A Systematic Study on Spectre Attacks and Defenses

Haehyun Cho*
Soongsil University, Seoul, South Korea
haehyun@ssu.ac.kr

Abstract

Spectre attacks is an important category of side channel methods, which allows attacker to obtain sensitive data by observing the system. Spectre attacks exploit modern processors' features designed for the performance: out-of-order execution and speculative execution. Also, in Spectre attacks, cache side-channel attack methods play an important role. The high-level goal of Spectre attacks is to load target data into the cache through the speculative execution. Once it has been done, the next step is leaking information. To the end, the cache side-channel methods are employed to leak information, because there is no direct way to read data from the cache. In this paper, we discuss variations of Spectre attacks and discuss defense mechanisms for each of them.

Keywords: spectre attacks, cache and side-channel, stack-based information leaks

1 Introduction

Spectre attacks is an important category of side channel methods, which allows attacker to obtain sensitive data by observing the system. Spectre variants abuse the optimization features of modern processors such as the out-of-order execution [1]. Especially, Spectre attacks are well-known for their ability that can break isolation between applications (and the operating system potentially) [1, 2]. In this paper, we revisit the Spectre attack variants from the root cause, and discuss proposed defense mechanisms for each of them to find limitations still remained. To this end, we discuss variations of Spectre attacks and introduce each generalized defense method and discuss their effectiveness as well as the efficiency.

1.1 Cache and Side-channel

Except the register which is the fastest and built in the processor, cache is on the top of the memory hierarchy [3]. Cache, which is a high-speed and small internal memory to buffer the data that is frequently used, is built for processors to overcome the latency of system memory access. The design of cache had been made based on two basic principles: (1) Cache is a shared resource by multiple processes; and (2) Cache supports processors in such a way that they can have very fast access time for any data in cache. Naturally, there is a noticeable difference in access time between a cache hit and a cache miss. If we check the time spent in loading data, we can figure out whether or not the data has been accessed by other processes. This simple fact allows attackers to leak sensitive data from other processes [4, 5, 6, 7, 8, 9].

Attacks using the difference of data access times is so called the cache side-channel attack. These attacks have been developed to have formalized and generalized attack methods for different attack situations [10, 11, 12, 13, 14, 15, 16]. Cache side-channel attacks have been exploited to steal cryptographic keys, user inputs, execution paths, and addressing information [17, 18, 19, 20, 21, 22, 23, 24].

In Spectre attacks, cache side-channel attack methods play an important role. The high-level goal of Spectre attacks is to load target data into the cache through the speculative execution. Once it has been

*Corresponding author: Tel: +82-(0)2-820-0945, Web: https://haehyun.github.io done, the next step is leaking information. To the end, since there is no direct way to read data from the cache, the cache side-channel methods are employed to leak information regarding the data finally to *guess* what the data is or what the address of the data is.

1.2 Optimized Execution Features

Modern processors have highly optimized techniques to maximize the performance. An *out-of-order* execution is an approach for handling multiple instructions in parallel manner by executing some instructions in advance instead of executing instructions in strict order of their sequence. Different instructions have different execution times, depending on operation types. To take an example, a store operation's execution time is much longer than a load operation. Therefore, if a processor can executes load instructions while it is processing a store operation, the processor can save many CPU cycles.

As proven, processors which are utilizing the our-of-order execution typically have another optimization feature: *speculative* execution. The speculative execution is also to handle instructions in advance for the performance reason. However, a processor execute instructions that it does not know whether to execute or not, which totally relies on a prediction made by a processor.

One representative example is when a processor meets a conditional branch instruction. In case where operation results of preceding instructions affect the branch condition, a processor executes instruction along with a path that is predicted by the branch predictor rather than waiting the results. If the prediction is correct, a processor can gain a lot of instruction cycles, maintaining a fully utilized pipeline. Otherwise, it has to rewind its execution state to the point before the execution of branch instruction happened and take the other path.

The result of the misprediction does not change program's logical state but gives a performance penalty on the program. However, the impact of the misprediction is actually not limited to the performance. More importantly, it introduced new micro-architectural attacks—Spectre. Even though program's logical state can be restored correctly, results of the speculative execution cannot reinstate some changes of hardware resources such as the cache. Once data has been fetched into the cache by a speculatively executed load instruction, the data will remain in the cache albeit the prediction was wrong. Consequently, the changes of the cache arisen from the speculative execution renders attacker to be able to obtain sensitive information via the cache side-channel.

2 Spectre Attacks

In this section, we discuss on four variations of Spectre attacks.

2.1 Variations of Spectre Attacks

According to the type of the speculative execution, four major Spectre variants have been discovered. Commonly, overall procedure of spectre attacks consists of three steps as follows: (1) Preparing an attack to execute a target code speculatively and to leak information *e.g.*, making a branch condition is being predicted to be true; (2) Executing the code executes speculatively; (3) Leaking information from the cache side-channel. We briefly overview how each variant utilizes different type of the speculative execution with its related example code.

Variant 1 (Bounds Check Bypass). This variant targets a conditional branch instructions. A processor can execute a branch based on a prediction result of the branch predictor without knowing whether it actually will be taken [25]. An example is illustrated in Code 1. When the variable `length` is uncached, a processor needs to wait until the data is arrived from the memory and the comparison operation is done, not executing Line 2. However, as far as the prediction result is *true*, the Line 2 can execute speculatively—the data will be cached.

```
if(offset < length) {
  *fp_target = arr[offset];
}
```

Code 1: Example of Spectre variant 1.

Variant 2 (Branch Target Injection). Branch target injection takes advantage of the indirect branch predictor. The core idea is to flush the cache line containing the address to which program need to jump. If it happens when a processor needs to jump and the cpu is waiting the address being fetched from main memory, the processor will not know or jump to the address [26]. Consequently, during the time the actual data is being fetched, we can make a processor speculatively execute instructions in maliciously crafted memory area where the indirect branch predictor pointed to.

```
offset = read_offset();
(*fptr_arr[offset])();
```

Code 2: Example of Spectre variant 2.

Variant 3 (Rogue Data Cache Load). Spectre Variant 3, as known as Metldown [27, 2], showed accessing kernel memory area from user applications was possible by exploiting the speculative execution. This variant is to make a processor read inaccessible memory area speculatively while the exception handler is taking care of an exception intentionally raised. Any kind of exceptions is available for causing this speculative execution.

```
fake = *kernel; // for an exception
data = arr[offset_to_kernel_area];
```

Code 3: Example of Spectre variant 3.

Variant 4 (Speculative Store Bypass). The most recently discovered Spectre variant 4 exploits an optimization feature that renders load instructions execute speculatively [28]. It typically happens during a store instruction is executing by aid of the memory disambiguation predictor. The point is it can happen even if the address from which a processor need to load has not decided by the preceding operations.

```
*offset = old_addr;
...
*offset = new_addr; // being bypassed
data = arr[*offset]; // load
```

Code 4: Example of Spectre variant 4.

3 Defenses against Spectre Attacks

Proposed countermeasures against Spectre attacks can be categorized into three-fold: (1) Disabling optimization features; (2) Using hardware instructions to suppress speculative executions; and (3) Approaches using software only such as sanitizing array indices. We introduce each generalized defense method and discuss their effectiveness as well as the efficiency. Unfortunately, any practical solution that meets both the effectiveness and the efficiency has not shown up yet to mitigate all kinds of Spectre attacks in Kernel.

3.1 Disabling Speculative Execution

First off, since the processors' optimization features are the root cause, Spectre attacks can be mitigated by disabling them. For currently deployed processors, disabling the speculative execution can be accomplished by updating a microcode. Typically, microcodes are used to interpret a processor's instructions when an instruction executes for providing additional functionalities of them or security purposes.

Manufacturers such as Intel has updated many CPUs' microcode to prevent Spectre attacks. Especially, updated microcode targets for mitigating Spectre variant 2, 3, and 4. For an example, Speculative Store Bypass Disable (SSBD) microcode has been introduced by Intel, AMD, and ARM for inhibiting a speculative store bypass (Sepectre variant). This microcode makes store instructions cannot be bypassed so that any instruction after a store instruction is not able to execute speculatively.

However, employing the microcode for the entire kernel cannot be a suitable solution. As the more microcode is being used for incapacitating the speculative execution, naturally the heavier performance degradation will be imposed. Also, the performance impact on kernels is directly related with user-level applications.

3.2 Using Memory Barrier

A memory barrier is a type of hardware instruction that constraints memory operations to enforce them execute in sequential order, prohibiting out-of-order executions. A processor manufacturer—Intel—suggested using the Load Fence (LFENCE) instruction to serialize programs' execution order. Especially, the LFENCE instruction is known for that it does not allow any following data loading operations until the execution of it has finished, in turn, consequently, the instruction seems it can be used to defend Spectre attacks.

The first concern with respect to the barrier instruction is the performance degradation similar to the use of microcode. Because, if the LFENCE is used broadly, strictly serialized executions caused by the barriers can make a devastating performance overhead. Therefore, the use of LFENCE instructions must be accompanied with a sound static analysis method to find appropriate and minimal places for inserting the instruction.

However, static analyses are neither complete nor sound. As far as a sound static analysis is not possible, applying the LFENCE instructions on kernels can be too liberal. When it comes to the simple patterns of code gadgets that can be exploited by Spectre attacks, we can easily expect that numerous false-positive results will come out. Consequently, huge manual efforts are inevitable to find code gadgets and insert the barrier instruction.

More importantly, the architecture manual's unclear explanations on the LFENCE instruction disorientate us to employ it. White papers of Intel guided that the LFENCE instruction between a store and the subsequent load can prevent the speculative execution, but, the architecture manual states that processors are *free* to fetch data to the cache from the memory speculatively even though the LFENCE instruction is used. Therefore, the effect of load fence instruction has to be refined clearly to do not misinform security researchers and developers and ultimately to prevent security accidents caused by it. In this work, we empirically show the effect of barrier instructions and a found zero-day-vulnerability that comes from misusing it.

3.3 Other Software Mitigations

3.3.1 Kernel Page Table Isolation (KPTI)

This mitigation method is designed to eliminate reachable kernel data from the user-land through the Spectre variant 3 (Meltdown). The kernel exposes the minimal set of page tables that can be used to call system functions from user-land applications. The other set of page tables for accessing the kernel code and data is active only when the kernel code is executing. Isolating the kernel page tables is an expensive solution in terms of the performance overhead but it effectively helps kernels to keep their sensitive data from the Spectre variant 3. However, since the kernel at least needs to open system functions for running applications on it, KPTI is not able to prevent Spectre attacks using code gadgets of the kernel.

3.3.2 Retpoline

It is a compiler-based method that can protect indirect branches from the Spectre variant 2. Retpoline utilizes a trampoline technique (loops constructed by indirect branches) until the speculative execution stops and jump to the actual address. In other words, to avoid the indirect branch predictor's prediction results, Retpoline let a processor execute code inserted by it. Even though this mitigation showed the promising result with a good performance compared with a mitigation using the microcode, but, it has been proved that some of the recent processors such as Intel's Skylake can ignore the Retpoline effect[1].

4 Concluding Remarks

In this work, we discuss on variations of Spectre attacks and defense mechanisms against them. Because the root causes of Spectre attacks were design issues designed to improve the performance of the modern processors, unfortunately, any practical solution that satisfies both the effectiveness and the efficiency has not shown up yet to mitigate all kinds of Spectre attacks in Kernel. Therefore, it still is of great importance to develop an effective and efficient security solution to mitigate Spectre attacks.

Acknowledgment

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. NRF-2021R1A4A1029650).

References

[1] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre Attacks: Exploiting Speculative Execution. *ArXiv e-prints*, Jan. 2018.

[2] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. Meltdown. *ArXiv e-prints*, Jan. 2018.

[1] https://lwn.net/Articles/744287, https://blogs.oracle.com/linux/an-update-on-retpoline-enabled-kernels-for-oracle-linux-v2

[3] D. Gullasch, E. Bangerter, and S. Krenn. Cache games–bringing access-based cache attacks on AES to practice. In *Proc. of the 32nd IEEE Symposium on Security and Privacy (S&P'11), Oakland, CA, USA*, pages 490–505, IEEE, May 2011.

[4] Y. Zhang, A. Juels, A. Oprea, and M. K. Reiter. HomeAlone: Co-residency Detection in the Cloud via Side-Channel Analysis. In *Proc. of the 32nd IEEE Symposium on Security and Privacy (S&P'11), Oakland, CA, USA*, pages 313–328, IEEE, May 2011.

[5] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee. Last-level cache side-channel attacks are practical. In *Proc. of the 36th IEEE Symposium on Security and Privacy (S&P'15), San Jose, CA, USA*, pages 605–622, IEEE, May 2015.

[6] G. Irazoqui, T. Eisenbarth, and B. Sunar. S $ A: A shared cache attack that works across cores and defies VM sandboxing-and its application to AES. In *Proc. of the 36th IEEE Symposium on Security and Privacy (S&P'15), San Jose, CA, USA*, pages 591–604, IEEE, May 2015.

[7] R. Guanciale, H. Nemati, C. Baumann, and M. Dam. Cache storage channels: Alias-driven attacks and verified countermeasures. In *Proc. of the 37th IEEE Symposium on Security and Privacy (S&P'16), San Jose, CA, USA*, pages 38–55, IEEE, May 2016.

[8] E. Tromer, D. A. Osvik, and A. Shamir. Efficient cache attacks on AES, and countermeasures. *Journal of Cryptology*, 23(1):37–71, 2010.

[9] M. M. Godfrey and M. Zulkernine. Preventing cache-based side-channel attacks in a cloud environment. *IEEE transactions on cloud computing*, 2(4):395–408, 2014.

[10] Y. Yarom and K. Falkner. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *Proc. of the 23rd USENIX Security Symposium (Security'14), San Diego, CA, USA*, pages 719–732, August 2014.

[11] D. Gruss, R. Spreitzer, and S. Mangard. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In *Proc. of the 24th USENIX Security Symposium (Security'15), Washington, DC, USA*, pages 897–912, August 2015.

[12] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard. ARMageddon: Cache attacks on mobile devices. In *Proc. of the 25th USENIX Security Symposium (Security'16), Austin, TX, USA*, pages 549–564, August 2016.

[13] M. Green, L. Rodrigues-Lima, A. Zankl, G. Irazoqui, J. Heyszl, and T. Eisenbarth. AutoLock: Why Cache Attacks on ARM Are Harder Than You Think. In *Proc. of the 26th USENIX Security Symposium (Security'17), Vancouver, BC, Canada*, pages 1075–1091, August 2017.

[14] D. Gruss, F. Schuster, O. Ohrimenko, I. Haller, J. Lettner, and M. Costa. Strong and Efficient Cache Side-Channel Protection using Hardware Transactional Memory. In *Proc. of the 26th USENIX Security Symposium (Security'17), Vancouver, BC, Canada*, pages 217–233, August 2017.

[15] D. Gruss, C. Maurice, K. Wagner, and S. Mangard. Flush+Flush: a fast and stealthy cache attack. In *Proc. of the 13th Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA'16), San Sebastián, Spain*, pages 279–299, July 2016.

[16] M. Schwarz, S. Weiser, D. Gruss, C. Maurice, and S. Mangard. Malware guard extension: Using SGX to conceal cache attacks. In *Proc. of the 14th Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA'17), Bonn, Germany*, pages 279–299, July 2017.

[17] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. Cross-VM side channels and their use to extract private keys. In *Proc. of the 19th ACM Conference on Computer and Communications Security (CCS'12), Raleigh, NC, USA*, pages 305–316, ACM, October 2012.

[18] Y. Zhang and M. K. Reiter. Düppel: Retrofitting commodity operating systems to mitigate cache side channels in the cloud. In *Proc. of the 20th ACM Conference on Computer and Communications Security (CCS'13), Berlin, Germany*, pages 827–838, ACM, October 2013.

[19] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. Cross-Tenant Side-Channel Attacks in PaaS Clouds. In *Proc. of the 21st ACM Conference on Computer and Communications Security (CCS'14), Scottsdale, AZ, USA*, pages 990–1003, ACM, November 2014.

[20] Z. Zhou, M. K. Reiter, and Y. Zhang. A software approach to defeating side channels in last-level caches. In *Proc. of the 23rd ACM Conference on Computer and Communications Security (CCS'16), Vienna, Austria*,

pages 871–882, ACM, October 2016.

[21] M. Hähnel, W. Cui, and M. Peinado. High-Resolution Side Channels for Untrusted Operating Systems. In *Proc. of the 2017 USENIX Annual Technical Conference (ATC'17), Santa Clara, CA, USA*, pages 299–312, July 2017.

[22] G. Irazoqui, M. S. Inci, T. Eisenbarth, and B. Sunar. Wait a minute! A fast, Cross-VM attack on AES. In *Proc. of the 17th International Symposium on Research in Attacks, Intrusions and Defenses (RAID'14), Gothenburg, Sweden*, pages 299–319, September 2014.

[23] O. Aciiçmez. Yet another microarchitectural attack:: exploiting I-cache. In *Proc. of the the 2007 ACM workshop on Computer Security Architecture (CSAW'07), Alexandria, VA, USA*, pages 11–18, ACM, October–November 2007.

[24] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A.-R. Sadeghi. Software grand exposure: Sgx cache attacks are practical. *arXiv preprint arXiv:1702.07521*, 2017.

[25] *CVE-2017-5753*. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-5753.

[26] *CVE-2017-5755*. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-5755.

[27] *CVE-2017-5754*. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-5754.

[28] *CVE-2018-3639*. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-3639.

Author Biography

Haehyun Cho received the Ph.D. degree from the School of Computing, Informatics and Decision Systems Engineering of Arizona State University, majoring in computer science, and especially concentrating on information assurance. He is currently an Assistant Professor with the School of Software and the co-Director of the Cyber Security Research Center at Soongsil University. His primary research interests lie in the field of systems security to discover and mitigate security concerns. He is, also, passionate about analyzing, finding, and resolving security issues in a wide range of topics.