOGHIMI, A., AND YAROM, Y. Cachequote: Efficiently recovering long-term secrets of sgx epid via cache attacks. *IACR Transactions on Cryptographic Hardware and Embedded Systems 2018*, 2 (2018), 171–191.

[13] EVTYUSHKIN, D., RILEY, R., ABU-GHAZALEH, N. C., PONOMAREV, D., ET AL. Branchscope: A new side-channel attack on directional branch predictor. In *Proceedings of the 23th International Conference on Architectural Support for Programming Languages and Operating Systems* (2018), ACM, pp. 693–707.

[14] FERRAIUOLO, A., BAUMANN, A., HAWBLITZEL, C., AND PARNO, B. Komodo: Using verification to disentangle secure-enclave hardware from software. In *Proceedings of the 26th Symposium on Operating Systems Principles* (2017), ACM.

[15] FOGH, A. Negative result: Reading kernel memory from user mode. `https://cyber.wtf/2017/07/28/`, July 2017.

[16] GE, Q., YAROM, Y., COCK, D., AND HEISER, G. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *Journal of Cryptographic Engineering 8*, 1 (2018), 1–27.

[17] GÖTZFRIED, J., ECKERT, M., SCHINZEL, S., AND MÜLLER, T. Cache attacks on intel sgx. In *Proceedings of the 10th European Workshop on Systems Security* (New York, NY, USA, 2017), EuroSec'17, ACM, pp. 2:1–2:6.

[18] GRUSS, D., LETTNER, J., SCHUSTER, F., OHRIMENKO, O., HALLER, I., AND COSTA, M. Strong and efficient cache side-channel protection using hardware transactional memory. In *USENIX Security Symposium* (2017).

[19] GRUSS, D., LIPP, M., SCHWARZ, M., FELLNER, R., MAURICE, C., AND MANGARD, S. Kaslr is dead: long live kaslr. In *International Symposium on Engineering Secure Software and Systems* (2017), Springer, pp. 161–176.

[20] GRUSS, D., MAURICE, C., FOGH, A., LIPP, M., AND MANGARD, S. Prefetch side-channel attacks: Bypassing smap and kernel aslr. In *Proceedings of the 2016 ACM Conference on Computer and communications security* (2016), ACM, pp. 368–379.

[21] GUERON, S. A memory encryption engine suitable for general purpose processors. *IACR Cryptology ePrint Archive* (2016), 204.

[22] GYSELINCK, J., VAN BULCK, J., PIESSENS, F., AND STRACKX, R. Off-limits: Abusing legacy x86 memory segmentation to spy on enclaved execution. In *Publication at the 2018 International Symposium on Engineering Secure Software and Systems (ESSoS'18)* (June 2018), LNCS, Springer. (in print).

[23] HÄHNEL, M., CUI, W., AND PEINADO, M. High-resolution side channels for untrusted operating systems. In *2017 USENIX Annual Technical Conference* (2017), ATC '17, USENIX Association.

[24] HORN, J. Reading privileged memory with a side-channel. `https://googleprojectzero.blogspot.com/2018/01/`, January 2018.

[25] INTEL. `https://sawtooth.hyperledger.org/docs/core/releases/latest/introduction.html`.

[26] INTEL. *Intel 64 and IA-32 architectures optimization reference manual*, December 2017.

[27] INTEL. *Intel 64 and IA-32 Architectures Software Developer's Manual – Combined Volumes*, December 2017.

[28] INTEL. *Intel Software Guard Extensions SDK for Linux OS: Developer Reference*, November 2017.

[29] INTEL. *Intel Analysis of Speculative Execution Side Channels*, January 2018. Reference no. 336983-001.

[30] INTEL. *Intel Software Guard Extensions (SGX) SW Development Guidance for Potential Edger8r Generated Code Side Channel Exploits*, March 2018. Revision 1.0.

[31] INTEL. *Retpoline: A Branch Target Injection Mitigation*, February 2018. Reference no. 337131-001.

[32] INTEL. *Speculative Execution Side Channel Mitigations*, May 2018. Reference no. 336996-002.

[33] JOHNSON, S. Intel SGX and side-channels. `https://software.intel.com/en-us/articles/intel-sgx-and-side-channels`, March 2017.

[34] JOHNSON, S., SCARLATA, V., ROZAS, C., BRICKELL, E., AND MCKEEN, F. Intel® software guard extensions: Epid provisioning and attestation services. *White Paper 1* (2016), 1–10.

[35] KAPLAN, D., POWELL, J., AND WOLLER, T. Amd memory encryption. *White paper* (2016).

[36] KOCHER, P., GENKIN, D., GRUSS, D., HAAS, W., HAMBURG, M., LIPP, M., MANGARD, S., PRESCHER, T., SCHWARZ, M., AND YAROM, Y. Spectre attacks: Exploiting speculative execution. *arXiv preprint arXiv:1801.01203* (2018).

[37] KRZANICH, B. Intel (intc) ceo brian krzanich on q4 2017 results. `https://seekingalpha.com/article/4140338-intel-intc-ceo-brian-krzanich-q4-2017-results-earnings-call-transcript`, January 2018.

[38] LEE, J., JANG, J., JANG, Y., KWAK, N., CHOI, Y., CHOI, C., KIM, T., PEINADO, M., AND KANG, B. B. Hacking in darkness: Return-oriented programming against secure enclaves. In *USENIX Security* (2017), pp. 523–539.

[39] LEE, S., SHIH, M.-W., GERA, P., KIM, T., KIM, H., AND PEINADO, M. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In *Proceedings of the 26th USENIX Security Symposium* (August 2017), USENIX Association.

[40] LIPP, M., SCHWARZ, M., GRUSS, D., PRESCHER, T., HAAS, W., MANGARD, S., KOCHER, P., GENKIN, D., YAROM, Y., AND HAMBURG, M. Meltdown. *arXiv preprint arXiv:1801.01207* (2018).

[41] MAENE, P., GÖTZFRIED, J., DE CLERCQ, R., MÜLLER, T., FREILING, F., AND VERBAUWHEDE, I. Hardware-based trusted computing architectures for isolation and attestation. *IEEE Transactions on Computers*, 99 (2017).

[42] MAISURADZE, G., AND ROSSOW, C. Speculose: Analyzing the security implications of speculative execution in cpus. *arXiv preprint arXiv:1801.04084* (2018).

[43] MCKEEN, F., ALEXANDROVICH, I., BERENZON, A., ROZAS, C. V., SHAFI, H., SHANBHOGUE, V., AND SAVAGAONKAR, U. R. Innovative instructions and software model for isolated execution. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy* (2013), ACM.

[44] MOBILECOIN. Mobilecoin. `https://www.mobilecoin.com/whitepaper-en.pdf`, December 2017.

[45] MOGHIMI, A., EISENBARTH, T., AND SUNAR, B. Memjam: A false dependency attack against constant-time crypto implementations. *arXiv preprint arXiv:1711.08002* (2017).

[46] MOGHIMI, A., IRAZOQUI, G., AND EISENBARTH, T. Cachezoom: How SGX amplifies the power of cache attacks. In *Conference on Cryptographic Hardware and Embedded Systems* (2017), CHES '17.

[47] MÜHLBERG, J. T., AND VAN BULCK, J. Reflections on post-Meltdown trusted computing: A case for open security processors. *;login: the USENIX magazine Vol. 43*, No. 3 (Fall 2018). to appear.

[48] NOORMAN, J., VAN BULCK, J., MÜHLBERG, J. T., PIESSENS, F., MAENE, P., PRENEEL, B., VERBAUWHEDE, I., GÖTZFRIED, J., MÜLLER, T., AND FREILING, F. Sancus 2.0: A low-cost security architecture for IoT devices. *ACM Transactions on Privacy and Security (TOPS)* (2017).

[49] O'KEEFFE, D., MUTHUKUMARAN, D., AUBLIN, P.-L., KELBERT, F., PRIEBE, C., LIND, J., ZHU, H., AND PIETZUCH, P. Sgxspectre. https://github.com/lsds/spectre-attack-sgx, 2018.

[50] OPEN WHISPER SYSTEMS. `https://signal.org/blog/private-contact-discovery/`.

[51] SCHWARZ, M., WEISER, S., GRUSS, D., MAURICE, C., AND MANGARD, S. Malware guard extension: Using sgx to conceal cache attacks. In *14th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment* (July 2017), DIMVA'17.

[52] SEO, J., LEE, B., KIM, S., AND SHIH, M.-W. SGX-Shield: Enabling address space layout randomization for sgx programs. In *Proceedings of the 2017 Network and Distributed System Security Symposium (NDSS 2017)* (Feb. 2017).

[53] SHIH, M.-W., LEE, S., KIM, T., AND PEINADO, M. T-SGX: Eradicating controlled-channel attacks against enclave programs. In *Proceedings of the 2017 Network and Distributed System Security Symposium (NDSS 2017)* (Feb. 2017).

[54] STRACKX, R., AND PIESSENS, F. The heisenberg defense: Proactively defending sgx enclaves against page-table-based side-channel attacks. *arXiv preprint arXiv:1712.08519* (Dec. 2017).

[55] TOMASULO, R. M. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of research and Development 11*, 1 (1967), 25–33.

[56] TRAMER, F., ZHANG, F., LIN, H., HUBAUX, J.-P., JUELS, A., AND SHI, E. Sealed-glass proofs: Using transparent enclaves to prove and sell knowledge. In *2nd IEEE European Symposium on Security and Privacy (Euro S&P)* (2017), IEEE.

[57] VAN BULCK, J., PIESSENS, F., AND STRACKX, R. SGX-Step: A practical attack framework for precise enclave execution control. In *Proceedings of the 2nd Workshop on System Software for Trusted Execution* (2017), SysTEX'17, ACM, pp. 4:1–4:6.

[58] VAN BULCK, J., WEICHBRODT, N., KAPITZA, R., PIESSENS, F., AND STRACKX, R. Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution. In *Proceedings of the 26th USENIX Security Symposium* (August 2017), USENIX Association.

[59] WEICHBRODT, N., KURMUS, A., PIETZUCH, P., AND KAPITZA, R. Asyncshock: exploiting synchronisation bugs in Intel SGX enclaves. In *European Symposium on Research in Computer Security* (2016), ESORICS '16, Springer.

[60] XU, Y., CUI, W., AND PEINADO, M. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *36th IEEE Symposium on Security and Privacy* (May 2015), IEEE.

[61] YAROM, Y., AND FALKNER, K. Flush+reload: A high resolution, low noise, L3 cache side-channel attack. In *23rd USENIX Security Symposium* (2014), USENIX Association, pp. 719–732.

A Foreshadow's Cache Requirements

In this appendix, we provide experimental evidence that Foreshadow requires enclaved data to be present in the L1 CPU cache. We attribute this condition to SGX's microarchitectural implementation, for previous Meltdown-type exploits targeting hierarchical kernel memory, do *not* have such strict caching requirements.

Placing Secrets at Specific Cache Levels. We rely on Intel's Transactional Synchronization eXtensions (TSX) to ensure that secrets only reside in the L2 and L3 cache levels, but not in L1. Particularly, we abuse that after a TSX transaction has started writes are cached in the L1 cache, without being propagated down to L2 and L3. When a transaction aborts and needs to be rolled back, all cache lines in the write set are simply marked invalid in the L1 cache. Future references to these addresses only hit the L2 cache, which still holds their original value.

Listing 3 displays how we leverage this mechanism to ensure that the secret is only present in the L2 and L3 caches. At Line 3 we start a new transaction. Next the secret is modified to ensure its updated value is located in the L1 cache. When finally the transaction is aborted, the L1 cache line holding the secret is marked as invalid, but the corresponding L2/L3 cache lines remain unaffected. Execution is rolled back to Line 3 where from a programmer's perspective `rtm_begin()` returned -1 immediately. The `mfence` instructions ensure that memory accesses cannot be reordered.

Verifying Cache Levels. As enclave memory is exclusively accessible to the enclave, we rely on a carefully crafted benchmark enclave that places a secret at the intended cache level. Unfortunately returning execution control from the enclave (`eexit`), may inadvertently evict

```
void load_in_L2( uint64_t *secret ) {
  asm volatile ( "mfence\n" );
  if ( rtm_begin() == 0 ) {
    *(secret) += 1;
    rtm_abort();
  }
  asm volatile ( "mfence\n" );
}
```

Listing 3: We evict secrets from the L1 cache by including them in the write set of an aborted TSX transaction.

enclave secrets to secondary cache levels or even to main memory. To detect such events, we confirm their current cache level after every attack iteration.

Verifying at which level enclave data is currently cached is challenging. SGX's abort page semantics prevent us from directly measuring the access times of enclave data: we did not observe any timing difference between accessing cached and non-cached secrets from outside the enclave. Moving such cache verification code into the enclave, on the other hand, is infeasible as `rdtsc` instructions cannot be executed in enclave mode on SGXv1 machines [27]. We therefore resort to creating a debug benchmark enclave and measure access times of reading enclave data through the `edbgrd` instruction. As `edbgrd` may inadvertently move enclave data to caches closer to the processor, we only perform this additional verification step *after* the actual Foreshadow attack attempt.

We carefully benchmarked the access times for enclave secrets residing in L1, L2, and main memory. Table 1 displays the median timing results for 100,000 runs. As expected, accessing enclave secrets in the L1 cache is only slightly faster than when they need to be fetched from the second-level L2 cache. This timing difference (6 cycles) is furthermore identical to L1/L2 cache hits of non-enclave memory. When SGX memory needs to be fetched from main memory, however, it needs to be decrypted by the memory encryption engine which adds significant additional latency.

Experimental Setup. As we are only interested in whether the attack variations succeed, not their bandwidth, we made some changes to our attack setting. Each attack operates in a guess/verify fashion; for every 256 possible values of the secret byte, we performed 100,000 Foreshadow rounds. Each round starts by first entering the benchmark enclave to explicitly place the secret at the desired cache level. After Foreshadow's transient execution phase, a single oracle slot (the current guess) is reloaded to receive the output of the transient instruction sequence. Finally we verify whether the enclave secret is still located at its intended cache level by measuring `edbgrd` timing. Any attack results from inadvertently evicted enclave secrets are discarded.

Table 1: Access times for enclave and non-enclave memory at various cache levels (median over 100,000 runs).

Cache event	Unprotected (cycles)	`edbgrd` (cycles)
L1 cache hit	40	1,400
L2 cache hit	46	1,406
Cache miss	238	1,734

Success Rates. We first execute the Foreshadow-L1 attack 100,000 times against an enclave secret residing in the L1 cache. When we observe `edbgrd` timings larger than 1,405 cycles after the attack attempt, we assume the secret must have been evicted from the L1 cache and discard the result. For *every* of the remaining 96,594 attack rounds, we successfully received the secret.

We repeated the same test for enclave secrets residing in the L2 cache. This time, we discarded results with `edbgrd` timings exceeding 1,408 ticks after the attack. Out of the 98,610 remaining attack attempts, *none* succeeded in speculatively loading a secret-dependent oracle buffer slot in the transient execution phase.

To rule out the possibility that the transient instructions may need more attempts to elevate the enclave secret from the L2 to the L1 cache, we ran the same benchmark with 1,000 repeated transient executions before actually reloading the oracle buffer. This severely reduced the number of accepted attack attempt down to 10,205. Still, *all* Foreshadow-L2 attack attempts failed.

Conclusions. As long as enclave secrets reside in the L1 cache, we observe 100% success rates. Even though L2 cache accesses only take a mere 6 cycles longer, the success rates sharply drop to zero. The Meltdown [40] attack to extract supervisor data does *not* suffer from such a hard limit, and has even been successfully applied to read kernel secrets directly from main memory. When applying Foreshadow against kernel data, we could indeed trivially extract kernel secrets from the L2 cache without noticing a significant success rate drop.

We conclude that both Meltdown and Foreshadow exploit a similar race condition vulnerability in the CPU's out-of-order pipeline behavior, but Intel SGX's abort page semantics apparently have a profound microarchitectural impact. Attack conditions are much more stringent to breach enclave than kernel isolation.