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 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

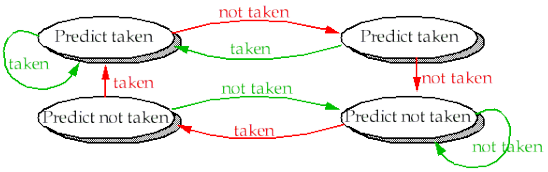
A signifcant number of implementations of branch prediction have been designed over the last 50 years, many of which are the evolution of previously designed

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

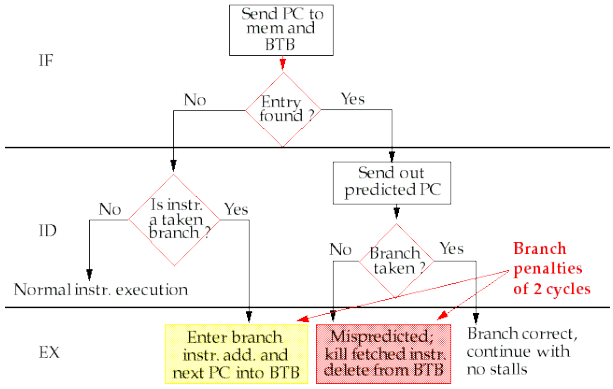
Dynamic Branch Predictors change their prediction based on information gathered at run-time, for an improved misprediction rate. A buffer, called Branch History Table(BHT) or Branch Prediction Buffer(BPB), is used to store predictions. The table maps a branch instruction address to bits used to store 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. FIX-AAAAAAAAAAAA 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 indirect jump is fetch 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 ... If not the entry is deleted from the BTB, and 2 cycles are lost. If the instruction PC is not in the BTB, if after being decoded turns it's an jump instruction then its PC and target address are saved in the table. Workflow can be seen in Figure underneath:



4.3 Return Stack Buffer

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.