CS 305/341 Project Report

Foreshadow

Breaking Intel's SGX using speculative execution

Akkapaka Saikiran, Aniruddha Chidar, Saumya Goyal,
Shalabh Gupta, Shashank Roy

{180050005, 180050008, 180050092, 180050095, 180050097}

Abstract

Speculative Execution is an optimization pivotal to the performance of all modern day processors. However it renders the CPU vulnerable to attacks, and Intel's SGX presents enclaved memory which was were resistant to such attacks and acted as Trusted Execution Environments (TEEs). But Foreshadow is a software-only microarchitectural attack which dismantles the protection promised by SGX and manages to reliably leak plaintext enclaved secrets by augmenting a flush-and-reload attack with a page table trick.

1 Introduction

High-speed processors nowadays have lots of optimizations built into them which are essential to their performance. One of these optimizations is speculative execution. In essence, the processors make certain assumptions, for instance, the value of a register, or a decision at a branch. If it finds out its assumptions are invalid, it rolls back the microarchitectural state so that uncommitted architectural state remains same and things go on as they were supposed to, barring some wastage of CPU cycles. But recent work in this area has uncovered that speculative execution has security implications. Meltdown [1] is an attack discovered in early 2018 which (on most Intel and some ARM

processors) allows user applications to read the contents of kernel memory. In the pre-meltdown era, software security was mostly focused at the application (source code) level, i.e. attackers go and exploit source code vulnerabilities before they are being fixed. Meltdown allows attackers to exploit perfectly safe (formally verified) source code and read secrets due to a bug in the implementation of hardware. This is quite worrying and requires a paradigm shift in the way people think about security [2].
However, there was still some good news. There was one fortress in the CPU which was unaffected by Meltdown [3]. We're talking about SGX (Software Guard Extensions), introduced by Intel in 2013 [4]. They provide a functionality called enclaves which remain immune to Meltdown due to a feature called abort page semantics. We discuss SGX in greater detail in section SGX.
Foreshadow, dubbed by Intel as L1TF (L1 Terminal Fault), is a speculative execution attack which affects modern Intel Processors for reasons similar to those of Meltdown.
In this report, we examine each of the above mentioned issues. First we present an analysis of how Meltdown works, then we follow it up with the security features of SGX, and finally we show how Foreshadow manages to breach this security. We conclude by discussing some mitigations.

2 Meltdown

First we present an overview of the meltdown subroutine as used in the execution of Foreshadow. The actual meltdown attack has several details different from what we describe here and these details are specific to the execution of Foreshadow. Successfully executing the meltdown subroutine on a processor involves three steps:

- Unauthorized access
- Transient out-of-order execution
- Execution of fault handler

2.1 Unauthorized Access

Assume that we have a pointer to the unauthorized location that we want to access. Denote the pointer by `secret_pointer`. Since the location pointed to by the pointer is unauthorized, dereferencing it is illegal and will cause the hardware to trap to the OS. However, the first step in Meltdown does exactly that, it dereferences the secret pointer.

2.1.1 Delayed validity checking

The L1 caches on x86 processors are VIPT (Virtually Indexed Physically Tagged) caches. This allows the processor to access data from the L1 cache even when only partial translation of the virtual address is completed by the MMU. This reduces the access time of data from the L1 cache. As a further optimization to this, the MMU delays checking the validity of the memory access to a point after it has translated the

required portion of the virtual address. This makes sense since in a usual scenario we would not expect a lot of illegal memory accesses. Due to this the L1 cache is able to forward the data to the CPU and the pipeline resumes execution with this data. The value obtained from the cache is denoted by `v` here onwards.

2.2 Transient out-of-order execution

The following instructions in the instruction pipeline start executing while MMU is checking the valid bits. These instructions consist of accessing the `i`th index of an oracle array. This array is entirely flushed from the L1 cache at the time of execution of this instruction. Additionally, the address translations for this array should be present in the TLB. The latter allows our out-of-order phase to complete quickly before the processor realises it allowed an illegal memory access. Array access brings the cache line corresponding to the accessed memory location into the L1 cache. This opens the possibility of comparing access times to various oracle indices to deduce the value returned from the cache.

The index of the array accessed above is in this case a constant multiple of `v`. This constant is set to be big enough so as not to get false-positives due to prefetching.

```
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.

Figure 1: Transient Execution

2.3 Fault handler

This is when the processor finally realizes that it has allowed us to work on illegal data. It quickly flushes the entire pipeline and restores the architectural state to the instruction before the illegal access instruction. Once the processor ensures that it has prevented the program from having direct access to the illegal location, it traps to the OS. What the processor does not do however, is flush the L1 cache due to which the oracle entry corresponding to the fetched data is still present in L1 cache.

After the processor traps to the OS, the OS executes the user space trap handler for incorrect memory access. The fault handler then simply goes over the oracle array, measuring the time it takes to access each entry. The only entry which it is able to

access fast is the one present in the L1 cache: the entry corresponding to the data fetched from the secret address. The fault handler is thus able to deduce the value of the byte accessed from the unauthorized location and meltdown attack has been successfully executed.

This type of side channel timing attack that compares access times from the main memory and the L1 cache in order to deduce unauthorized data is called a FLUSH+RELOAD attack.

3 SGX: High Melting Point

Software Guard Extensions (SGX) is a set of security-related instruction codes that are built into some modern Intel CPUs. They allow user-level as well as operating system code to define private regions of memory, called enclaves. SGX involves encryption by the CPU of a portion of memory which mitigates the meltdown attack by redirecting to an abort page as any other execution context reads an encrypted form.

An SGX-enabled CPU 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. Have a look at [5].

3.1 Enclaves

The purpose of an enclave is to limit internal access to a portion of certain data. It is necessary when the set of resources differs from those of the general surroundings. They are standalone assets that do not interact with other information systems or networks. Enclave protection tools can be used to provide protection within specific security domains. It's contents are protected and unable to be either read or saved by any process outside the enclave itself, including processes running at higher privilege levels. It is decrypted by the CPU from inside itself. The code and data in the enclave utilize a threat model in which the enclave is trusted but no process outside it can be trusted (including the operating system itself and any hypervisor), and therefore all of these are treated as potentially hostile. It is like a mirror house. You can't see inside from out of the house (except your reflection). You need to enter it in order to view it. sgx_create_enclave() creates the enclave on which we perform Foreshadow.

3.2 Abort Page Semantics

Attempts to read SGX data from outside the enclave receives special handling from the processor. They are not subjected to page faults, instead -1 is returned on all reads and writes to enclave memory are completely ignored. This special handling is referred to as "abort page semantics". This is why meltdown, despite being a speculative execution attack, is unable to read enclave memory because it relies on a page fault happening

leading to an exception. But any transient instruction after the rogue data fetch will never see the enclave secret but always the value -1. Have at look at [6].

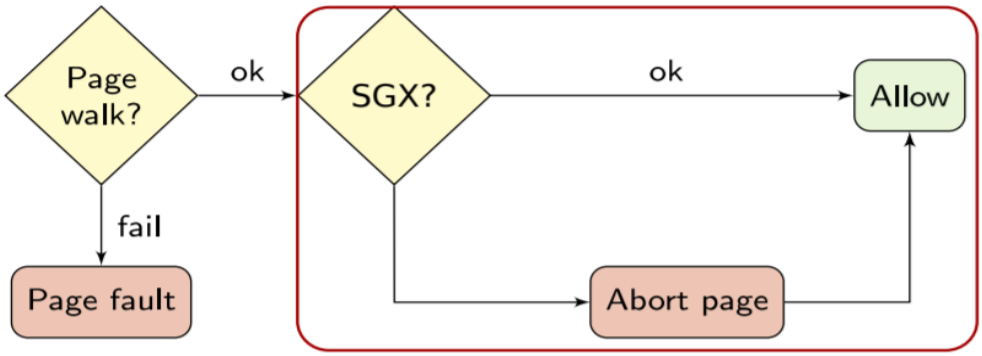

Figure 2: Abort Page Semantics

We will now see how Foreshadow bypasses this security measure cleverly.

4 Foreshadow

Abort page semantics apply only if the "present" bit in the page table entry of the secret memory being read is set to True. In this case page fault does not occur. In Foreshadow, the page table entry of the secret memory is unmapped to deliberately let a page fault occur to let the transient window execute the instructions following the rogue fetch. Being able to read enclave memory is one of the major differences between meltdown and Foreshadow.

4.1 Unmapping page table entry

In Linux, we can make use of an unprivileged system call mprotect() to revoke all access permissions to the enclave page we wish to read by setting it as "not present".

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

This piece of code simply clears the present bit of the page table entry corresponding to the address and now any accesses to this page will eventually lead to a page fault and the user fault handler will be called since the user has himself changed the permissions in the page table entry.

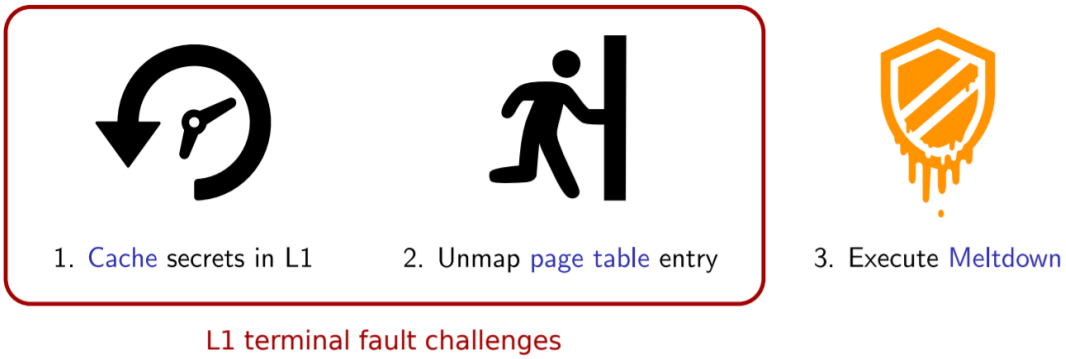

Figure 3: Foreshadow Overview

4.2 Caching the secret

In [7] enclave secrets do not reach the transient instructions if they are not already residing in the L1 cache during the illegal access. We think the reason for this is time taken to get the secrets from enclave memory is significantly more than time taken to access L1 cache. Therefore the small speculation window before the CPU decides to roll back the rogue data fetch results does not allow secrets from enclave memory to be fetched. Another reason could be 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. This was confirmed by Intel as well and they dubbed the attack 'L1 terminal fault' because of this reason. "Prefetching" secrets into L1 cache solves this problem. To bring the secrets into L1 cache, the victim enclave is executed to cache plaintext secrets.

4.3 Last step of Foreshadow : Flush and Reload

Each time we enter or exit the enclave, TLB entries are flushed. This means that accessing the oracle slots in the transient execution will result in an expensive page table walk Therefore after executing the enclave and unmapping the page table entry, we reload all the oracle slots back in the TLB. This is required for fast access of the oracle during the transient execution phase. Furthermore, we ensure that none of the oracle slots are already present in cache.

After this, the meltdown flush-and-reload attack is carried out as described above to read the enclave secrets.

5 Mitigating Foreshadow

Even though Foreshadow employs a side channel attack step, mitigation of Foreshadow through side-channel hardening techniques offers little protection. The reason is that most of such mitigation techniques rely on TSX support which might not be available in

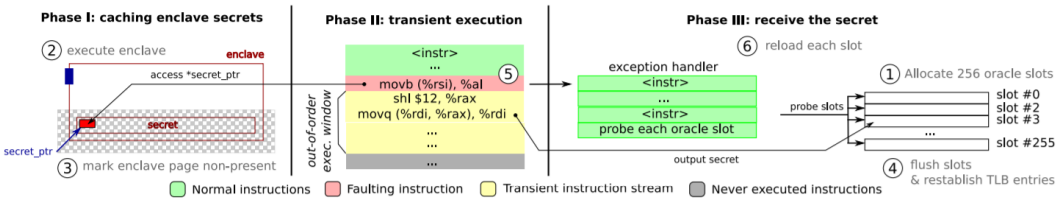

Figure 4: Basic overview of the attack to extract a byte from SGX enclave

every SGX capable processor. Even if it is available, Foreshadow attackers can circumvent attack detection techniques. An example is circumventing detection of suspicious page fault and interrupt rates through Hyper Threading based Foreshadow variants.

One other possibility for Foreshadow mitigation is for Intel to patch hardware-level vulnerabilities. Modern Intel processors employ silicon-based changes to the processor which makes the processor non-vulnerable to the Foreshadow attack. Intel has also released microcode patches which mitigate Foreshadow.

Since it was found that Foreshadow requires enclave data to be necessarily present in the L1 cache, hardware-software co-design mitigation strategies are also possible. For the simple variant of Foreshadow that we studied, such a mitigation strategy should ensure that L1 cache is flushed upon each enclave enter/exit.

6 Challenges

In this section we outline the several difficulties faced by us in this project and the means and methods by which we solved them.

6.1 Understanding the theory

When we started out, we had yet to cover a substantial portion of out Operating Systems and Computer Architecture courses. Due to this, we faced a fair amount of trouble understanding [7]. We had to refer to a couple of blogs ([5], [6]) to get a better grasp of various non-trivial concepts. Conversion of instructions to μ-ops (micro-operations), storing micro-architectural state in the re-order buffer (ROB), and out-of-order execution of instructions were totally new ideas at the time. Further, VIPT (Virtually Indexed Physically Tagged) and multi-level caches were difficult to comprehend. It took multiple reads of the paper coupled with patience to get a hang of things, but over time we gained enough confidence to move on and try to read the code of the Proof-of-Concept (PoC) [8].

6.2 Understanding the code

The codebase for the demo was quite large thus we had a hard time in understanding what was going on. We couldn't figure out how the program control shifted from kernel drivers to the calling conventions for secure hardware primitives. We overcame this difficulty through long hours of discussion and reading the code together as a team.

6.3 Executing the code

To start off, setting up the prerequisites for the installation was a painful process. SGX was disabled by default in BIOS and this caused many cryptic errors. Once this was resolved, it turned out our laptops did not support TSX (Transactional Synchronization Extensions) which were imperative to get good results on the PoC which was quite crude. To add to our woes, our devices also had microcode patches which flushed the L1 cache on enclave entry/exit so the attack was unsuccessful.
Thus there were multiple hardware requirements which we could not meet with any of our devices. We tried, in vain, to find alternative resources like other laptops and cloud-based virtual machines, but they were all insufficient. Thus we are extremely grateful for the timely and overwhelming assistance on the GitHub issue [9] raised by us on the repository. Jo Van Bulck went out of his way to add a commit which allowed us to simulate enclaves and thus demonstrate Foreshadow successfully in this simulated environment.

7 Conclusion

We presented Foreshadow, an efficient transient execution attack that completely compromises the confidentiality guarantees pursued by contemporary Intel SGX technology. It is a novel attack methodology that abuses the untrusted OS's control over the page table to provoke terminal faults when dereferencing enclave memory.

It is superior to meltdown and various other attacks. It occurs due to the flaw in microarchitectural implementations and not due to the underlying hardware design.

8 Acknowledgement

We are grateful for the help received from Jo Van Bulck, one of the chief investigators who found the Foreshadow attack, in executing L1TF on our laptops successfully. Our demo is derived largely from the official Foreshadow PoC [8].

References

[1] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[2] Jo Van Bulck. Usenix security '18 - foreshadow: Extracting the keys to the intel sgx kingdom. https://www.youtube.com/watch?v=fEV6eA9o21o&ab_channel=USENIX.

[3] Ambuj Kumar. Meltdown melted down everything, except for one thing. https://www.ibtimes.co.uk/meltdown-melted-down-everything-except-one-thing-1663785.

[4] Intel. Intel® software guard extensions programming reference. https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf.

[5] Medium. Meltdown and spectre and what it means for intel sgx. https://medium.com/anjuna-security/meltdown-spectre-and-what-it-means-for-intel-sgx-492ec9c8b689.

[6] RedHat. Understanding l1 terminal fault aka foreshadow: What you need to know. https://www.redhat.com/en/blog/understanding-l1-terminal-fault-aka-foreshadow-what-you-need-know.

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

[8] Jo Van Bulck. Foreshadow: Extracting the keys to the intel sgx kingdom with transient out-of-order execution. https://github.com/jovanbulck/sgx-step.

[9] Enclave creation failure #34. https://github.com/jovanbulck/sgx-step/issues/34.