Theme Article

Breaking Virtual Memory Protection and the SGX Ecosystem with Foreshadow

Jo Van Bulck
Katholieke Universiteit Leuven

Marina Minkin
University of Michigan

Ofir Weisse
University of Michigan

Daniel Genkin
University of Michigan

Baris Kasikci
University of Michigan

Frank Piessens
Katholieke Universiteit Leuven

Mark Silberstein
Technion–Israel Institute of Technology

Thomas F. Wenisch
University of Michigan

Yuval Yarom
University of Adelaide and Data61, CSIRO

Raoul Strackx
Katholieke Universiteit Leuven

***Abstract*—Foreshadow is a speculative execution attack that allows adversaries to subvert the security guarantees of Intel's Software Guard eXtensions (SGX). Foreshadow allows access to data across process boundaries, and allows virtual machines (VMs) to read the physical memory belonging to other VMs or the hypervisor.**

Over the past year and a half, *speculative execution attacks*—attacks that subvert memory protection by exploiting the speculative execution features of modern out-of-order processors—have gone from a hypothetical possibility to a very real threat.[1,2] These attacks allow software to exploit microarchitectural flaws, wherein speculative *wrong-path* instructions are allowed to access memory in violation of architectural protection guarantees. For example, the Meltdown flaw[2] allows the user-level code to access kernel memory locations that are mapped into the process address space but are restricted to privileged

Digital Object Identifier 10.1109/MM.2019.2910104
Date of publication 15 April 2019; date of current version 8 May 2019.

Published by the IEEE Computer Society

accesses only. Speculative execution attacks exploit the transient data access of wrong-path instructions by leaking secret data into a *side channel*. Side-channel attacks exploit subtle timing variations in program execution resulting from state changes in microarchitectural (i.e., unspecified in the instruction set architecture) structures or contention on microarchitectural CPU resources to extract otherwise-unavailable secret information.

Processor vendors have invested significant effort[3] to roll out mitigations for the earliest discovered speculative execution attacks, Meltdown[2] and Spectre.[1] Unfortunately these measures are ineffective against Foreshadow (also known as "L1 Terminal Fault (L1TF)" in Intel's product literature[4]). Foreshadow allows the attacker code direct access to arbitrary physical addresses through speculative wrong-path execution. Unlike previously disclosed speculative execution attacks, this vulnerability fully subverts the isolation provided by the traditional virtual memory abstraction, upon which the classic security mechanisms of operating systems and virtual machines are built, by allowing attackers to access data that has no virtual address mapping. The key restriction of Foreshadow/L1TF is that as far as we know, only physical addresses cached in the L1 data cache can be exfiltrated. However, as we show, there are often mechanisms available to cause the processor to load desired data into cache; in some circumstances, entire address spaces can be exfiltrated.

Foreshadow allows the attacker code direct access to arbitrary physical addresses through speculative wrong-path execution.

FORESHADOW-SGX

Our international collaboration discovered Foreshadow/L1TF in the context of efforts to investigate the security of Intel's Software Guard eXtensions (SGX). SGX is a set of architectural extensions, which collectively provide strong security guarantees to software running in the presence of powerful adversaries. SGX promises secure execution on adversary-controlled machines. For example, in the context of cloud computing, SGX guarantees that the processor executes the unmodified code provided by the customer and that data accessed during execution cannot be observed by the privileged cloud vendor.

To support such private and secure code execution, SGX provides isolated execution environments, called *enclaves*, which offer confidentiality and integrity guarantees to programs running inside them. Enclaves are secure even in the presence of a malicious operating system, hypervisor, or firmware. Enclaves are also resilient to physical attacks from outside the CPU package, such as probes on the memory bus.

At a high level, SGX achieves these strong security guarantees by encrypting the enclave memory and protecting it with a secure authentication code, making the associated cryptographic keys inaccessible to software. Finally, SGX provides a *remote attestation* mechanism. Attestation allows enclaves to prove to remote parties that the code and data upon which they operate are unmodified and, importantly, the processor upon which they execute is a genuine (hence, presumed secure) Intel processor.

Notwithstanding its strong security guarantees, SGX does not protect against microarchitectural side-channel attacks. Since their introduction over a decade ago, microarchitectural attacks have been used to break numerous cryptographic implementations, track program behavior, and create covert communication channels. Since SGX's release, there have been many works that demonstrated side-channel attacks on SGX enclaves (e.g., work presented by Xu *et al.*[5])—exploiting vulnerabilities *that already exist* in enclaves' code to leak sensitive data.

Instead, our investigation was motivated by the following question: Can an adversary extract secret data from an enclave's address space when the code running in that enclave does not itself have any security vulnerabilities?

Next, given the importance of SGX remote attestation we asked: What are the implications of such security breaches on the SGX integrity guarantees? Can an adversary using a side channel erode trust in SGX remote attestation?

We answer both questions in the affirmative.

OUR CONTRIBUTIONS

Breaking SGX's Confidentiality

Foreshadow-SGX exploits the speculative execution features present in all SGX-enabled

Intel CPUs to read the entire address space of victim enclaves. Crucially, unlike previous Spectre-style speculative execution attacks on SGX, our attack *does not require* any form of cooperation or particular flaw in the victim enclave. In fact, our attack reads all the secrets of the victim enclave without requiring that enclave to execute any instruction.

Breaking the Integrity of Sealed Data

Going beyond the attacks on the SGX confidentiality properties, we show that Foreshadow also compromises SGX's *long-term storage integrity* guarantees. Specifically, beyond secure computation, SGX also provides private and authenticated long-term storage, which is implemented via a special *sealing* application programming interface (API). This storage mechanism allows enclaves to encrypt and verify data stored by the (untrusted) operating system.

As we show, we can use our attack on SGX's confidentiality to extract the sealing key from a victim enclave that uses the SGX sealing mechanism.[11] After recovering the sealing key, we use it to unseal and read the sealed information, then modify and reseal it. As SGX provides no means to detect such a change, the victim consequently operates on data corrupted by the attacker.

Breaking Remote Attestation

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

As we show, we can mount our attacks on enclaves written by Intel and extract their sealing key. Using the extracted sealing key, we then unseal the persistent storage of the SGX Quoting Enclave, which contains the machine's private attestation key. With this key, we can construct malicious SGX simulators that can forge the attestation process, masquerading as enclaves that are allegedly running on genuine Intel processors with the SGX security guarantees. As the simulated enclaves do not offer any security guarantees, this attack undermines the trustworthiness of SGX's attestation mechanism.

Exploiting SGX's Privacy Guarantees

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

Demonstrating the Brittleness of the SGX Ecosystem

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

First real demonstration of the L1TF vulnerability

Our international collaboration reported our proof-of-concept exploit of SGX to Intel. Prompted by our reports, Intel subsequently discovered and revealed the underlying L1TF microarchitectural flaw, with consequences much broader than SGX, including subverting memory protections essential for cloud computing.

BACKGROUND—SIDE-CHANNEL AND SPECULATIVE EXECUTION ATTACKS

We briefly explain past side-channel and speculative execution attack mechanisms.

The Flush+Reload Attack

Flush+Reload[7] is a cache-based microarchitectural attack technique that detects access to a shared memory location. The technique comprises two main operations. The *flush* operation evicts the contents of a monitored memory

location from the cache. Typically, the eviction is done via a dedicated instruction, such as the x86 CLFLUSH instruction. The *reload* operation later measures the time it takes to access the monitored location. Based on this time, the attacker determines whether the monitored location was loaded into the cache in the interim since the flush operation.

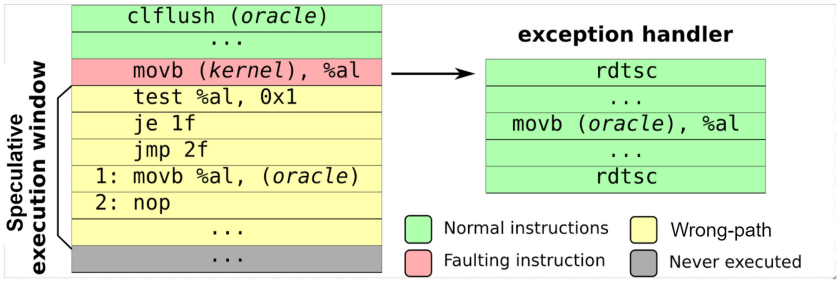

Figure 1. Rogue data cache loads (Meltdown) can be leveraged to leak sensitive data from more privileged security layers.

In a typical attack scenario, an attacker flushes one or more monitored locations. The attacker then either executes an operation it wants to analyze or waits until the operation is naturally executed by the victim. The attacker then reloads the monitored locations while recording the amount of time required to perform the reload. As the analyzed operation accesses (and thereby caches) some of the monitored locations, but not others, the attacker is then able to learn information about the victim memory access pattern. Flush+Reload has been extensively used for side-channel attacks and is used as a reliable *covert* channel in speculative execution attacks.[1,2]

Spectre and Meltdown

Spectre and Meltdown attacks perform unauthorized data access while executing wrong-path instructions in vulnerable processors. When the CPU squashes these instructions, it does not fully revert all the side effects they have on microarchitectural state, and in particular, processor cache state. Spectre and Meltdown[1,2] exploit these side effects to leak information across protection domains. The attacks cause the CPU to speculatively execute a code *gadget,* which implements the transmitting side of a covert channel, sending information that would otherwise be unavailable to the receiver.

Specifically, the Meltdown attack (Figure 1) is possible because vulnerable processors handle memory exception faults at instruction retirement rather than as instructions execute, creating a window of vulnerability where data may be forwarded to wrong-path instructions that follow the faulting access. So, when a user program attempts to read from a kernel address, the processor may speculatively execute the wrong-path instructions that follow the kernel memory read. By placing a gadget that sends the value forwarded by the faulting load through a covert channel, an attack can retrieve the contents of that address even for accesses that violate memory protection.

Figure 1 illustrates a toy Meltdown example wherein an attacker exfiltrates one bit of information across privilege levels. In the first step, the attacker attempts to read data from a more privileged protection layer, which will eventually lead to a memory protection exception. But, the subsequent dependent wrong-path instructions can receive the value from the read—even though it will eventually fault—and encode secrets in the CPU cache. The example uses a reliable Flush+Reload[7] covert channel, where the wrong-path instruction sequence loads a predetermined memory location into the cache, dependent on the least significant bit of the kernel data. When the speculation resolves, the load faults and an exception handler is invoked. In this example, we show the attacker code for retrieving the secret within the exception handler, wherein the adversary retrieves the secret bit by carefully measuring the amount of time it takes to reload the predetermined memory location.

The Meltdown attack, however, does not allow attackers to perform unauthorized reads from memory protected by SGX. In contrast to Meltdown, which relies on a page fault after accessing kernel space, accessing SGX memory *does not* produce a page fault. Instead, such accesses trigger *abort page semantics* and the load instead returns the dummy value *0xFF,* precluding wrong-path code from leaking a secret value.

BACKGROUND—INTEL SGX

Memory Isolation

SGX enclaves live in the virtual address space of a conventional user 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/illegal enclave memory mappings. Subsequent address translations are cached in the processor's TLB. The processor performs a TLB shootdown for enclave entries upon enclave exit to prevent access to enclave memory from code executing outside the enclave. Any attempt to directly access private pages from outside the enclave results in abort page semantics: reads return the value 0xFF and writes are ignored.

SGX further protects enclaves against motivated adversaries that exploit Rowhammer DRAM bugs, or resort to physical cold boot attacks. A hardware-level memory encryption engine[8] transparently safeguards the integrity, confidentiality, and freshness of enclaved code and data while it resides outside of the processor package. That is, any access to main memory is first authenticated and decrypted before cached on chip in plaintext.

Enclave code can only be invoked 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 procedure, which securely stores and wipes CPU register contents before transferring control to the untrusted operating system. A dedicated ERESUME instruction allows the unprotected application to resume a previously interrupted enclave.

Enclave Measurement

While an enclave is prepared for launch by untrusted system software, the processor composes a secure hash (i.e., a "measurement") of the enclave's initial code and data. Besides this content-based identity (MRENCLAVE), each enclave also features an alternative, author-based identity (MRSIGNER) that 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. Thus, 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 within the processor and exclusively accessible to key derivation hardware. Enclaves make use of the key derivation facility by using the SGX instruction EGETKEY. For instance, enclaves can invoke EGETKEY to generate *sealing keys* based on either the calling enclave's content-based (MRENCLAVE) or developer-based (MRSIGNER) 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: 1) a *launch enclave*, which controls entry into other enclaves; 2) a *provisioning enclave* to supply the long-term platform attestation key; and 3) a *quoting enclave*, which uses the asymmetric platform attestation key to sign local attestation reports for a remote stakeholder.

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 mounted entirely from user space. Our attack essentially implies that current SGX implementations cannot even protect enclave

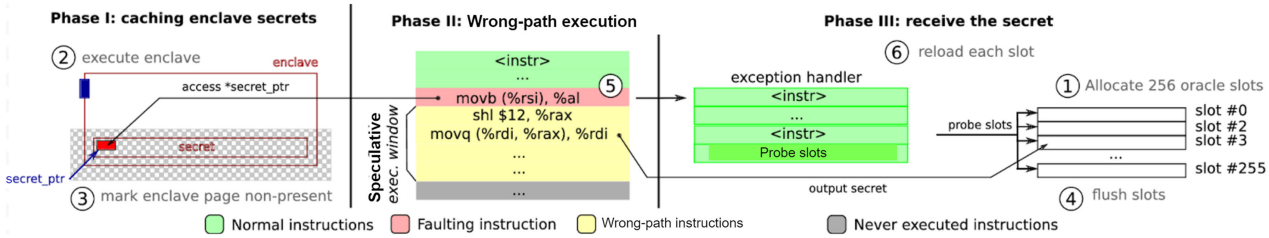

Figure 2. Basic overview of the Foreshadow attack to extract a single byte from an SGX enclave.

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 and existing Spectre-style speculative execution attacks[5] against SGX enclaves, Foreshadow does not require any side-channel vulnerabilities, code gadgets, or even knowledge of the victim enclave's code. 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)."[9] As Foreshadow compromises confidentiality of production enclave memory, this Intel SGX security objective is broken.

Our basic attack requires enclave secrets to reside in the L1 data cache. 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 requiring the victim enclave's cooperation. When combined with a state-of-the-art enclave execution control framework, such as *SGX-Step*,[10] our root attack can 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."[11] While we cannot break integrity of enclaved data directly, we leverage Foreshadow to extract enclave sealing and attestation keys. The former comprises the confidentiality and integrity of sealed secrets, whereas the latter can be used to forge attestation reports. Our attack on Intel's trusted quoting enclave for remote attestation further collapses confidentiality and integrity guarantees for remote computations and secret provisioning.

THE FORESHADOW ATTACK

The basic Foreshadow attack extracts a single byte from an SGX enclave in three distinct phases, visualized in Figure 2. As part of the attack preparation, the untrusted enclave host application first allocates a covert channel array ① with 256 slots, each measuring 4 KB in size (padded to avoid hardware prefetcher interactions). In Phase I of the attack, plaintext enclave data are loaded into the cache. With a malicious OS, the relevant cache line can be fetched easily without the victim running. Next, Phase II dereferences the enclave secret and speculatively executes a wrong-path instruction sequence that loads a secret-dependent line into the cache. Finally, Phase III acts as the receiving end of the Flush+Reload covert channel and reloads the array slots to determine the secret byte.

Phase I: Caching Enclave Secrets

In contrast to previous research,[1] and consistent with Intel's vulnerability disclosures,[4] we found consistently that enclave secrets never reach the wrong-path instructions in Phase II if they do not already reside in the L1 cache. The first phase of the basic Foreshadow attack

executes the victim enclave ② to load plaintext secrets into cache.

Bringing Data Into the L1 Cache on Behalf of the Victim

The SGX design explicitly relies on an untrusted OS for oversubscribing the limited encrypted physical memory, called the enclave page cache (EPC). To do so, the untrusted OS may make use of the privileged EWB and ELDU SGX instructions that respectively copy encrypted 4 KB enclave pages out of, and back into the EPC. We observed that, when decrypting an encrypted enclave page, the ELDU instruction loads the entire page as plaintext into the CPU's L1 cache. Thus, Foreshadow can bring the victim's data into the L1 cache without having the victim execute any instructions.

Phase II: Wrong-Path Execution

In the second phase, we dereference *secret_ptr* and speculatively execute a wrong-path instruction sequence to exfiltrate the secret. Under SGX, abort page semantics normally prevent wrong-path instructions from observing values that originate in enclave memory—they instead observe only the dummy value 0xFF. However, Foreshadow circumvents this protection by taking advantage of the fact that abort page semantics apply only *after* the legacy page table permission check succeeds without triggering a page fault. Specifically, the attacker removes access permissions from the enclave's memory to cause, as dubbed by Intel, a translation terminal fault.[4] Despite the failed translation, the processor's load unit nevertheless forms a physical address using the faulty translation and accesses the L1 data cache. Consequently, the actual value at the corresponding physical address, rather than a dummy value, is propagated from L1 to dependent wrong-path instructions. In our running example, we proceed by revoking ③ all access permissions to the enclave page we wish to read via the mprotect user space API:

```
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 accesses to the page lead to a fault. To ensure the successful exfiltration of the secret byte via the covert channel array, we flush the slots from the cache ④. Finally, the attacker dereferences *secret_ptr* and executes the wrong-path instruction sequence ⑤.

Phase III: Receiving the Secret

When the faulting load retires, the processor rolls back subsequent wrong-path instructions, discarding uncommitted register changes and raises a page fault. After the fault is caught by the operating system, the attacker's user-level exception handler is called. Here, she carefully measures ⑥ the timing to reload each slot. If the wrong-path instruction sequence reached the step that touches the array slot at the secret index, the corresponding cache line will hit, resulting in a much shorter access time.

Implications

Foreshadow allows dumping the entire address space of SGX enclaves at any time. We successfully used Foreshadow to extract the launch and attestation secrets protected by the Intel's Launch and Quote enclaves. Extracting the private attestation keys from the Quote Enclave allowed us to sign counterfeit attestation proofs, which were approved by Intel Attestation Services.

ATTACKING INTEL'S ENCLAVES

Whereas SGX is largely realized in hardware and microcode, Intel implements certain critical functionality in software through dedicated *architectural enclaves*. These enclaves are part of the trusted computing base and were written by experts with detailed knowledge of SGX. No obvious security flaws have ever been found, and Intel's architectural enclaves implement various defense-in-depth mechanisms. For example, even though private memory should never leak from enclaves, sensitive data are erased as soon as possible.

Attack and Exploitation

SGX remote attestation relies on the SGX sealing mechanism to encrypt and store the attestation key on disk. An attacker with an attestation key can sign arbitrary enclave measurements,

which are the SHA-256 of the enclave code. Using Foreshadow, the attacker can steal the quote enclave sealing key. After unsealing the attestation key, the attacker can sign rogue enclave measurements and forge proofs to Intel and external parties that the enclave runs on genuine hardware (see the work given by Van Bulck *et al.*[12] for details).

Impact

The ability to spoof remote attestation responses has profound consequences. Attestation is typically the first step to establish a secure communication channel, for example, via an authenticated Diffie–Hellman key exchange.[11] 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 and 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 eliminating the ability to trust remote SGX-based cloud services. For instance, an end-user might be fooled to reveal his private crypto currency key to a remote SGX-based wallet while in fact the private key is intercepted by the adversary. Apart from such confidentiality concerns, adversaries can also fabricate arbitrary remote SGX computation results.

BROADER IMPLICATIONS OF FORESHADOW / L1TF

We reported our Foreshadow-SGX findings to Intel in January 2018, describing how we defeated enclave memory isolation, sealing, and attestation guarantees (CVE-2018-3615). Subsequent investigation by Intel[4] identified the root cause for Foreshadow as the L1TF vulnerability. Unfortunately, L1TF has much broader and more direr consequences than leaking enclave memory, for it essentially allows dumping the entire contents of the L1 data cache, regardless of the owner of the data. In particular, Intel identified two closely related variants of Foreshadow, which we collectively call Foreshadow-Next Generation (Foreshadow-NG).[6] At a high level, Foreshadow-NG might be exploited by unprivileged applications to access kernel memory (CVE-2018-3620) or by malicious guest virtual machines to access memory belonging to the hypervisor and other guest machines (CVE-2018-3646).

Importantly, where previous Meltdown[1] attacks access unauthorized supervisor data *within* the attacker's virtual address space, Foreshadow-NG variants access unauthorized physical memory locations that are *not* mapped in the attacker's virtual address space. As such, Foreshadow-NG fully escapes the virtual memory sandbox—page table isolation is no longer sufficient to prevent unauthorized memory access.

Foreshadow and other speculative execution attacks exploit subtle microarchitectural races and, therefore, are highly elusive.

Crucially, page table isolation mitigations that are effective against Meltdown do not prevent Foreshadow-type L1TF attacks. Foreshadow, therefore, brings a paradigm shift in the way we should think about mitigating Meltdown-like threats: merely unmapping secrets from an untrusted application's address space is an insufficient countermeasure.

Final thoughts

Foreshadow and other speculative execution attacks exploit subtle microarchitectural races and, therefore, are highly elusive. For example, we found that seemingly insignificant variations of the covert channel implementation may cause the attack to succeed or fail. Unfortunately, due to the scarcity of public information about the processor architecture, we can only guess what real architectural flaws underlie the attack. Yet, the higher level insight is independent of the hardware implementation: speculation across memory protection boundaries is dangerous in the presence of hardware side channels, and should be avoided or explicitly safe-guarded against potential attacks.

ACKNOWLEDGEMENTS

This work was supported in part by the Research Fund KU Leuven, the Technion Hiroshi Fujiwara cyber security research center, the Israel

cyber bureau, the National Science Foundation (NSF) under Awards 1514261 and 1652259, the financial assistance award 70NANB15H328 from the U.S. Department of Commerce, the 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. The work of J. Van Bulck and R. Strackx was supported by a grant from the Research Foundation—Flanders (FWO).

REFERENCES

1. P. Kocher *et al.*, "Spectre attacks: Exploiting speculative execution," *in Proc. 40th IEEE Symp. Security Privacy*, 2019.
2. M. Lipp *et al.*, "Meltdown: Reading kernel memory from user space," in *Proc. USENIX Security Symp.*, 2018, pp. 973–990.
3. Intel, "Speculative execution side channel mitigations." 2018. [Online]. Available: https://software.intel.com/security-softwareguidance/api-app/sites/default/files/336996Speculative-Execution-Side-ChannelMitigations.pdf
4. Intel, "Deep dive: Intel analysis of L1 terminal fault." 2018. [Online]. Available: https://software.intel.com/security-software-guidance/insights/deep-dive-intel-analysis-l1-terminal-fault
5. Y. Xu, W. Cui, and M. Peinado, "Controlled-channel attacks: Deterministic side channels for untrusted operating systems," in *Proc. IEEE Symp. Security Privacy*, 2015, pp. 640–656.
6. O. Weisse *et al.*, "Foreshadow-NG: Breaking the virtual memory abstraction with transient out-of-order execution," 2018. [Online] Available at: https://foreshadowattack.eu/foreshadow-NG.pdf
7. Y. Yarom and K. Falkner, "FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack," in *Proc. 23rd USENIX Security Symp.*, 2014, pp. 719–732.
8. S. Gueron, "Memory encryption for general-purpose processors," *IEEE Security Privacy*, vol. 14, no. 6, pp. 54–62, Nov. 2016.
9. Intel, "SGX SDK for Linux." 2018. [Online]. Available: https://01.org/sites/default/files/documentation/intel_sgx_sdk_developer_reference_for_linux_os_pdf.pdf
10. J. Van Bulck, F. Piessens, and R. Strackx, "SGX-Step: A practical attack framework for precise enclave execution control," SysTEX, Shanghai, China, 2017.
11. F. McKeen *et al.*, "Innovative instructions and software model for isolated execution," in *Proc. 2nd Int. Workshop Hardware Archit. Support Security Privacy*, 2013, Paper 10.
12. J. Van Bulck *et al.*, "Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution," in *Proc. 27th USENIX Security Symp.*, 2018, pp. 991–1008.

Jo Van Bulck is currently a PhD student in computer science at imec-DistriNet, Katholieke Universiteit Leuven, Belgium. Contact him at jo.vanbulck@cs.kuleuven.be.

Marina Minkin is currently a PhD student in computer science and engineering at the University of Michigan. The work was performed when she was with Technion—Israel Institute of Technology. Contact her at minkin@umich.edu.

Ofir Weisse is currently a PhD student in computer science and engineering at the University of Michigan. Contact him at oweisse@umich.edu.

Daniel Genkin is an assistant professor of computer science and engineering at the University of Michigan. Contact him at genkin@umich.edu.

Baris Kasikci is an assistant professor of computer science and engineering at the University of Michigan. Contact him at barisk@umich.edu.

Frank Piessens is a professor of computer science at imec-DistriNet, Katholieke Universiteit Leuven, Belgium. Contact him at frank.piessens@cs.kuleuven.be.

Mark Silberstein is an associate professor of electrical engineering at Technion—Israel Institute of Technology. Contact him at mark@ee.technion.ac.il.

Thomas F. Wenisch is an associate professor of computer science and engineering at the University of Michigan. Contact him at twenisch@umich.edu.

Yuval Yarom is a senior lecturer in computer science at the University of Adelaide and Data61, CSIRO. Contact him at yval@cs.adelaide.edu.au.

Raoul Strackx is a postdoctoral research fellow in computer science at imec-DistriNet, Katholieke Universiteit Leuven, Belgium. Contact him at raoul.strackx@cs.kuleuven.be.