Meltdown Attack

Project Report

180050003 Abhinav Kumar
180050068 Nimay Gupta
180050090 Samarth Singh
180050096 Shashank Agrawal

1 December 2020

Contents

Abstract

One of the important abstractions provided by the OS and hardware combined is the concept of virtual memory and memory isolation i.e kernel address and other process's memory separated from user process's memory and is inaccessible by user. Meltdown attack breaks all these security guarantees by exploiting out-of-order execution to read arbitrary kernel memory locations. Out-of-order execution is an important performance feature present in wide range of processors. We discuss in this project how Meltdown enables an adversary to read arbitrary kernel memory and a proof-of-concept code demonstrating the attack as well mitigations (software based) such as KAISER and KPTI.

1 Introduction: Meltdown

One of the important ideas proposed by OS developers when era of batch computing ended and the computing environments started running multiple applications from different users at the same time was the concept of memory isolation. On modern processors, the isolation between the kernel and user processes is typically realized by a supervisor bit of the processor that defines whether a memory page of the kernel can be accessed or not. The basic idea is that this bit can only be set when entering kernel code and it is cleared when switching to user processes. This allows OS to map entire kernel memory into the address space of each process for smooth transition between kernel and user mode.

Out-of-order execution is an important performance feature of today's processors which allows an instruction sequence to be executed in an non-sequential manner. However, this optimisation allow an unprivileged process to load privileged memory into a temporary register and may even perform computations on that value. On ISA level, this does not poses any security issue as no ISA level change is observed after discarding illegal access but it leaves micro-architectural side effects such as influence on cache which can be detected using side-channels. As a result, an attacker can dump the entire kernel memory by reading privileged memory in an out-of-order execution stream, and transmit the data from this elusive state via a micro-architectural covert channel (e.g., Flush+Reload) to the outside world.

2 Background

2.1 Address Space

To support memory isolation for processes, OS with help from hardware support virtual addressing where virtual addresses are converted to physical address during memory access. This provides illusion to the process of having whole memory available to it while in reality, OS does all the hard job and provides clean abstraction to the process. Virtual address space is divided into

pages which is mapped to physical memory and translation tables do the translation and also check the memory access privileges such as readable, writable etc. Each process has its own page table and a CPU register holds the pointer to the page table which is changed on context switch for it's updated value. Virtual address space of each process is divided into user and kernel part for easy transitioning of modes and the kernel part can be accessed only if CPU is in kernel mode. OS needs to maintain list of free pages in physical memory for allocating it to user processes. Consequently, entire physical memory is typically mapped into kernel part of address space. On Linux, entire physical memory is directly mapped to a pre-defined virtual address.

2.2 Out-of-Order Execution

Out-of-order execution is an optimisation technique allowing maximum utilisation of the execution units of a CPU core. To do so, CPU follows dataflow model and fires the instructions as soon as required data is available, rather than waiting for the previous instr to finish. However, the results of all instructions are committed in the order in which instructions were initially thus ensuring correct program execution. To achieve this optimisation, CPU compromises with protection bit checking for each memory access in out-of-order exec as checking protection from page table for each memory access will cause the execution to be inherently sequential. However, this was not realised earlier that it may lead to memory leakage to an adversary.

2.3 Cache

In order to bridge the gap between CPU and main memory access speed and speed up memory accesses and address translations, multiple levels of small buffers called caches are used in modern processors. CPU caches hide the slow memory access by main memory by storing frequently used data in smaller and faster buffers. Consequently, address translations are also cached in caches.

Cache side channel attacks exploit timing differences introduced due to a memory access miss or hit. In this project, we use Flush-Reload which has more noise resistance and simpler implementation as compared to Evict-Reload. In Flush-Reload, attacker frequently flushes the target address and measures the time taken to reload the data and the secret which was accessed by victim process and cached will be accessed faster leading to adversary discovering the secret.

A special use case of a side-channel attack is a **covert channel**. Here the attacker controls both, the part that induces the side effect, and the part that measures the side effect. This can be used to leak information from one security domain to another, while bypassing any boundaries existing on the architectural level or above.

3 Building Blocks of the Attack

3.1 Toy example

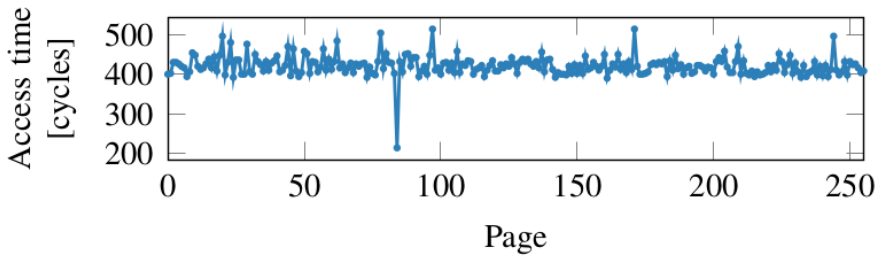
In the code in Figure 1, ideally the execution of instr at line 3 should not be possible as after raising exception, the control goes to kernel and the process most likely exits. However, due to the out-of-order execution, the CPU might have already executed the following instructions as there is no dependency on the instruction triggering the exception. This is illustrated in Figure 2.

Figure 1:

```
1 raise_exception();
2 // the line below is never reached
3 access(probe_array[data * 4096]);
```

.

Figure 2:



We call such an instruction, which is executed out of order and leaving measurable side effects, a transient instruction. Furthermore, we call any sequence of instructions containing at least one transient instruction a transient instruction sequence. In order to leverage transient instructions for an attack, the transient instruction sequence must utilize a secret value that an attacker wants to leak. This forms the first building block for meltdown. The second building block is to transfer the microarchitectural side effect to an architectural change by further processing.

3.2 Executing Transient Instructions

Executing the transient instruction is the first step of Meltdown. Transient instructions introduce an exploitable side channel if their operation depends on

a secret value. We focus on addresses that are mapped within the attacker's process, i.e., the user-accessible user space addresses as well as the user-inaccessible kernel space addresses. Accessing user inaccessible pages triggers exception which is SIGSEGV in this case and generally terminates the process. The attacker copes with this using exception handling.

Exception Handling: Since illegal memory access result in segmentation fault, handling sigsegv for attack is necessary. One important point is that SIGSEGV unlike SIGTERM and SIGKILL, can be handled and hence, adversary may continue after illegal memory access without ending the process. In our project we use this approach. However, another approach more robust than exception handling could be to fork a child process and raise exception in child and the parent can continue even after child process has been terminated.

3.3 Building a Covert Channel

Executing transient instructions is the sending end of the covert channel. The receiving end involves transferring the microarchitectural change to an architectural change thus deducing the secret.

We leverage cache attack techniques to transfer the cache state into an architectural state using Flush+Reload. After the transient instruction accessed some address, the address is cached for subsequent accesses. The receiver can then monitor which address was accessed by measuring the time taken for access. As in Fig.1, data is of size 1 byte and when accessed, is cached. In Fig.2, the attacker iterates over all the possible values of 8 bit data (256 in total) and the access time of value 84 is much less denoting its presence in cache.

4 Proof of Concept

Figure 3: Our proof-of-concept

```
44  #define MELTDOWN
45    asm volatile("1:\n"
46                 "movzx (%%rcx), %%rax\n"
47                 "shl $12, %%rax\n"
48                 "jz 1b\n"
49                 "movq (%%rbx,%%rax,1), %%rbx\n"
50                 :
51                 : "c"(phys), "b"(mem)
52                 : "rax");
53
```

4.1 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 parallel to translating a virtual address into a physical address, the CPU also checks the permission bits of the virtual address. As the hardware-based isolation is considered to be secure, modern os always map entire kernel into the virtual address of the process. And, normally, the user space cannot read the content of kernel address. However, out-of-order execution still executes instruction in the small window between illegal access and exception raising.

In meltdown attack setting, in Fig3. line 46, **MOVZX** loads the least significant byte at the kernel address pointed by **RCX** into register **RAX**.

During the retirement, any interrupts and exceptions that occurred during the execution of the instruction are handled. Thus, if the MOVZ instruction that loads the kernel address is retired, the exception is registered, and the pipeline is flushed to eliminate all results of subsequent instructions which were executed out of order. However, there is a race condition between raising this exception and our attack step 2 as described below.

4.2 Transmitting the Secret

As already discussed, the transient instruction sequence needs to encode the secret into the microarchitectural cache state. To do so, we have already allocated a probe array of size 256*4096 whose address is stored in $RBX. The reason for particular size of probe array will become clear as we discuss step 3. Before accessing the secret, we ensure that whole probe array is not cached as this step is particularly important in decoding the secret. In line 47 of Fig3. the secret value is multiplied by page size i.e. 4096 and then in line 49, first line of the page in probe array indexed by $RAX is accessed hence is cached too.

4.3 Receiving the Secret

When the transient instruction sequence of step 2 is executed, exactly one cache line of the probe array is cached. The position of the cached cache line within the probe array depends only on the secret which is read in step 1. Thus, the attacker iterates over all 256 pages of the probe array and measures the access time for every first cache line (i.e., offset) on the page. The number of the page containing the cached cache line corresponds directly to the secret value.

Iterating over 256 pages of the probe array is performed during exception handling as in Fig.4 which handles the SIGSEGV raised during execution of line 46 in Fig.3. In normal handling of sigsegv, after handling the exception, the control again returns to the instruction which caused the fault. To avoid that, we use longjmp and setjmp (similar to jal and jr in assembly language). In line 83, setjmp(jump_buffer) returns 0 and links jump_buffer to this address and then MELTDOWN causes SIGSEGV which is handled by line 72 which after

Figure 4: Exception Handling

```
64  // SEGMENTATION FAULT HANDING ---------------------------------------------
65  static void unblock_signal(int signal_number __attribute__((__unused__))) {
66    sigset_t signal_set;
67    sigemptyset(&signal_set);
68    sigaddset(&signal_set, signal_number);
69    sigprocmask(SIG_UNBLOCK, &signal_set, NULL);
70  }
71
72  static void segfault_handler(int signal_number) {
73    (void)signal_number;
74    unblock_signal(SIGSEGV);
75    longjmp(jump_buffer, 1);
76  }
77
78  int __attribute__((optimize("-Os"), noinline)) read_signal_handler() {
79      uint64_t retries = meltdown_config.retries + 1;
80      uint64_t start = 0, end = 0;
81
82      while (retries--) {
83          if (!setjmp(jump_buffer)) {
84              MELTDOWN;
85          }
86          int i;
87          for (i=0; i<256; i++) {
88              if (flush_reload(mem + (i<<12))) {
89                  if (i >= 1) return i;
90              }
91              sched_yield();
92          }
93          sched_yield();
94      }
95      return 0;
96  }
```

unblocking the signal, calls longjmp which jumps to address previously set by setjmp i.e. line 83 and returns 1 thus "IF" condition evaluates to 0 and line 86 executes and adversary iterates over probe array, measuring the time of access using flush_reload function and if index ¿ 0, it is the secret byte value hence returned. The loop is continued when the secret value is 0 (Discussed in greater detail in section 5).

Note that it is important to access probe array with successive indices 1 page apart as it eliminates false positives due to the prefetcher, as the prefetcher cannot access data across page boundaries.

5 Mitigations and Limitations

5.1 Inherent bias towards 0

While CPUs generally stall if a value is not available during an out-of-order load operation, CPUs might continue with the out-of-order execution by assuming a value for the load. We observed that the illegal memory load in our Meltdown implementation (line 46 in Fig.3) often returns a '0'. The reason for this bias to '0' may either be that the memory load is masked out by a failed permission check, or a speculated value because the data of the stalled load is

not available yet. In an unoptimized version, the probability that a value of '0' is erroneously returned is high. Consequently, our Meltdown implementation performs a certain number of retries (line 82 in Fig. 4) when the code in Fig.3 results in reading a value of '0' from the Flush-Reload attack.

5.2 Hardware Fixes

As Meltdown exploits the out-of-order execution, a trivial countermeasure would be to completely disable out-of-order execution. However, the performance impacts would be devastating, as the parallelism of modern CPUs would not be leveraged anymore. Hence, this is not a viable solution. Meltdown is kind of a race condition between the fetch of a memory address and the corresponding permission check for this address. Serializing the permission check and the register fetch can prevent Meltdown, as the memory address is never fetched if the permission check fails. However, this involves a significant overhead to every memory fetch, as the memory fetch has to stall until the permission check is completed.

A more plausible solution would be to introduce a hard split of user space and kernel space. This could be enabled optionally by modern kernels using a new hard split bit in a CPU control register. If the hard split bit is set, the kernel has to reside in the upper half of the virtual address space, and the user space has to reside in the lower half of the virtual address space. With this hard split, a memory fetch can immediately identify whether the destination would violate a security boundary, as the privilege level can be directly derived from the virtual address without any look-up. The performance impact of such a solution would be minimal. Furthermore, the backward compatibility is ensured, since the hard-split bit is not set by default and the kernel only sets it if it supports the this feature.

5.3 KASLR: Kernel Address Space Layout Randomization

Modern operating system kernels employ address space layout randomization (ASLR) to prevent control-flow hijacking attacks and code-injection attacks. While kernel security relies fundamentally on preventing access to address information, recent attacks have shown that the hardware directly leaks this information. Strictly splitting kernel space and user space has recently been proposed as a theoretical concept to close these side channels. However, this is not trivially possible due to architectural restrictions of the x86 platform.

ASLR can be used to make some kernel addresses unpredictable for an attacker. Knowledge of virtual/physical address information can be exploited to bypass KASLR. Side channel attacks targeting the page translation caches provide information about virtual and physical addresses to the user space.

With KASLR the direct-physical map offset is randomized, thus the attacker is required to obtain the offset before mounting the Meltdown attack. If the attacker can successfully obtain a value from a tested address, the attacker can

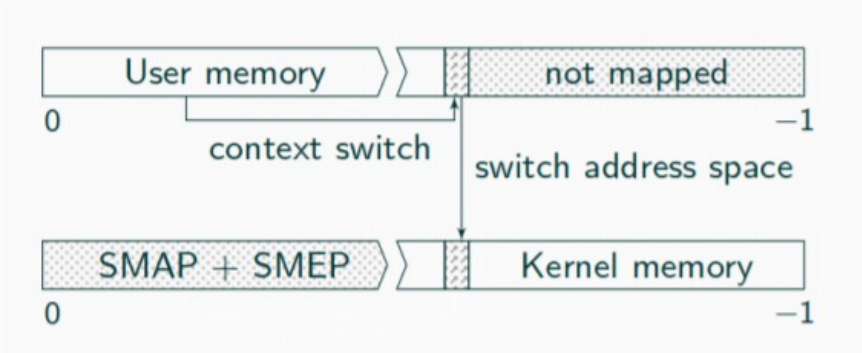
Figure 5: KASLR

proceed to dump the entire memory from that location. This allows mounting Meltdown on a system despite being protected by KASLR within seconds.

5.4 KAISER: Kernel Address Isolation

The idea of Stronger Kernel Isolation is to unmap kernel pages while the user process is in user space and switch to a separated kernel address space when entering the kernel. This prevents all microarchitectural attacks on kernel address space information on recent systems.

Figure 6: Shadow Address Space and Kernel Address Space



KAISER uses a shadow address space paging structure to separate kernel space and user space. The lower half of the shadow address space is synchronized between both paging structures. KAISER reduces the number of overlapping pages between user and kernel address space to the absolute minimum required

to run on modern x86 systems. It enforces a strict kernel and user space isolation, the hardware does not hold any information about kernel addresses while running user processes.

The main side-channel defense in KAISER is based on the separate shadow address spaces. This inadvertently protects from Meltdown attack as well. The invalid memory access instruction in Meltdown will not be able to load data from the inaccessible kernel address and thereby the transient instructions do not succeed. Thus, KAISER successfully eliminates the leakage.

6 References

References

[1] Meltdown and Spectre `https://meltdownattack.com/`

[2] Meltdown official paper `https://arxiv.org/abs/1801.01207`

[3] Jann Horn: Google Project Zero `https://googleprojectzero.blogspot.com/`

[4] Official github repository `https://github.com/IAIK/meltdown`

[5] StackOverFlow `https://stackoverflow.com/questions/48306932/whats-the-purpose-of-these-assembly-register-modifiers`