Meltdown and Spectre

Began Bajrami, Rahmi El Mechri

June 27, 2022

Abstract

With this paper we want to make easier to understand what Meltdown and Spectre vulnerabilities are and how they work.

1 Introduction

Spectre and meltdown have deeply evolved since their discovery. The purpose of this paper is to allow people interfacing with these issues to have an overview on the current situation without being forced to rely on other sources. In order to do so, we have comprehended in-depth background sections.

2 Out of Order Execution

Out of order execution is the key feature that Meltdown vulnerability exploits on Intel's microarchitectures. Basically, what Meltdown does is possibile only because how Intel processors microarchitecture 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. 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

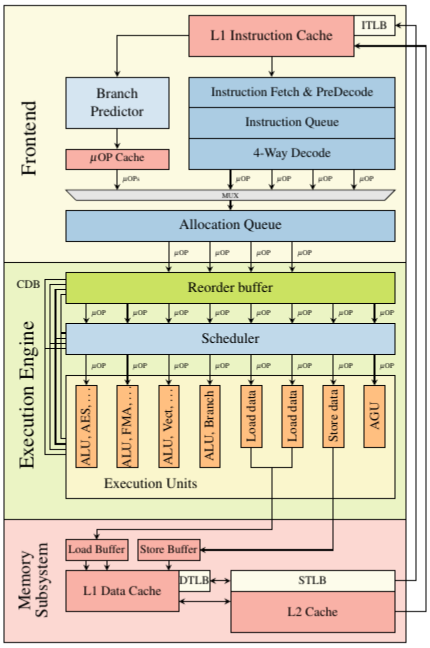

Figure 1: Simplified illustration of a single core of the Intel's Skylake microarchitecture. Credits goes to Meltdown research team

RISC, the processor must decode complex operations into smaller, less complex micro-operations in order to leverage out-of-order execution), the backend (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 Skylake 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 previously 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). These attributes are verified everytime an instruction is accessing them, resulting in a CPU trap (hardware exception) if the processes which required that address is not allowed to. This is handled by the Reorder Buffer on superscalar/superpipelined processors, when reordering results of instructions that have been executed out-of-order. Every process 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. 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. Specifically for our use case, side channels includes: Evict+Time, Prime+Probe and Flush+Reload. We will discuss only the latter since, as stated by Meltdown researchers, this is the faster and the more reliable way of doing it. 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 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 wether 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.

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

Branch Prediction is a technique that greatly enhances performances in superpipelined CPUs. When a conditional jump is encountered an entity called Branch Predictor decides if the content of that branch should be executed while fetching for data stored in memory needed to check for the condition. Different type of branch prediction have been implemented over the years. In this section we will describe Static Branch Prediction, prevalently used in old CPUs, and Dynamic Branch Prediction, present in every modern high-performance CPU.

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 Figure 2.

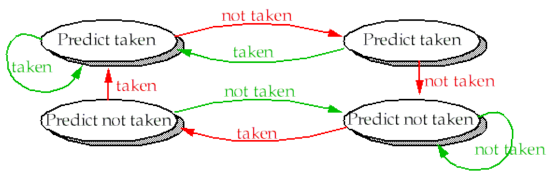

Figure 2: 2-bit branch prediction schema.

Dynamic Branch Predictor evolved and became more complex, and the concept of 2-level prediction arised. It is based on 2 concepts: Global Branch Correlation, or how a branch outcome is influenced by other branches, and Local Branch Correlation, that is how a branch is influenced by past predictions. This type of prediction uses what's called Patther History Table, or PHT, which associates the pattern(other branches + past outcomes) with the 2-bit schema seen before. This notably enhances the number of correct prediction. Modern Branch Predictors' PHT 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 its explaination.

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 Branch 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 achieved a boost in performances. 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 instruc-

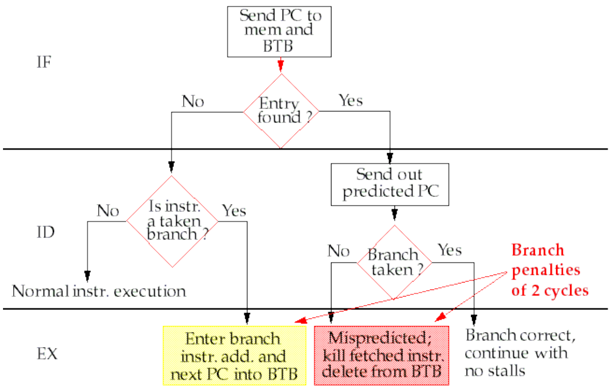

Figure 3: Branch Target Prediction workflow.

tion even before fetching it. Workflow can be seen in Figure 3.

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.

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.

5.1 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

5.1.1 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);
...
```

5.1.2 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.

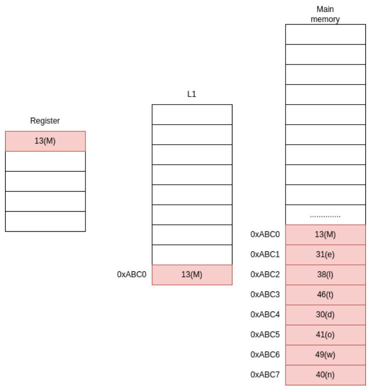

Figure 4: Current microarchitectural state

```
...
    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 and returns its address. 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.

5.1.3 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 Reorder Buffer 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 from a more realistic point of view, 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".

5.1.4 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 Reorder Buffer 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 previously 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 cached, 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 Reorder Buffer since Line 7 is not doing anything wrong from his point of view. On Line 8 we flush the cache line

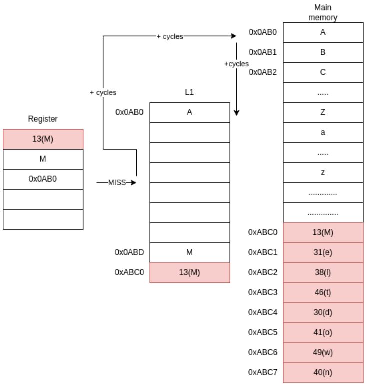

Figure 5: Microarchitectural state on cache miss

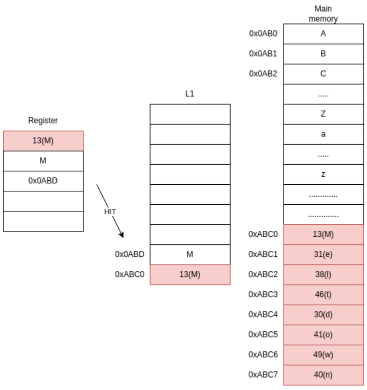

Figure 6: Microarchitectural state on cache hit

so to leave pages of probe_array unloaded until we read the next privileged address.

5.2 The importance of Transient Instructions

The Meltdown attack is possible only because these instructions are executed out-of-order.When subsequent instructions are executed earlier than intended are called transient instructions. As previously discussed, instructions accessing privileged addresses are not denied by the CPU even if the process is not allowed to: there's another mechanism handling the privileged access to these information, meaning instructions are executed regardless of their privilege but their result will raise an exception. Exceptions are in fact raised by Meltdown, but handled so that the program continues its execution regardless. A trivial approch is to fork the attacking application before accessing invalid memory location, so that when exception raises the child program crashes, but the parent can still observe the microarchitectural state, e.g. through a side-channel. Another way of handling exceptions is to suppress them thourgh Intel TSX which allows to group multiple instructions to a "transaction", which appears to be an atomic instruction. If one instruction within the transaction fails, already exectuted instructions are reverted, but no exception is raised. The microarchitectural effects are still visible.

6 Spectre

Spectre vulnerability is entirely based on the exploitation of speculative execution. When transient instructions are executed due to a wrong prediction, CPU is restored to its pre-prediction state, but many side effects remain unchanged, such as cache status, thus being one of the main side channels used for this attack, in particular the attacker might use Flush + Reload(x86 architecture supplies the clflush instruction for that purpose) or Evict+Time. As we've seen different speculation techniques are used nowadays, increasing the attack surface. This lead to the discovery of many variants, exploiting different techniques, using different side channels. For this reason we won't go into too much depth for evry one of them, and will instead give a brief description of every variant.

6.1 V1 - Conditional Branch Misprediction

The first variant exploits conditional branch mispredictions, allowing the attacker to aribtrarily read from another context. As seen in the branch prediction section, when a conditional branch is encountered and a taken branch is predicted, the branch instructions are executed while checking for the condition. Besides preparing the side channel the attacker must mistrain the branch predictor to make it execute transient instructions. This can be achieved in different ways, like inserting a certain number of passed condition The attacker might use different conditions, with the most one being a bound check, and then accessing an array out of its bounds. When using this condition the attack is known as Bound Check Bypass. The following is an example of this attack in C language:

```
if(x < array1_size)
        y = array2(array1[x] * 4096)
```

The ideal situation for the success of the exploit is such that array2 and array1_size are uncached, even though in some scenarios the exploit works even if array1_size is actually cached. In this example the secret(array1[x]) is byte-sized, and array2 dimension is 1MB. X is the offset from the starting address of array1 to the address we want to access. The content of array2 in position array1[x] * 4096 (with 4096 being the size of a page) is cached. To recover the secret value the attacker typically tries to access array2 in the 255 possible indexes and times every access. Accessing the cached content will take way less clock cycles, and at that point the secret is easily recovered by dividing the array2 index by 4096. The following is an example of this last described method. We are assuming to run this instructions on a machine where the cache access time is at worst 50 cycles.

```
int max_cache_access_time= 50;
int secret;
for(int i=0; i<256; i++){
        current_clock= __builtin_ia32_rdtsc ();
        y = array2( i * 4096);
        spent_clocks=
        __builtin_ia32_rdtsc () - current_clock;
        if(spent_clocks<=max_cache_access_time){
                secret = i;
                break;
        }
}
```

6.2 v2 - Branch Target Injection

This variant exploits the Branch Target Predictor, in particular its ability to predict indirect branches. The idea is mistraining the Branch Target Predictor in order to execute speculatively instrusction chosen by the attacker. What the attacker does is finding functions contained in the libraries used by the victim program. The concept is borrowed from Return Orienter Programming, a security exploit where arbitrary functions in a program are chained to together to execute what the attacker wants. We will call this functions gadget just like in the just explained security threat. To mistrain the Branch Target Predictor the attacker runs from its own context a program that reproduces the pattern of branches taken by the victim process before reaching the branch that must be mispredicted, thus exploiting the Branch Target Buffer. How it must be mistrained varies among architectures, as the number of bit used per destination address changes. After choosing the gadget, mistrainer we must branch to the same virtual address the predictor should mispredict to. It doesn't matter what it's branching to, the goal is correctly mistraining the predictor. This concept can be seen from Figure 7.

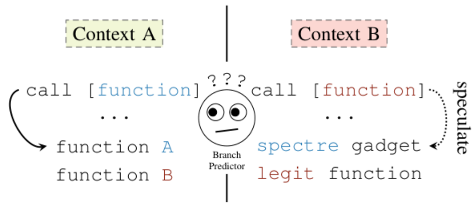

Figure 7: the attacker mistrains the predictor from his context(A), by jumping to a function that has the same virtual address as the gadget we want to be mispredicted in victim context(B)

We must also note that the mistrainer must run on the same core the victim program will run on, as prediction tables are not shared between different cores. This is true for every type of predictor mistraining explained in this paper. It has been proved that this attack allows to leak host memory from inside a guest Virtual Machine if the attacker has access to guest ring 0.

6.2.1 Branch History Injection

When Branch Target Injection first come out in 2018 Intel and ARM implemented respectively the eIBRS and CSV2 mitigations that prevent lower privilege programs from training the predictor into mispredicting branch target in higher privilege programs, by making the Branch History Buffer take into account the privilege the program is running in. We will better characterize these two mitigations later. However at the beginning of 2022 researchers of the VUSec group have found another way to mistrain the Branch Target Predictor allowing cross-privilege mistraining, and called this technique Branch History Injection. What they realized is that isolating Branch Target Buffer across different privileges is not enough, as the BTP relies on Branch History Buffer, that actually contains global entries. From userland attackers can inject entries into the BHB and fill it with gadgets' address. When kernel-level progams are executed the predictor will base its prediction on the manually inserted entries, thut achieving cross-privilege mistraining. Unlike BTI, AMD processors seem to be uneffected from this vulnerability, as only Intel and ARM CPUs are effected.

6.3 SpectreRSB

This variant of Spectre exploits the Return Stack Buffer, which job we have already explained in the Speculative Execution section. The way this is exploitation is done is by polluting the RSB(i.e. manually injecting entries into it), in order to have the victim program execute gadgets the attacker has chosen. Although not officialy adressed as an extension of Spectre v2, it uses an approach similar to the Branch HIstory Injection technique.

6.4 v3 - Rogue Data Cache Load

The third variant of Spectre is very rarely referred as Spectre v3, as it is more known as Meltdown. To deepen this topic, read Meltdown section.

6.5 v4 - Speculative Store Bypass

This last variant of Spectre, also known as SpectreSTL, exploits Speculative Store Buffer Bypass, allowing an attacker to read arbitrary priviliged data, or run older command speculatively than can lead to cache allocations, thus readable via common side-channel techniques.

7 Meltdown mitigations

Since Meltdown is a micro-architectural vulnerability, there is no software update that can completaly make mahcines secured from Meldown attacks. Not even KAISER (also known as PTI, page-table isolation, or KPTI on Linux kernel) which is the proposed mitigation by meltdown researchers make machines secure. That's because Mletdown, and Spectre, acts on hardware and bypasses the hardware-enforced isolation of security domains. A countermeasure would be to completely disable out-of-order execution but this will make processors slow enough to make any modern CPU parallelism mechanism completely useless and the performance impact would be devastating. As of 2022, PTI (KAISER) is enabled by default on Linux kernels as a countermeasure to Meltdown.

7.1 KAISER

KAISER (Kernel Address Isolation to have Side-channels Efficiently Removed) was not originally intended for Meltdown, but has as side effect the mitigation of it since KAISER prevents side channel attacks breaking KASLR. But this has its own limitations: first of all, performances will decrease since every context switch will need more clock cycles for address mappings; second, there is still a residual attack surface for Meltdown since several privileged memory locations are required to be mapped in user space. However, these memory locations do not contain any secrets, but they might contain pointers to Kernel Address space. This information, if leaked, is enough to break KASLR, as the randomization can be calculated from the pointer value.

8 Spectre Mitigations

In this section we will discuss the different countermeasures proposed for the Spectre vulnerability.

8.1 Speculative Execution Prevention

As stated multiple times, Spectre totally relies on Speculative Execution. Removing it would eliminate the vulnerability, but it would also mean renouncing to decades of performance improvement, therefore making this solution impracticable. AMD and Intel suggested, as software solution, using the lfence instruction, that doesn't allows instruction following it to be executed. In practice this means renouncing to speculative execution, hence should be used only when transient instructions could lead to dangerous otucomes.

8.2 Secret Data Access Prevention

Spectre vulnerability can be exploited using code written in JavaScript, thus browser, can and have to implement solutions to make the access to secred data more difficult. Google Chrome runs every website on a separate process as certain Spectre exploits are limited by victim permissions. Webkit implements 2 solutions: replaces array bounds checking with index masking, and xores pointers with pseudo random poison value. The first limits bounds violation, the latter prevents attackers that don't know the poison value to access those pointers.

8.3 Limiting Covert Channel

As all spectre variants use covert channels, limiting their exploitability is crucial. The researchers that discovered Spectre suggest that future processors could track if data was fetched by a transient instruction and prevent it to be leaked by limiting the number of operations that can be performed on it. Another way to limit covert channel is making known side channels harder, such ad the approachtaken by different browser that downgraded JavaScript timer, therefore making time-based side channels less efficient.

8.4 Branch Poisoning Prevention

Intel and AMD have implemented different mechanisms to prevent Branch Poisoning, released with a microcode patch extending the ISA: Indirect Brnach Restricted Speculation(IBRS), Single Thread Indirect Branch Predictio(STIBP) and Indirect Brnach Predictor Barrier(IBPB). IBRS prevents cross-privilege Branch Target Prediction mistraining, by isolating the BTB based on privilege. A similar mechanism is implemented by ARM's CSV2. STIBP limits the influence on branch predictor caused by thread running on the same core. IBPB prevents software running before a certain moment from influencing software executed after it. These mechanisms require OS support. Google suggests a mechanism, called retpolines, that replaces indirect branches with return instructions, and implementing a benign infinite loop to prevent speculation of indirect branches.

9 Conclusions

The discovery of Spectre and Meltdown vulnerabilities demonstrated that the neverending run for higher perfomances must be backed up by a thorough security check on what type of exploits this can lead to. Four years passed since their discovery, and researchers still find new ways to leverage speculative execution. It is clear that a Spectre-and-Meltdown-proof CPU is still far from being achieved, but a positive side effect is that the research for new microarchitectural bugs has been incentivized, and this resulted in the recent discovery of PACMAN vulnerability in new M1 chips.

10 References

- https://meltdownattack.com/meltdown.pdf
- https://spectreattack.com/spectre.pdf
- https://www.cs.cmu.edu/afs/cs/academic/class/15740-s17/www/lectures/L19-BranchPrediction.pdf
- https://web.archive.org/web/20190717130447/http://web.engr.oregonstate.edu/ benl/Projects/branch_pred
- https://ece-research.unm.edu/jimp/611/slides/chap4_5.html
- https://www.usenix.org/system/files/conference/woot18/woot18-paper-koruyeh.pdf
- https://www.cs.umd.edu/ meesh/411/CA-online/chapter/dynamic-branch-prediction/index.html
- https://docs.oracle.com/cd/E23824_01/html/819-3196/hwovr-14.html
- https://www.redhat.com/en/blog/speculative-store-bypass-explained-what-it-how-it-works
- https://www.techrepublic.com/article/spectre-and-meltdown-explained-a-comprehensive-guide-for-professionals/
- https://course.ece.cmu.edu/ ece740/f15/lib/exe/fetch.php?media=18-740-fall15-lecture05-branch-prediction-afterlecture.pdf
- https://download.vusec.net/papers/bhi-spectre-bhb_sec22.pdf
- https://www.usenix.org/sites/default/files/conference/protected-files/woot18_slides_koruyeh.pdf
- https://gruss.cc/files/kaiser.pdf
- https://www.youtube.com/watch?v=D1DNz5sNDgE
- https://en.wikipedia.org/wiki/Covert_channel
- https://www.usenix.org/system/files/conference/usenixsecurity14/sec14-paper-yarom.pdf