Meltdown and Spectre

Began Bajrami, Rahmi El Mechri

June 26, 2022

Abstract

TODO

1 Introduction

2 Out of Order Execution

Out of order is a technique used by many CPUs nowadays, 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 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) and use the last logical one to solve 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 which well illustrates the architecture. 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 make the most of 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. It is also responsible of 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 previosly the branch was taken or not taken. This can be implemented with 1-bit or 2-bit counters.

3 Address Spaces

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. Paging is also used 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.

3.0.2 Side channel attacks

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 allow to detect the exact location of kernel data structures or derandomize ASLR. A combination of software bug and the knowledge of these addresses can lead to privileged code execution. 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] Moreover, there's also the Covert Channel attacks, which 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.

3.0.3 Flush+Reload

An attacker frequently flushes a targeted memory locatiojn using the clflush 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.

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

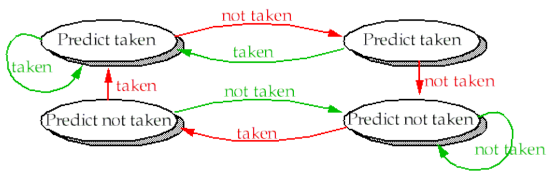
READ PIPELINING AND WRITE THIS INTRODUCTION FFS.

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

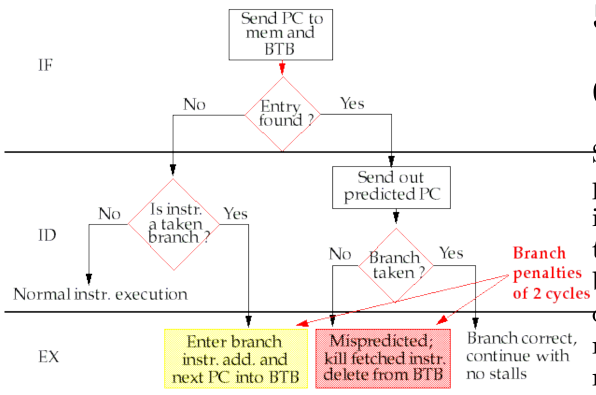
Dynamic Branch Predictors change their prediction based on information gathered at run-time, for an improved misprediction rate. A buffer, called Branch History Table(BHT) or Branch Prediction Buffer(BPB), is used to store predictions. The table maps a branch instruction address to bits used to store 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.



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 saved - TODO - 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:


IF
Send PC to mem and BTB
Entry found ?
No
Yes
ID
Send out predicted PC
Is instr. a taken branch ?
No
Yes
Normal instr. execution
Branch taken ?
No
Yes
Branch penalties of 2 cycles
EX
Enter branch instr. add. and next PC into BTB
Mispredicted; kill fetched instr. delete from BTB
Branch correct, continue with no stalls

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

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:

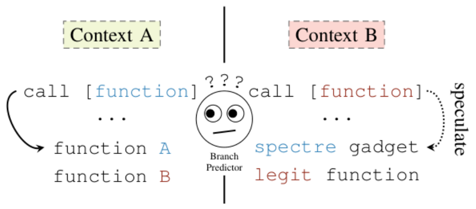
```
if(x < array1_size)
y = array2(array1[x] * 4096)
```

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 the following schema:



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 v3 - Rogue Data Cache Load

6.4 v4 - Speculative Store Bypass (STL)