Meltdown and Spectre

Began Bajrami, Rahmi El Mechri

June 26, 2022

Abstract

TODO

1 Introduction

2 Out of Order Execution

Out of order execution is the key feature that Meltdown vulnerability exploits on Intel's micro-architectures. Basically, what Meltdown does is possibile only because how Intel processors micro-architecture is designed.

Out of order is a technique used by almost every CPU both for Desktop and Server/Cloud machines, the main reason being the improvements on perfomance that it brings, allowing CPU to execute instructions in a different order than how the program was compiled, in order to avoid wasting of computational power. In this paper we refer to out-of-order as 'out-of-order issue out-of-order completion'.

2.0.1 Tomasulo's algorithm

For a better understaindg of the Intel's CPU architecture, here is a brief introduction to Tomasulo's algorithm which first introduced to techniques like register renaming, reservation station and common data bus (CDB), which allowed out-of-order execution.

> In 1967, Tomasulo developed an algorithm that enabled dynamic scheduling of instructions to allow out-of-order execution.

[1]

Tomasulo's reservation station allows instructions that operate on the same physical registers to rename registers (register renaming), i.e. duplicating register names in order to allow different instructions operate on the same register at the same time. This techinque solves read-after-write (True data dependeny, or RAW), write-after-read (Antidependency, or WAR) and write-after-write (WAW) hazards. Moreover, this lets the execution units use data values as soon as they are computed rather then reading value from a register, writing the result on the register and then, again, reading it. [1][2 wikipedia] All execution units are directly (and individually) connected to the reservation station via a common data bus (CDB), where operands of instructions are passed as soon as they're available. This is useful if an instruction is waiting for an operand that is not already on the register, so it can directly listen on the CDB to recive the operand as soon as it is available.

2.0.2 Intel Architecture

Meltdown researchers provide a simplified illustration of a single core of the Intel's Skylake microarchitecture: The pipeline of Intel's Skylake processors consists of the front-end, which fetches instructions from memory and decodes them into micro-operations (since intel's processors are CISC, while Superscalar/superpipelined processors suits better on RISC, the processor must decode complex operations into smaller, less complex micro-operations in order to leverage out-of-order execution), the back-end (execution engine), which implements out-of-order execution, and the memory subsystem. The Reorder Buffer is responsible of register allocation, register renaming and retiring (reordering instruction outputs as was intended by the program(mer)). Micro-operations are directly forwarded to the Unified Reservation Station that queues the operations on exit ports that are connected to Execution Units. Of course, Intel's Sky-

lake has it's branch predictor. Usally branch predictors are implemented with *taken/not taken* bits which tracks the history of a branch and indicates if previosly the branch was taken or not taken. This can be implemented with 1-bit or 2-bit counters. More on this on Branch Predictiors section.

2.0.3 How Meltdown leverages Reservation Station on Intel's micro-architecture

Since out-of-order execution allows the processor to execute instructions before previous instructions have effectivily terminated their tasks, it is impossibile for the processor to verify if any of the instructions that should be executed before raises an exception, e.g. access to a memory address where the program should not be able to, Meltdown leverages exactly this concept with transient instructions computing data that the program should not be able to access. More on this on the Meltdown chapter.

3 Side channel

Usually, CPUs support virtual address spaces to isolate processes from each other and to let compilers use logical addresses instead of directly accessing physical memory addresses. Virtual addresses are then translated to physical addresses. For optimization of memory usage, paging is also used to reduce memory usage and to separate User Space addresses from Kernel Mode addresses, in order to let only privileged processes to access kernel address space. Translation tables are used in order to define virtual to phisical mappings and also protection properties such as readable, writeble, executable and whether the page is accessible by user or not (meaning that only kernel mode processes can access the page). Every proces has its own translation table which is held on a special CPU register, so "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.

3.0.1 Exploitation and mitigation

Attacks that are targeting memory corruption bugs often requires the knowledge of addresses of specific data. ASLR mitigation has been introduced to randomize address space layout in order to obfuscate memory mapping to attackers. KASLR (Kernel Address Layout Randomization) was introduced to protect the kernel, randomizing the offsets where drivers are located on every boot, making attacks harder as they now require to guess the location of kernel data structures. But KASLR is not sufficent to mitigate Meltdown attacks since a simple bruteforcing of the memory physical addresses can leak such information.

3.0.2 What is side-channel

From Wikipedia, here's a definition of side-channel attack

> In computer security, a side-channel attack is any attack based on extra information that can be gathered because of the fundamental way a computer protocol or algorithm is implemented, rather than flaws in the design of the protocol or algorithm itself or minor, but potentially devastating, mistakes or oversights in the implementation.

Side channel attacks allows leaking of sensible information, like what pages a processes has recently accessed. These attacks allow detection of the exact location of kernel data structures or derandomize ASLR. Moreover, software bugs and the knowledge of these addresses can lead to privileged code execution.

3.0.3 How is side channel implemented

More in depth on side channels, there are many ways we can gather information, for example: timing, RF, electromagnetic emissions, and others. [reference: https://www.youtube.com/watch?v=D1DNz5sNDgE] In our case, simply monitoring the time a cache lines needs to reload leaks information about wether this information was in fact already loaded or not.

3.0.4 Covert channels

Covert Channel attacks are a special use case of side channels, where basically we intentionally send information to a system in order to induce the side effects we want to measure. [reference: meltdown, https://en.wikipedia.org/wiki/Covert_channel] Specifically for our use case, side channels includes: Evict+Time, Prime+Probe and Flush+Reload. We will discuss only the latter. These attacks are specifically designed to leak information from the cache exploiting timing differences induced by them selfs.

3.0.5 Flush+Reload

Flush+Reload is a variant of the Prime+Probe technique [reference: https://www.usenix.org/system/files/conference/usenixsecurity14/sec14-paper-yarom.pdf] where an attacker frequently flushes a targeted memory location using the clflush (cache line flush) instruction. By measuring the time it takes to Reload the data, the attacker determines wheteher data was loaded into the cache by another process in the meantime. An attack consists of three phases: first, the attacker flushes the memory cache line that he wants to monitor, then, he just waits for the victim process to read the same memory line; if the victim has, in fact, accessed that same memory line again, the value will be stored on the cache, otherwise no cache line will be loaded. Which brings the attacker to the third phase: if the cache line was loaded, the access time to that line will be very fast, otherwise the attempt will result on a "cache miss" which means that the victim process didn't access to the memory line again (in other words: the attacker will wait much longer to access that value). Usually, the unit measure used is "cycles the CPU needs to fetch the data" instead of microseconds or any othe time measure. Meltdown and Spectre attacks use this technique to know what is the value of the secrets the attacker wants to leak from a specified process or a specified physical memory range, in the Meltdown attack case.

4 Speculative Execution

Speculative execution is a technique implemented by the majority of modern CPUs to maximize perfomances. As the name suggests, it is based on the execution of operations that might or might not be performed. In case it's discovered that such instructions shouldn't have had been executed, all results are discarded, and CPU previous state is restored. For this reason speculatively executed instructions are also referred as transient instructions. In this section we will give a look at different speculation techniques, to better understand how the different versions of Spectre vulnerability work.
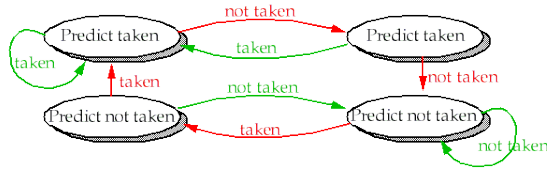
4.1 Branch Prediction

READ PIPELINING AND WRITE THIS INTRODUCTION FFS.

4.1.1 Static Branch Prediction

Static Branch Prediction is the simplest type of Branch Prediction. Predictor behaviour does not change during the execution of a program. The simplest examples are predictors that either predict that branch are always taken or always not taken. Some ISAs give the possibility, when using branch instructions, to insert a bit that hints wether a branch should be predicted taken or not.

### 4.1.2 Dynamic Branch Prediction

Dynamic Branch Predictors change their prediction based on information gathered at run-time, for an improved misprediction rate. A buffer, called Branch History Table(BHT) or Branch Prediction Buffer(BPB), is used to store predictions. The table maps a branch instruction address to bits used to store information about predictions' outcome. BHT implementations differ on how the mapping is done(Hash functions, k least significant bits, ...) and the number of bits associated with each address. The simplest way is using a single bit that stores the last outcome of the branch instruction(taken, not taken). This method doesn't take it count if the last prediction was or wasn't right, plus for every loop it's always wrong at least once. Using 2 bits can fix this problem, how the prediction changes can be summarized by the following state diagram.

Modern Branch Predictors use machine learning, state-of-the-art predictors use what's called perceptron predictor. This improves misprediction rate but increases latency. For the sake of this paper we will not dive into this argument.

4.2 Branch Target Prediction

Another type of speculation implemented in modern CPUs is Branch Target Prediction. Every time a jump instruction is encountered fetch cycles are lost to fetch and decode the instruction. To fasten up this process, in order to fetch the target instruction as soon as possible, modern CPUs implement what's called a Branch Target Predictor. Branch Target Predictor uses a buffer called Branche Target Buffer(BTB), which structure is analog to a cache: it associates instruction PCs to branch target PCs. Every time a new jump is fetched and decoded, its PC and target address are stored in the BTB. For every entry in the table 2 predictions bits are added, just like branch prediction 2-bit schema, to improve target prediction. This means that new entry have 2 prediction bits set as 'Predict Taken'. Every time an instruction is fetched, the BTB is looked up to check if it contains the instruction PC, if so, then the associated target address is sent out. If it target turns out to be correct then we've saved ... If not the entry is deleted from the BTB, and 2 cycles are lost. If the instruction PC is not in the BTB and after being decoded turns out it's a jump instruction then its PC and target address are saved in the table. This means that when the same jump instruction is encountered it is recognized as a jump instruction even before fetching it. Workflow can be seen in Figure underneath:
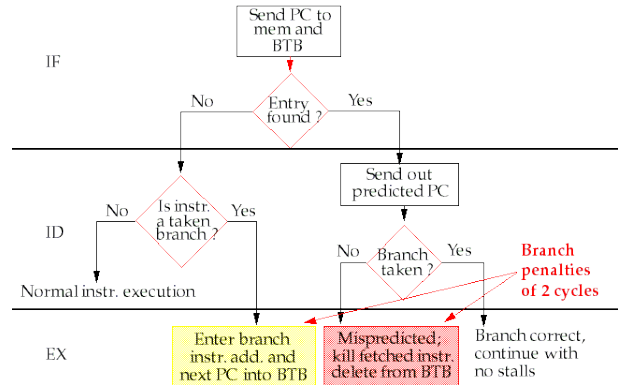
4.3 Return Address Prediction

Indirect jumps are jump instructions where the target address is not directly passed, a register or a memory address containing the target is given instead. This means that once the CU decodes the indirect jump instruction, clock cycles are spent to fetch the address from the register, cache or, worst-case scenario, a cache-miss happens and the target is fetched from main memory. The majority of this calls are from procedure returns. Even though a Branch Target Predictor could be used in this situation, its accuracy can be low in this situations. A buffer called Return Stack Buffer is used instead. It acts as a stack, so it pushes the latest return address on the stack and pops it off when a return is called.

4.4 Speculative Store Buffer Bypass

In order to improve performances, write operations(also called stores) are saved in a high speed buffer called Store Buffer. This allows the CPU to not wait for the buffer to be written back in slower memory. This implies that every time a read operaton on main memory is done, the CPU must check if a previously store operation on the same address was done and not written back. Modern CPUs bypass this check and assume that such stores are already written back, thus proceed to speculatively execute later instructions, and concurrently check the Store Buffer. If conflicts are found, results of transient instructions are thrown away, otherwise a significant speedup is achieved.

4

5 Meltdown

In this section we will explain how Meltdown attack works and how we managed to test it on our machines. This paper has the target of explaining how Meltdown works in a more "human readable" manner, and so we will provide pseudocode along with assembly code for reference.

5.1 Terminology

Meltdown resarchers have used their own terminology which helps the reader to better understand the main concepts.

Transient instructions are those instructions that are executed out-of-order, meaning those instructions that, in a program logic, should be executed after another instruction but, because of the micro-architecture of che processor, are executed at the same time.

Transient instruction sequence is a sequence of transient instructions.

5.2 How it works

Meltdown attacks consists of 3 main steps:

Step 1: Load content of (inaccessible) memory location on a register

Step 2: Allocate "Probe Array" on main memory

Step 3: Use the previosly loaded data to transimt secret on a legitimate instruction execution

Step 4: Store leaked secret on main memory leveraging Flush+Reload and previosly allocated Probe Array

5.2.1 Step 1: Fetch privileged data

Meltdown's main objective is to get privileged data from main memory which is otherwise inaccessible, and to do so the attack starts with simple access to an unauthorized memory location. In our example we will refer to such address as the "0xABC0" memory address, which our code is not autorized to access to, which points to the first byte of an array containing our secret ("Meltdown"). So our secret is stored from "0xABC0" ('m', first letter) to "0xABC7" ('n', last letter).

```
...
    secret = readAddress(0xABC0);
...
```
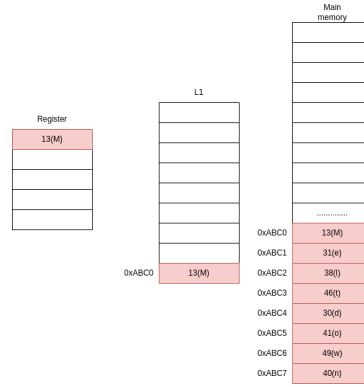

5.2.2 Step 2: Allocate Probe Array

In our example, we allocate a so called Probe Array which holds an array of "acceptable" values for our secret.

```
...
    secret = readAddress(0xABC0);
    probe_array=no_cache_array("A", "B", "C", ...
              , "Z", "a", ... "z");

...
```

For the sake of simplicity, no_cache_array is a function that allocates an array without caching. For example, accessing to probe_array[2] will result, at current micro-architectural state, in a "cache miss". This is a fundamental step for out-of-order exploitation. On the flush+reload step, what we want is that none of the pages holding these data is loaded but the one which store the value "M" since it is the first char of the secret value.

5.2.3 Step 3: Transmit secret

Now we have a register containing the value of the accessed secret and an array that contains all possible values that secret may be equal to, all is left to do is to legitimately get that value in order to store it on the main memory without the Reservation Station deleting its result after realizing that we should have not accessed that location genearting an architectural exception (also called "trap").

```
...
    secret = readAddress(0xABC0);
    probe_array=no_cache_array("A", "B", "C",
                , "Z", "a", ... "z");
    probe_array(secret);
...
```
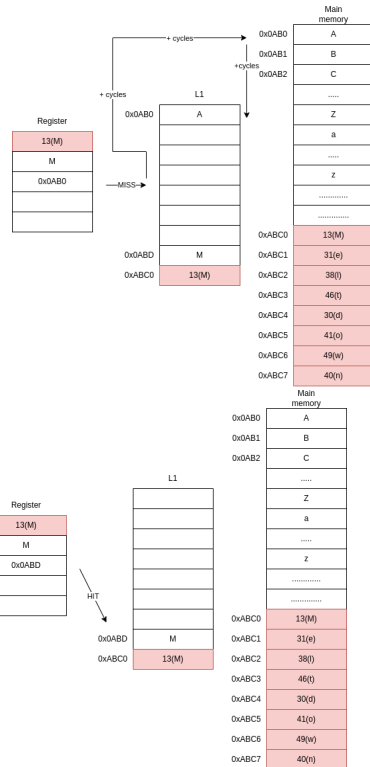

What we oversimplified on line 3 is what loads the desired page on our core cache. On a micro-architectural level, accessing the secret-th value of probe_array will first result on a "cache miss" and then the processor loads the value from main memory into the cache. Note that the pseudocode we provide dosen't really make sense on a more realistic lens, since we are assuming that the address "0xABC0" is storing the exact offset in which the value is stored on our probe_array. In a more realistic example we should first load the value on a register, e.g. RAX, and then translate that value in something that can be used to retrive a specific page from the memory which, like hasing functions, is equal to a well-known value. Also, an important note here is that each page of probe_array contains exactly and only a single value, being "A" for the first page, "B" for the second, and so on until "z".

5.2.4 Step 4: Flush+Reload to store the value

At this point all that's left to do is to store the secret value in a manner that Reservation Station will not delete its result.

```
...
    secret = readAddress(0xABC0);
    probe_array=no_cache_array("A", "B", "C", ...
                , "Z", "a", ... "z");
    probe_array(secret);
    for(i = 0; i < 52; i++){
        cycle_count_set(0);
        probe_array(i);
        if(cycle_count_get < 100) p = probe_array(i);
        clflush(probe_array(i))
    }
...
```

In our example, we iterate every page relevant to our probe_array in order to leverage Flush+Reload technique previosly discussed. We iterate all 52 pages of the probe_array and measure how cycles does it take to load the value: in our assumptions, if the cycle count is lesser than 100, then the page was already on cache, which means that line 3 was the last and the only who could have done that. we now procede to save the value on a register which will not be ereased by the Reservation Station since Line 7 is not doing anything wrong from the point of view of the Reservation Station. On Line 8 we flush the cache line so to leave pages of probe_array unloaded until we read the next privileged address.

5.3   More realistic example

6   Spectre

6.1   Spectre v1 - PHT

6.2   Spectre v2 - BTB

6.3   Spectre v3 - RSB

6.4   Spectre v4 - STL