Protecting against cache-based attacks

MATTHEW BAUER, University of Kansas, USA

Recently, significant side-channel vulnerabilities have been discovered utilizing the CPU cache. These vulnerabilities are incredibly widespread in modern processors and a significant concern to the security community. In this report, three attacks involving the CPU cache are analyzed along with software-based prevention methods. Unfortunately, these preventions methods come with a cost. They end up mitigating some usages of the CPU cache that provide critical performance benefits. This report examines whether the benefits of preventing each vulnerability is worth its cost.

Additional Key Words and Phrases: Side-channel attack, cache-based vulnerability, Spectre, Meltdown

1 INTRODUCTION

In late 2017, two major vulnerabilities were discovered affecting many different types of CPUs. They were named "Meltdown" and "Spectre", making international news in early January[1][2]. The significance of these vulnerabilities stems from the fact that they effect an entire CPU and can be exploited by any machine code that is executed. Meltdown and Spectre are just two examples of a wide class of vulnerabilities called cache-based side-channel attacks. Broadly, side-channel attacks leak information that is meant to be private through "indirect mechanisms". Cryptographic primitives still hold, but underlying information is leaked[4].

In this report, the details behind cache-based side-channel attacks will be analyzed. The specific "Meltdown" and "Spectre" attacks will be examined alongside the "FLUSH+RELOAD" method used in both attacks. After each vulnerability is laid out, prevention mechanisms are analyzed. Because prevention often handicaps the machine's cache, emphasis is placed on how much each prevention mechanism will cost. Finally, the report concludes with a cost benefit analysis of preventing these attacks.

2 VULNERABILITY ANALYSIS

2.1 CPU Cache

All vulnerabilities studied in this work will look at the CPU cache. So it is important to first understand what the CPU cache is and how it works in modern processors. The behavior can be

In modern processors, caches are used to avoid repeated access to memory. Implementation details vary signficantly between processors, but they all aim to make the computer perform faster. The basic idea is that when the same memory is frequently accessed by the CPU, the long trip to the memory can be avoided. This means that repeated accesses will not have to touch the memory at all. As a result, when an address of memory is available in the cache, the access operation is significantly faster[3].

The time these operations take is an implementation detail to programmers, but a critical detail to attackers. Timing attacks are a common way for attackers to leak information. Early work by Kocher showed that it is possible to use this information to find secrets used in cryptographic primitives[6]. Chip manufacturers such as Intel and AMD take some precautions to avoid these attacks. But, the discovery of new vulnerabilities has proven that they are not enough[1].

2.2 Flush+Reload

Flush+Reload is a method for reading data that is loaded into the CPU cache. It will be used as a final step in both the Meltdown and SPECTRE attacks. Flush+Reload requires memory to be loaded

Author's address: Matthew Bauer, University of Kansas, 1450 Jayhawk Blvd, Lawrence, KS, 66045, USA.

into the CPU cache to work. Loading memory into the CPU cache will be accomplished by both SPECTRE and Meltdown.

As the name implies, the attack flushes memory and then waits for the memory to be reloaded. There are three phases in the Flush+Reload method. They are,

- Phase 1. Clear the address of memory from the cache. When the victim tries to access this line, it will have to read it directly from memory because it is now unavailable in the cache.
- Phase 2. Attacker waits for the victim to access the address of memory. The attacker will know that the victim has access the address by how long it takes to reload the address. The Flush+Reload paper uses 120 cycles as the differentiater between the two. If it takes less than 120 cycles to load, the memory is in the cache. Otherwise, the memory must not be in the cache.
- Phase 3. The attacker retrieves the timing information from the probes. The attacker can then make deductions based on how many cycles certain operations take. The entire process can be repeated many times to get more information, both in quality and quantity.

Timing used in Flush+Reload is highly machine dependent. Different CPU architectures, implementations, and even operating systems will effect how long each load takes. Things like speculative execution and other processor quirks can make these timings unreliable. The result is that this attack is highly temperamental.

However, the Flush+Reload technique makes it very clear that it is possible, given the right environment, an attacker can get the contents of memory that has been stored in a CPU cache. Flush+Reload provides a baseline in getting side channel information out of the CPU cache. It is proof that once memory leaks into the CPU cache, it is possible for an attacker to find its value. [9]

2.3 Meltdown

Meltdown takes advantage of out-of-order execution in modern processors. To increase performance, processors will often execute instructions out of order. A common scenario is for one instruction to take a long time while nearby instructions are very quick. Sophisticated heuristics in processors are in place to execute the quick instructions in the background as the longer instructions "stall". Programmers benefit from these optimizations without having to understand what the processor is doing. The processor produces identical results to an in-order execution. Attackers, however, are able to take advantage of this behavior to break memory isolation.

From Meltdown's paper, the "core" of the Meltdown vulnerability is included,

```
; rcx = kernel address, rbx = probe array
1. xor rax, rax
2. retry:
3. mov al, byte [rcx]
4. shl rax, 0xc
5. jz retry
6. mov rbx, qword [rbx + rax]
```

Meltdown consists of 3 steps:

- Step 1. A memory location known to be inaccessible is loaded into a register. This is seen in #3 from the listing above. This will raise an exception that will halt execution. Later, we will leak the contents of this memory location.
- Step 2. A later instruction uses the loaded register for an operation. This is seen in #4 from the listing above. If the processor executes this instruction before #3, it will keep the access

to rcx within the CPU cache. After the exception is called, the secret memory will remain there.
- Step 3. Use FLUSH+RELOAD to get the contents of the secret memory that have leaked into the CPU cache.

The specifics get more complicated, from here. The authors of Meltdown wrote the attack so that it could maximize the chance of the memory appearing in the CPU cache. The attack relies on speculative execution to work so it is important that the processor runs the cache access instruction before the forbidden access instruction. This is not guaranteed so many retries may be required to get the processor in the correct state.

Meltdown enables attackers to break the memory isolation that is the cornerstone of computer security. An arbitrary memory access can get all kinds of secrets that can be used to break into other parts of the computer. Every CPU with speculative execution could be vulnerable against this attack. Prevention would require a CPU being much more careful on what it loads into the cache. [7]

2.4 Spectre

Spectre takes advantage of speculative execution found in certain Interl, ARM, and AMD processors. Unlike Meltdown, only a subset of modern processors are vulnerable to this attack. At the same time, the Spectre attack provides another mechanism that an attacker can use to exploit private memory.

Speculative execution is used by processors to optimize performance. The processor predicts ahead of time what branch is most likely to be used. It will then execute that branch before the conditional. The processor guarantees that any side effects produced will be reverted if the conditional doesn't run as predicted. However, the values that have been cached are not reverted in vulnerable processors. Spectre uses this vulnerability to exploit CPUs.

As a first step, the Spectre attack must trick the CPU into speculatively executing some branch. There are two variants of this. One utilizes conditional branching. The other utilizes indirect branches. The attacker trains the CPU by repeatedly running the branches to make the CPU speculatively execute the branch.

Spectre lays out many different variants that tweak the attack to work in different ways. They leverage the side effects of speculatively executed code. One variant provides an attack to code that doesn't have any conditional branches at all. Another variant utilizes how the process uses store-to-load in execution. Together these show that there are multiple tweaks possible on the original Spectre attack that can give attackers plent of opportunities. [5]

3 PREVENTION AT THE SOFTWARE LEVEL

The vulnerabilities above are part of the way that the CPU executes code. They are essentially hardware vulnerabilities that require changes at the hardware level to fix properly. After the attacks were discovered, hardware manufacturers were able to rework their designs to avoid the underlying vulnerability of both Spectre and Meltdown. However, there are still tons of computers out there still vulnerable to Spectre and Meltdown. Patches are needed to existing software to work around the limitations found in the hardware. [1]

3.1 Flush+Reload

Because both Meltdown and Spectre rely on Flush+Reload, if we can prevent Flush+Reload, we can also prevent Meltdown and Spectre. So, what methods are available to prevent Flush+Reload.

Unfortunately, the core of the Flush+Reload attack is part of the x86 architecture. “clflush” is the instruction used by Flush+Reload to perform the “flush”. There are legitimate uses of clflush in adding and removing parts to the cache. For performance, programs can remove memory from the cache that it knows it will no longer need.

At the hardware level, CPUs could forbid the “clflush” instruction from flushing memory that it does not own. This change would make the Flush+Reload impossible. But, this won't provide a fix for Flush+Reload to the many vulnerable processors out in the wild.

The main way to prevent this attack at a software level is by changing how accurate timing probes are. One method to accomplish this is by adding noise to clock times. This may work for this immediate attack, however, other methods are possible to get the clock such as getting the time over the network. Other methods may be available, but only one method can protect against every variation of this attack.

The best solution in preventing Flush+Reload is to use constant time operations so that the attacker cannot differentiate set and unset bits. This would require reworking how we use cryptographic primitives. As of today, some constant time cryptographic methods are being developed but are still under development and not widely available. In addition, the performance costs of this constant time cryptography is very high and may never be feasible.[9]

3.2 Meltdown

Patches were released to protect against Meltdown before it was released to the public. [1] The Linux kernel already had a patch that protected against it called KAISER. This change to the Linux kernel provides two page tables for each processes. One is reserved for kernel mode and contains all contents of the page table in a normal kernel. The other page table contains only the process's address space. The kernel's address space is hidden completely from the process and every system call is mapped outside of the process table. Together, KAISER eliminates the Meltdown attack as the kernel space is no longer accessible. However, KAISER significantly hurts performance because every system call has to go through its table mapping first. This results in about a 5% decrease in performance. More optimizations are possible but some decrease in performance is inevitable. [**?**]

Other countermeasures are possible. For instance, we could just disable out of order execution completely. However, this would result in unacceptable performance for normal use. Another potential countermeasure involves preventing the race condition between memory fetch and permission check. This solves the core issues but also comes with a large cost. Every memory fetch will have to wait for the permission check to be completed before it can finish. This was deemed unacceptable for normal usage.

The solution accepted by most operating systems has been to adapt the KAISER patch found in the Linux kernel to their own use cases. Apple and Microsoft have modified their kernels to separate address spaces for the process and the kernel. Again, this results in about a 5% decrease in performance. Other mitigations may be possible, but research is still ongoing. [7]

3.3 Spectre

The Spectre attack could be prevented by completely avoiding speculative execution. Because this is the core of the attack, disabling it makes the attack impossible. Again, however, this would cost significant performance as speculative execution is critical source of improved performance. In addition, no processors provide a way to disable speculative execution entirely at the software level, so this solution is currently not feasable.

Another approach to this mitigation would be to produce code that blocks at speculative execution. Using certain instructions, the processor can be told to block speculation at certain branches.

Compilers can produce code that uses these instructions everywhere so as to avoid the Spectre vulnerability. Again, this would come with a not insignificant cost.

In browsers, Spectre can be partially prevented using different processes for each tab. This means that an attacker would only be able to see what is in their own tab - not others that are open. Process isolation separates the page tables from each other, but like all other mitigations, comes with a cost. However, that cost is not huge and this method is already used in the Google Chrome browser. In addition, this only mitigates Spectre partially while still allowing complete access to the page table. [5]

Because there are many variants of Spectre, no one solution will be available. Each variant will require different fixes. [8] Altogether Spectre provides a significant attack surface for any processor that utilizes speculative execution. [5]

4 COST BENEFIT ANALYSIS

It has been shown above that cache-based side-channel attacks can break process isolation. An attack can read arbitrary parts of the host computer's memory. However, for the attack to happen, the attacker must already be able to execute code on the victim's machine. Computers are only vulnerable if they run untrusted code. So the question follows, how often is untrusted code run on different computers?

Computers are used for many different applications. Certain scenarios requrie a processor to execute lots of untrusted code. These include things like cloud computing infrastructure and other things utilizing virtual machines. These computers are almost certainly going to require both the KAISER patches as well as better handling of speculative execution. On the other hand, many personal computers do not utilize as many of these features. Most code run on a personal computer is trusted and there is little need to worry about leaking. On the other hand, many web browsers end up executing lots of untrusted code in the form of JavaScript. Historically, JavaScript has been a huge attack vector and it appears to be the case in Spectre and Meltdown as well. It seems inevitable that more intensive fixes will be needed for web browsers and cloud container infrastructure that need not go into general Linux kernel usage. [8]

In addition, the software preventions to these attacks are quite costly. They change critical performance optimisations in ways that can prevent these attacks. How much do these preventions effect different computers?

In their review of the impact of Spectre and Meltdown, Prout et. al find a 15% slow down in kernel performance after all available patches for Spectre and Meltdown. This study used machine intensive computations to measure performance, but it is likely this would be felt even by average computer users. Altogether this is a fairly large cost that could hurt normal computer usage. In addition, older hardware that is already slower than average will likely be slowed to an unacceptable level. [8]

Finally, we must compare the cost of the prevention to the potential of being attacked. Does the cost of preventing Spectre and Meltdown at the software level outweight the benefits of a more secure system?

This will depend highly on the use case of the computer system. For embedded systems, it is very likely the cost to avoid Spectre and Meltdown is too high for the cost. In comparison, cloud computing infrastructure will almost certainly require every patch available as keeping virtual machines isolated is a crucial part of their services. Desktop and server systems may wind up somewhere in the middle on this continuum. They will need some of the lowest cost patches for Spectre and Meltdown while keeping some of the performance benefits of out of order execution and speculative execution. [8]

5 CONCLUSION

In this work, the basic mechanisms of both Spectre and Meltdown have been described. The two share many common parts but use different behaviors of the CPU to work. As a result, each needs to be protected against separately. The software prevention and mitigations were describes and compared. The results give a good idea of how these mitigations work. Finally, the cost and benefits of preventing Spectre and Meltdown at a hardware level were examined. Questions were asked and we tried to provide as best answers as are currently available.

REFERENCES

[1] Peter Bright. 2018. "Meltdown" and "Spectre:" Every modern processor has unfixable security flaws. *Ars Technica* (Jan 2018). https://arstechnica.com/gadgets/2018/01/meltdown-and-spectre-every-modern-processor-has-unfixable-security-flaws/

[2] Brad Chacos and Michael Simon. 2018. Meltdown and Spectre FAQ: How the critical CPU flaws affect PCs and Macs. *PCWorld* (Feb 2018). https://www.pcworld.com/article/3245606/security/intel-x86-cpu-kernel-bug-faq-how-it-affects-pc-mac.html

[3] Ulrich Drepper. 2007. Memory part 2: CPU caches. *LWN.net* (Oct 2007). https://lwn.net/Articles/252125/

[4] Wayt W. Gibb. 2009. How Hackers Can Steal Secrets from Reflections. *Scientific American* (May 2009).

[5] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2018. Spectre Attacks: Exploiting Speculative Execution. arXiv:arXiv:1801.01203

[6] Paul C. Kocher. 1996. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology (CRYPTO '96)*. Springer-Verlag, London, UK, UK, 104–113. http://dl.acm.org/citation.cfm?id=646761.706156

[7] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown. arXiv:arXiv:1801.01207

[8] Andrew Prout, William Arcand, David Bestor, Bill Bergeron, Chansup Byun, Vijay Gadepally, Michael Houle, Matthew Hubbell, Michael Jones, Anna Klein, Peter Michaleas, Lauren Milechin, Julie Mullen, Antonio Rosa, Siddharth Samsi, Charles Yee, Albert Reuther, and Jeremy Kepner. 2018. Measuring the Impact of Spectre and Meltdown. arXiv:arXiv:1807.08703

[9] Yuval Yarom and Katrina Falkner. 2013. FLUSH+RELOAD: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. *International Association for Cryptologic Research* (2013). https://eprint.iacr.org/2013/448.pdf