University of Business and Technology (UBT)

Computer Science and Engineering

Advanced Operating Systems

SPECTRE attack and mitigations

Professor: Agon Memeti

Student: Kushtrim Pacaj

June, 2020

Abstract

One of the core aspects of operating systems is providing a level of separation between different processes running at a given time, meaning process A should not be able to access or modify the data of the process B. For example we wouldn't want some random process to be able to access the data of our banking application. SPECTRE is an attack that can manage to do just that.

SPECTRE belongs to the family of attacks exploiting a flaw on the feature of modern CPUs called *'speculative execution'*. This flaw can allow a rouge process on your computer manage to stealthy gain private data from either a different process, or from kernel itself. Ultimately the flaw is a hardware flaw and can be properly fixed only by upgrading your hardware. but there are some ways to mitigate it with software changes.

The goal of this paper is to showcase some forms of SPECTRE attack, explain how it works in details, and to also show steps that can be taken to protect from it.

Keywords: Spectre, Security, Speculative Execution, covert channel attack

Contents

I. Introduction

Moore's law states that "the number of transistors in an integrated circuit doubles every two years". This observation has held for decades, and our computers have been getting faster and faster. But all the speed improvements of late have not all been only due to more transistors, or the improved manufacturing process of CPU. Part of the reason for this speed-up have been multiple architectural decisions of CPU makers, that optimize the workload of the CPU in order to run as fast as possible. One of those optimizations is called **"speculative execution".**

"Speculative execution" is a feature of modern CPU that essentially in some cases executes instructions before it is asked to. It does this by guessing that it might have to in the future (e.g. speculates, hence the name), so it goes ahead and runs it. If it turns out that it guessed correctly, we just gained an huge speed improvement. If on the other hand turns out that it guessed incorrectly, the CPU rolls-back the changes it did, and continues normal execution. So essentially, it was thought that by doing this, in some cases we gain speed, in some cases not, but that there was no downside. It turns out that there is.

SPECTRE uses a flaw in the "*speculative execution"* in order to make the CPU leak data about the process that it was running. This is done by exploiting the CPU cache after a speculative execution, and running a covert-channel attack to extract the data from the cache, details which will be explained in further on this paper.

The objectives of this paper are to detail how SPECTRE works, and what has be done to fix the problem, and also to understand what are the limitations of the fixes, and speed penalties they incur.

1.1 Motivation

In my opinion, security and privacy are rights of every individual. These rights are very important in the context of computers and computing as well. When using any operating system, it is expected by users that the operating system (specifically kernel) will impose restrictions on what a process can do to affect other processes running concurrently with it. Normally, in most modern operating systems, unless said private process data is explicitly shared by the process via some IPC (Intra-process communication) tools such as Binder, or other, that data should not be accessible by others.

What is very important about SPECTRE attack is that it's a very wide ranging attack. It was possible to carry it out in Intel, AMD and ARM CPUs, and in all modern operating systems. The reason for this wide range of attack is that it's really a bug in CPU architecting, and not on a specific program or OS.

Due to SPECTRE allowing an attacker to in a sense breach the intra-process barriers and spy onto data from other processes, and it being such a wide-spread problem in the computing world, I thought it important to discuss how it is carried out and what can be done to mitigate it.

1.2 Problem Statement

Speculative execution attacks are a relatively new form of attack and more importantly a very problematic one to solve. SPECTRE is just one of these attacks (Meltdown being the other), and it has many variants. All of them force the CPU to speculate on a specific invocation of a program call, and use side-channel attacks to get the data from cache. But all variants have some specifics that we will go into later.

This paper will focus on and try to answer the following questions:

1. How does SPECTRE and its variants work?
2. How can we mitigate SPECTRE ?

3. How can we more automatically identify code paths where a SPECTRE attack could leak important information, and how do we fix them

1.3 Limitation/Scope

The original SPECTRE paper [1] identified only two variants at the time (*CVE-2017-5753* and *CVE-2017-5715*), but since then other researchers have identified dozens more. For simplicity's sake, this paper will solely focus in Spectre v1 and Spectre v2 when explaining how the attack is carried out.

In order to mitigate it, we have 3 options:

1. Upgrade to newer hardware, newer CPUs that will solve the issue in hardware level.
2. Turn off speculative execution altogether (a nuclear option, which would cause a major perceived slowdown of computer systems)
3. Identify and modify important parts of code containing very sensitive data (e.g. PII, keys) , and turn off CPU speculation on those parts only.

Option 1 and 2 are out of scope of this paper, and I will focus mostly on option 3, which is the more interesting one. Although a combination of 1 and 3 will also be discussed.

1.4 Methodology

I used a systematic literature review approach to gathering information about this paper. Since the main stated goal is to explain SPECTRE and mitigations, my paper search criteria contained the following keywords *"spectre"*, *"spectre mitigations"*, *"spectre versions"*, *"spectre detections"*. Another inclusion criteria was that the papers had to be 2018 or newer, and since this attack is quite new, there is a large number of papers that match this criteria.

The sites used to search for papers were *Google Scholar, JSTOR and Research Gate*. Since the search yielded more papers than I could possibly use, I had to add some filtering and priority mechanism to my search. More priority was given to papers appearing on renowned peer-review publications. After that I also turned to papers appearing in pre-print servers such as arXiv, making sure to prioritize papers that had more citations.

Afterwards, all papers were read to make sure that they are relevant and can answer some of the questions posed in the problem statement.

In the end, I filtered the list of papers to only 10 most relevant ones, which are used and listed as references in the end of this paper.

1.5 Contribution

The most important contribution to this topic has been done by the authors of the original SPECTRE attack[1], which were 3 groups that independently of each other found and exploited the problem. They then followed with suggestions for mitigations, one of which was ConTeXT [2]. The main authors of the paper were Jann Horn of *Google Project Zero* and Paul Kocher. A group of PhD students from Gratz university contributed some possible ways to easily find and mitigate Spectre.

This paper is mainly meant as a consolidation and synthesis of information regarding the problem statement, while also trying to add something new by comparing current approaches and adding recommendations.

II. Literature review (State of the Art)

(Provide short literature review & related work about your Case study)

As mentioned before, one of the goals of the paper is to show in detail how SPECTRE can be exploited to attack and leak data from another process. I will aim here to show a short summary of the state of SPECTRE attacks , and in the next section of Critical Review I will review these in more detail.

The paper from Kocher et. al.[1] is the first paper which first alerted the research community and the world on SPECTRE. They detailed how the attacked can mis-train the branch predictor of the CPU in order to get it to speculate wrongly, how it can get a ‘protected memory value’ from the target process into the CPU cache, and how the attacker can then get that value from cache. They explain the second variant too, which differs from the 1st on not using if-conditionals as targets, but indirect branching. Chládek[4] shows a Rust implementation of the attack.

Other SPECTRE variants came later, such as v3a (Rogue System Register Read) , v4 (Speculative Store Bypass)., SpectreRSB/ret2spec which are explained by Hill et. al. [7], and Koruyeh et.al. [9]. Another one, NetSpectre which is done over the network shown in Schwarz et.al[6]

Schwarz et. al. [2] unveil a novel method of protecting against SPECTRE, by adding a new annotation which developers can use in code to mark areas of the program which should really be private, such as some encryption key, password, bank card number etc. Then their patches to the compiler mark those memory areas as un-cacheable memory. So their goal is to only protect important information, while still allowing for possible leaks of other data by a SPECTRE attack. They draw here a middle line between security and between speed.

While the approach of Schwartz was considered a valid one by researchers, it meant that programmers of many applications will have to re-compile their binaries and use their mitigation approach. Khaled et.al.[8]'s SafeSpec achieves a mitigation by creating shadow structures on speculation so that the data is not leaked directly. Their approach has the benefit on not having to re-compile every program out-there, but unfortunately requires some small hardware modifications too.

Most of the other papers I reviewed like the ones mentioned above try to explain how they either found a new variant of SPECTRE, or how they have a possible solution on easier mitigation. There was one that stood out though, Depoix et. al. [3] approaches the problem in a novel way, it doesn't try to mitigate SPECTRE, it tries to detect SPECTRE in action, using machine learning. The idea behind the paper is that when a SPECTRE attack is being performed by a rouge process, that process will be acting in weird ways, accessing and clearing or evicting the cache in a more or less predictable manner. The goal therefore is to identify and shut-down an attack while it’s happening. A very unique take by Depoix.

Wang[5] tries to do something similar, but via analysis and not machine learning.

III. Critical Review || Use Cases

In the previous section we listed the current state of research regarding SPECTRE and mitigations. In this section I will go into more detail on these, and in addition to explaining the use cases, I will also add my own comments in regards to them.

3.1 Spectre v1 attack explanation

As mentioned before, SPECTRE exploits *‘speculative execution’*. Let’s consider the following code:

```
if (x < arrayA_size)
   y = arrayB[arrayA[x] * 4096];
```

Normally, from a programmer’s point of view, we assume that the first line is executed first always, and then after the boolean expression is calculated, the second line is run. And if we were running this code more than a decade ago, we would be right. But this isn’t what always happens in moderns CPUs. When the CPU runs the first statement with the boolean expression, it might need to go to main RAM memory to fetch a value. Fetching a value from RAM takes a lot more time than executing something inside CPU. So the CPU now has two options:

a) Wait for the fetched value from RAM, then based on it execute the second line (or not).

b) Speculate about whether the first line will evaluate to true or not, and run the if or else branch.

Since doing b) we will gain some speed if the speculation was correct, CPUs choose to do speculate. Whether they speculate that we will go to the IF clause or ELSE clause, is decided by the CPU’s branch predictor, which does it’s job by analyzing the earlier invocations.

For SPECTRE v1, our goal as an attacker is to force the CPU to execute the second line even when it shouldn’t. This would mean that arrayA is accessed with an X value out-of-bounds, so we access a memory area that is outside of this array. This memory area is our target, the data that we want to ‘steal’.

The way we fool the CPU into speculating wrongly is to mis-train the branch predictor by calling this piece of code with a valid X many times, so that when we call it with an X out of bounds, the CPU will speculate that it’s within bounds of arrayA.

After CPU speculates and executes the second line, now our goal is to *GET* this data. Normally, when the CPU gets the result from the IF check and finds out that it speculated wrong, it will rollback the changes it did to it’s registers when running the 2nd line, and go back to a clean state. Or so it was assumed…..

The huge BUG that Kocher et. al.[1] managed to find and exploit is that the data calculated is not cleared from the CPU cache. Yes, registers are rolled back, but the CPU cache is still dirty. It is really stressed out in the paper that this bug is a hardware bug on the CPUs of Intel, AMD and ARM, and not a bug on the high level operating system (HLOS).

The first phase of SPECTRE was to mis-train branch predictor, the second phase to invoke the function with the value we want and having the CPU speculate badly, while the third phase is to get the data from cache. This is done by using a technique called *cache-based side channel attack*.

Harris [10] explains three possible modes of side channel attack that could be used:
Prime+Probe, Flush+Reload and Evict+Time. Each approach a little, but the general idea behind all of them is this: timing value access, and surmising whether we have a cache hit or miss, depending on the time it takes. Here’s a more watered down example by me:
Suppose that we have in CPU cache the secret value “A” located originally in memory address XXX. Attacker then proceeds to create a probe array, which is not cached. It then tries to access a value in the array, by targeting a memory address in it, value of which memory address is calculated by using value of address XXX (value which is in cache but we do not know yet). This is the sender part of the side channel attack. When the value is accessed, part of the probe array is pulled into cache.

Now by this time, CPU might have figured out it speculated wrong and revert the changes. But the receiver end of the side-channel now can try to access the probe array little by little. In the part of the where it was cached before, it will take significantly shorter time to access. Based on this, we can summarize that this is the value accessed before, and from this information we can summarize what the value in cache used to be.

3.2 Spectre v2 vs v1

This is a variant of v1, and most of the attack is the same as with v1. Let's look at that piece of code again:

```
if (x < arrayA_size)
   y = arrayB[arrayA[x] * 4096];
```

In version 1, we could control the value of X. When we cannot control that value always, then we can use version 2 of SPECTRE. This relies on another exploit of branch predictor, and indirectly triggers the branch speculation we want, irrelevant of the value of X.
All other parts of the attack are the same, and do not merit a repeat.

3.3 Spectre mitigations

As we can see based on the SPECTRE attack vector, this attack has a wide surface area. Any exposed point of entry into our program, and any branching after could theoretically be used to mount a SPECTRE attack. More importantly, this attack was not limited to only stealing data from other processes, the same exploit could be used to spy on protected kernel memory as well.

Due to the sheer number of devices that were vulnerable (most processors from Intel, AMD, ARM) before the exploit was made public, it was first responsibly disclosed to the CPU vendors, who in turn alerted the OS vendors. This meant that by the time the exploit was made public, there were already patches for main operating systems to mitigate the problem.
Like I said before, SPECTRE is a hardware bug, but we can mitigate it's effect via software changes.

One of the first mitigations developed was the usage of the *lfence* CPU instruction in order to block the CPU from speculation. The implementation might look like this:

```
if (x < arrayA_size)
   /// a call to lfence here, means next instruction won't be speculatively executed
   y = arrayB[arrayA[x] * 4096];
```

This will work, but it is a very tedious work identifying all the possible points of attack. Because of this hardship, new ways were developed.

ConTeXT[2] can be used to mark areas of memory that should be secret as un-cacheable memory, which would protect against SPECTRE. SafeSpec[8] goes one level lower, by making changes to the CPU and compilers, so that the attack while it could still be carried out, it would never leak relevant data.

IV.Conclusion and Future work

SPECTRE attack was one of the worst CPU flaw exploits in the last couple of years. By using it, and running on an un-patched system, a rouge process could read data from other processes and even the kernel itself.

Nowadays, while the operating systems and important programs have mostly been patched, there are still thousands of softwares running on older CPUs that are still vulnerable. I would recommend that everyone makes sure that at least important software that they use and that has data that should be kept secret (such as your banking app) is already patched.

As specified on introduction, the first objective of this paper was to show how SPECTRE works, and we have done that in detail for both variants 1 and 2. I have also explained some of the possible mitigations, such as using *lfence*, using *ConTeXT*, or *SafeSpec* (which requires new CPUs) which was the second objective.

Lastly I should mention that all mitigations come with speed penalties, because we're turning off a CPU optimization. These speed penalties vary depending on the implementation.

I would consider as future work for SPECTRE attack finding new mitigations which fix the problem, but have lower speed penalties (due to only turning off speculation where strictly necessary)

V. References

1. Kocher, P., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M., & Yarom, Y. *(2019).* ***Spectre Attacks: Exploiting Speculative Execution.*** *2019* IEEE Symposium on Security and Privacy (SP), 1-19.

2. Schwarz, Michael & Lipp, Moritz & Canella, Claudio & Schilling, Robert & Kargl, Florian & Gruss, Daniel. (2020). ***ConTExT: A Generic Approach for Mitigating Spectre.*** 10.14722/ndss.2020.24271.

3. Jonas Depoix and Philipp Altmeyer. 2018**.** ***Detecting Spectre Attacks by identifying Cache Side-Channel Attacks using Machine Learning***. Advanced Microkernel Operating Systems (2018), 75.

4. Chládek, Jaroslav & Kokeš, Ing. (2019). ***Feasibility of the Spectre attack in a security-focused language.*** 10.13140/RG.2.2.36524.21120/4.

5. Wang, Guanhua & Chattopadhyay, Sudipta & Gotovchits, Ivan & Mitra, Tulika & Roychoudhury, Abhik. (2019). oo7: ***Low-overhead Defense against Spectre attacks via Program Analysis.*** *IEEE Transactions on Software Engineering.* PP. 1-1. 10.1109/TSE.2019.2953709.

6. Schwarz, Michael & Schwarzl, Martin & Lipp, Moritz & Masters, Jon & Gruss, Daniel. (2019). ***NetSpectre: Read Arbitrary Memory over Network***. 10.1007/978-3-030-29959-0_14.

7. Hill, Mark & Masters, Jon & Ranganathan, Parthasarathy & Turner, Paul & Hennessy, John. (2019)***. On the Spectre and Meltdown Processor Security Vulnerabilities***. IEEE Micro. PP. 1-1. 10.1109/MM.2019.2897677.

8. Khasawneh, Khaled & Mohammadian Koruyeh, Esmaeil & Song, Chengyu & Evtyushkin, Dmitry & Ponomarev, Dmitry & Abu-Ghazaleh, Nael. (2018). ***SafeSpec: Banishing the Spectre of a Meltdown with Leakage-Free Speculation.***

9. Mohammadian Koruyeh, Esmaeil & Khasawneh, Khaled & Song, Chengyu & Abu-Ghazaleh, Nael. (2018). ***Spectre Returns! Speculation Attacks using the Return Stack Buffer.***

10. Harris, Rae, ***"Spectre: Attack and Defense"*** (2019). Scripps Senior Theses. 1384. https://scholarship.claremont.edu/scripps_theses/1384