Abstract

- Meltdown exploits side effects of out-of-order execution to read arbitrary kernel-memory locations
- Breaks all security assumptions given by address space isolation as well as paravirtualized environments
- Enables an adversary to read memory of other processes or virtual machines without any permissions or privileges
- KAISER developed to address KASLR issues also (inadvertently) impedes meltdown but you pay a performance penalty to deploy (current situation on most all platforms)
- Affects all Intel micro-architectures since 2010 and some others
- Tests show ability to dump entire memory of system at 503KB/s
Introduction

Memory Isolation realized via supervisor bit associated with page table

CPU Mode set when entering kernel and cleared when switching to user process

Allows Kernel to be mapped into address space of every process and creates a fast transition from user process to the kernel

Out-of-order execution is an important performance feature of today’s processors in order to overcome latencies of busy execution units, e.g., a memory fetch unit needs to wait for data arrival from memory

But there is a flaw associated with how this is implemented that we will exploit

From a security perspective, one observation is particularly significant:
Out-of-order; vulnerable CPUs allow an unprivileged process to load data from a privileged (kernel or physical) address into a temporary CPU register. Moreover, the CPU even performs further computations based on this register value, e.g., access to an array based on the register value

In particular the value in the register register can be used to influence the cache providing a side channel to communicate the value read (For example using Flush+Reload)
Background

Out-of-order execution running operations speculatively

In this discussion, we refer to *speculative execution* in a more restricted meaning, where it refers to an *instruction sequence following a branch*, and use the term out-of-order execution to refer to any way of getting an operation executed before the processor has *committed* the results of all prior instructions.

We have been utilizing out-of-order execution in CPUs *since 1967!* (Intel didn't figure out how to implement it incorrectly and create this security flaw until around 2010)

*Simplified illustration of a single core of the Intel's Skylake microarchitecture.*

*The Reorder Buffer is responsible for register allocation, register renaming and retiring.*
The BIG IDEA TO GRASP is

CPUs usually do not run linear instruction streams, they have branch prediction units that are used to obtain an educated guess of which instruction will be executed next.

Branch predictors try to determine which direction of a branch will be taken before its condition is actually evaluated.

Instructions that lie on that path and do not have any dependencies can be executed in advance and their results immediately used if the prediction was correct.

If the prediction was incorrect, the reorder buffer allows to rollback by clearing the reorder buffer and re-initializing the unified reservation station.
**Address Spaces**

A virtual address space is divided into a set of pages that can be individually mapped to physical memory through a multi-level page translation table.

The translation tables define the actual virtual to physical mapping and also protection properties that are **used to enforce privilege checks, such as readable writable, executable and user-accessible**

The location of the *currently used translation table* is held in a special CPU register. (Intel CR3)

On each context switch, the operating system updates this register with the next process’ translation table address in order to implement per process virtual address spaces.

Each virtual address space itself is split into a user and a kernel part. While the user address space can be accessed by the running application, the kernel address space can only be accessed if the CPU is running in privileged mode (this is where the flaw will be revealed)

The kernel address space does not only have memory mapped for the kernel's own usage, but it also needs to perform operations on user pages, e.g., filling them with data.

Consequently, **the entire physical memory is typically mapped in the kernel !!!**

**On Linux and OS X,** this is done via a **direct-physical map**, i.e., the entire physical memory is directly mapped to a pre-defined chunk of kernel virtual address

**Windows is a little different in this design aspect**

Instead of a **direct-physical map**, Windows maintains a multiple so-called **paged pools, non-paged pools**, and the **system cache**. These pools are virtual memory regions in the kernel address space mapping physical pages to virtual addresses which are either required to remain in the memory (non-paged pool) or can be removed from the memory because a copy is already stored on the disk (paged pool). The system cache further contains mappings of all file-backed pages.

**Bottom Line is**

**Combined, these memory pools will typically map a large fraction of the physical memory into the kernel address space of every process. Thus, the same attack strategy used in Linux/OS X setting will work.**
KASLR

The exploitation of memory corruption bugs often requires the knowledge of addresses of specific data.

In order to impede such attacks, address space layout randomization (ASLR) has been introduced as well as nonexecutable stacks and stack canaries.

In order to protect the kernel, KASLR randomizes the offsets where drivers are located on every boot, making attacks harder as they now require to guess the location of kernel data structures.

This "seemed to work" until last February when we figured out how to break it. **Side-channel attacks allow to detect the exact location of kernel data structures**

**This is an essential piece of the puzzle we will need to create the Meltdown Attack**

**Cache side-channel attacks exploit timing differences** that are introduced by the caches.

Different cache attack techniques have been proposed and demonstrated in the past, including
Evict+Time
Prime+Probe
Flush+Reload.

**Flush+Reload attacks work on a single cache line granularity. These attacks exploit the shared, inclusive last-level cache (the L3 cache, NOTE it is shared among ALL cores)**
**This is the one we use to do Meltdown !!!**

No time for complete explanation of Flush+Reload (will do offline) but basic scheme is
An attacker frequently flushes a targeted memory location using the `clflush` instruction. By measuring the time it takes to reload the data, the attacker determines whether data was loaded into the cache by another process in the meantime.
The first building block of Meltdown is to make the CPU execute one or more instructions that would never occur in the executed path. we call it a transient instruction

While we can build meltdown attacks simply using exception handling in faults caused by forked processes which is applicable to all Intel CPUs I will discuss the more devious approach of using exception suppression because it lets us dump out the memory of the machine faster. This technique will require later processors with the TSX instruction.

Exception suppression. Rather than using a signal handler to catch an exception a different approach to deal with exceptions is to prevent them from being raised in the first place. Transactional memory allows to group memory accesses into one seemingly atomic operation, giving the option to roll-back to a previous state if an error occurs. If an exception occurs within the transaction, the architectural state is reset, and the program execution continues without disruption.
Meltdown consists of 3 steps:

**Step 1** The content of an attacker-chosen memory location, which is (suppose to be) inaccessible to the attacker, is loaded into a register.

**Step 2** A transient instruction accesses a cache line based on the secret content of the register.

**Step 3** The attacker uses *Flush+Reload* to determine the accessed cache line and hence the secret stored at the chosen memory location.
**Details of the Attack**

**Step 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, i.e., whether this virtual address is user accessible or only accessible by the kernel.

Meltdown exploits the out-of-order execution of modern CPUs, which still executes instructions in the **small time window** between the illegal memory access and the raising of the exception.

**Core sequence to create Meltdown Attack**

```
1 ; rcx = kernel address
2 ; rbx = probe array
3 retry:
4 mov al, byte [rcx]
5 shr rax, 0xc
6 jz retry
7 mov rbx, qword [rbx + rax]
```

In line 4 of the listing, we **load the byte value located at the target kernel address**, stored in the RCX register, into the least significant byte of the RAX register represented by AL.

The MOV instruction is fetched by the core, decoded into µOPs, allocated, and sent to the reorder buffer. There, architectural registers (e.g., RAX and RCX in the listing) are mapped to **underlying physical registers** enabling out-of-order execution.

Trying to utilize the pipeline as much as possible, subsequent instructions (lines 5-7) are already decoded and allocated as µOPs as well. The µOPs are further sent to the reservation station holding the µOPs while they wait to be executed by the corresponding execution unit. The execution of a µOP can be delayed if execution units are already used to their corresponding capacity or operand values have not been calculated yet.

When the kernel address is loaded in line 4, it is likely that the CPU **already issued the subsequent instructions as part of the out-of-order execution**, and that their corresponding µOPs wait in the reservation station for the content of the kernel address to arrive. As soon as the fetched data is observed on the common data bus, **the µOPs can begin their execution**.

When the µOPs finish their execution, they retire inorder, and, thus, their results are **committed** to the architectural state. During the retirement, any interrupts and exception that occurred during the execution of the instruction are handled.

Thus, if the MOV 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**.
**Step 2: Transmitting the secret.**

The instruction sequence from step 1 which is executed out of order has to be chosen in a way that it becomes a *transient instruction* sequence. If this transient instruction sequence is executed before the `MOV` instruction is retired (i.e., raises the exception), and the transient instruction sequence performed computations based on the secret, it can be utilized to transmit the secret to the attacker.

As already discussed, we utilize cache attacks that allow to build fast and low-noise covert channel using the CPU's cache. Thus, the transient instruction sequence has to encode the secret into the microarchitectural cache state.

We allocate a probe array in memory and ensure that no part of this array is cached. To transmit the secret, the transient instruction sequence contains an indirect memory access to an address which is calculated based on the secret (suppose to be inaccessible) value. In line 5 of Listing 2 the secret value from step 1 is multiplied by the page size, i.e., 4 KB. The multiplication of the secret ensures that accesses to the array have a large spatial distance to each other. This prevents the hardware prefetcher from loading adjacent memory locations into the cache as well.

Here, we read a single byte at once, hence our probe array is 256×4096 bytes, assuming 4KB pages.

Note that in the out-of-order execution we have a noise-bias towards register value ‘0’. We discuss the reasons for this later. However, for this reason, we introduce a retry-logic into the transient instruction sequence.

In case we read a ‘0’, we try to read the secret again (step 1). In line 7, the multiplied secret is added to the base address of the probe array, forming the target address of the covert channel. This address is read to cache the corresponding cache line. Consequently, our transient instruction sequence affects the cache state based on the secret value that was read in step 1.

Since the transient instruction sequence in step 2 races against raising the exception, reducing the runtime of step 2 can significantly improve the performance of the attack. For instance, taking care that the address translation for the probe array is cached in the TLB increases the attack performance.
**Step 3: Receiving the secret.**

In step 3, the attacker recovers the secret value (step 1) by leveraging a microarchitectural side-channel attack (i.e., the receiving end of a microarchitectural covert channel) that transfers the cache state (step 2) back into an architectural state.

As discussed previously, Meltdown relies on Flush+Reload to transfer the cache state into an architectural state.

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**

**OH YOU ARE SO SCREWED**
Dumping the entire physical memory.

By repeating all 3 steps of Meltdown, the attacker can dump the entire memory by iterating over all different addresses. However, as the memory access to the kernel address raises an exception that terminates the program, we use one of the two methods described in to handle or suppress the exception. (Catch the signal or better yet use TSX)

As all major operating systems also typically map the entire physical memory into the kernel address space in every user process, Meltdown is not only limited to reading kernel memory but it is capable of reading the entire physical memory of the target machine.

OH YOU ARE SO SCREWED
How To Fix It

Kaiser