Claremont Colleges
Scholarship @ Claremont

Scripps Senior Theses

Scripps Student Scholarship

2019

Spectre: Attack and Defense

Rae Harris

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

This Open Access Senior Thesis is brought to you for free and open access by the Scripps Student Scholarship at Scholarship @ Claremont. It has been accepted for inclusion in Scripps Senior Theses by an authorized administrator of Scholarship @ Claremont. For more information, please contact scholarship@cuc.claremont.edu.

SPECTRE: ATTACKS AND DEFENSES

by

RAE YAN YAN HARRIS

SUBMITTED TO SCRIPPS COLLEGE IN PARTIAL FULFILLMENT OF THE DEGREE OF BACHELOR OF ARTS

PROFESSOR ELEANOR BIRRELL

PROFESSOR CHRISTOPHER TOWSE

APRIL 25, 2019

Copyright © 2018 Yan Yan Harris

The author grants Pomona College the nonexclusive right to make this work available for noncommercial, educational purposes, provided that this copyright statement appears on the reproduced materials and notice is given that the copying is by permission of the author. To disseminate otherwise or to republish requires written permission from the author.

Abstract

Modern processors use architecture like caches, branch predictors, and speculative execution in order to maximize computation throughput. For instance, recently accessed memory can be stored in a cache so that subsequent accesses take less time. Unfortunately microarchitecture-based side channel attacks can utilize this cache property to enable unauthorized memory accesses. The Spectre attack is a recent example of this attack.

The Spectre attack is particularly dangerous because the vulnerabilities that it exploits are found in microprocessors used in billions of current systems. It involves the attacker inducing a victim's process to speculatively execute code with a malicious input and store the recently accessed memory into the cache.

This paper describes the previous microarchitecture side channel attacks. It then describes the three variants of the Spectre attack. It describes and evaluates proposed defenses against Spectre.

Acknowledgments

I would like to thank Professor Birrell for her support as I explored a part of computer science that I had little background in. I would also like to thank my family for supporting me as I moved between majors and for listening as I worked through my thesis.

Contents

Abstract . . . i
Acknowledgments . . . iii

1 Introduction 1

2 Background 3

3 Cache-Based Side Channel Attacks 5
3.1 Prime+Probe . . . 5
3.2 Flush+Reload . . . 6
3.3 Evict+Time . . . 8

4 Spectre Attack 9
4.1 Spectre Variant 1 . . . 9
4.2 Spectre Variant 2 . . . 11
4.3 Spectre Variant 3: Meltdown . . . 12

5 Spectre Attack Defenses 15
5.1 Emergency Patches . . . 15
5.2 Microsoft BIOS Update . . . 16
5.3 Independent Benchmarking . . . 18

6 Conclusion 23

Appendix A 23

Bibliography 29

Chapter 1

Introduction

A system is *secure* if it does what is supposed to do and nothing else. For example, when a secure program is run, there should be no hidden side effects in the output or the state of the machine it ran on. Current hardware and software development focus primarily on optimizing performance. Standard optimizations include various microarchitecture components like caches and branch predictors. For example, when a process accesses memory during execution, it will store it within the cache. If the executed code was in a conditional branch, then the record of the branch is stored in the branch predictor. This memory storing can have unintended but measurable side effects, depending on how processes execute. These side effects, introduced by the optimizations, introduce vulnerabilities that can be exploited by attackers.

One way for information to be accessed is through what is known as a side channel attack. A *side channel attack* is an attack that exploits the implementation of a computer system to access unprivileged information. When performing side channel attacks, it is often assumed that the attacker has physical access to the user hardware. However, in some cases [Kocher et al. [2018]] attacks can be successfully executed remotely.

In January, 2018, a series of side channel attacks that exploit speculative execution were documented and reported. These attacks were named Spectre Variant 1 [Horn [January 3, 2018]], Spectre Variant 2 [Kocher et al. [2018]], and Meltdown (or Spectre Variant 3) [Lipp et al. [2018]]. In this paper, Variant 1 and Variant 2 will be collectively referred to as the Spectre attack unless specified otherwise.

This paper will proved a general background of cache based side channel attacks and how the Spectre attacks work. Then it will detail how, in response to three Spectre Variants, companies like Intel and Microsoft developed various defense patches. The patches, while efficient, were unrefined in the beginning and caused a decrease in system performance. Later patches were more robust but are also limited in their application to later generations of CPU [Miller [March 15 2018], Hruska [October 22 2018], int [January 9 2018]].

Chapter 2

Background

Computer processor designers aim to optimize the computational performance and output while minimizing the power drain, the time for an operation to fully execute, and the memory footprint. The developers achieve these goals by the creation of various microprocessor components, such as data and instruction caches, branch prediction units [Acıiçmez and Koç [2009]], and branch prediction buffers [Acıiçmez et al. [2007]]. Developers also utilize strategies such as parallel execution, speculative execution, and multithreading. These tactics allow the machine to minimize wasted or idle clock cycles. For instance, most modern Intel CPUs save memory space by sharing memory pages between cores. The Intel CPU processes in each processor also share the last level cache (LLC) (i.e. the L4 if there are four cache levels or L2 if there are two).

Caches help reduce CPU latency because their stored data can be accessed by the processor at a faster rate than the same data from main memory. The cache is split into multiple levels where the deeper levels have more memory space, but require more clock cycles to be accessed. Each level is comprised by cache sets which are in turn made up of cache lines. Addresses are mapped to these sets, and the index that determines which set the address is mapped to is dependent upon the physical and virtual addresses. The last cache level is typically shared across all cores. Therefore, any changes made to that cache level can affect processes running on other cores [Gruss et al. [2016]]. The number of cache levels available is determined by the CPU manufacturer [Falkner and Yarom [2014]].

The amount of time a memory access takes to complete depends on whether it is a *cache hit* or a *cache miss*. A cache hit occurs when a process is able to find the memory it needs using what is stored in a cache, and thus data access time is faster. If the memory it wants is not stored in the cache, a cache miss has occurred and the process needs to fetch the data from main memory and load it into the cache for future use. Since accessing main memory is slower than accessing a cache, the execution time is slower when a cache miss occurs.

Out of order execution is where instructions are executed out of the order that they are written; once all instructions have been fully executed, they are committed in the program's execution order. Out of order execution introduces a vulnerability due to any side effects caused by the execution of code that follows code that causes a program to halt or crash.

Branch prediction units are used to make an educated guess of which instruction should be executed next. They are used by speculation units to determine which conditional branch will be taken after the condition is computed. This guess is made by using the Branch Target Buffer (BTB) which keeps a record of recently executed branch instructions and their destination addresses [Kocher et al. [2018]]. From this record, the processor analyzes the BTB's structure history in order to predict the future code addresses before the branch instructions are decoded.

Execution time can be reduced by using idle clock cycles, which occur when a condition for a conditional loop is being evaluated and the data required must be retrieved from main memory instead of a cache, to execute other code. A process will use *speculative execution* to make use of these idle cycles. Speculative execution uses the branch predictor to speculate which branch is most likely to be taken, records the state of the registers, and preemptively executes the code within as the condition is being evaluated, storing any data accessed within the cache. If the guess was correct, then more work was finished, the idle cycles utilized, and the changes are committed. Otherwise the processor will revert back to the recorded state, and there was minimal performance loss since the speculation used previously idle cycles. In both cases, the changes to the cache state are not reversed, creating a data vulnerability.

An attacker accesses protected data by launching a *side channel attack*, which is an attack that allows unprivileged processes to attack other parallel processes [Gruss et al. [2016]]. According to researchers, the earliest side channel attacks were reported in 1965 [Zhou and Feng [2005]]. Traditional side channel attacks include cryptoanalysis, algorithm, and cache timing attacks [Acıiçmez and Koç [2009]].

Timing attacks are where the attacker is measuring the time that a user process takes to execute a specific task. Any significant deviation in the time can be used by the attacker to obtain the protected information. These are one of the easier ways to remotely attack because timing attacks do not require any extra hardware or physical access to the machine [Brumley and Boneh [2005]].

Attackers who monitor the time of an instruction execution are able to measure the increased time from a cache miss, a *time spike*, which helps enable the attackers to find out which data was not previously cached. An example of the time spikes can be seen in Figure 2.1, where prefetched data stored in the cache had a lower latency than the data not prefetched.

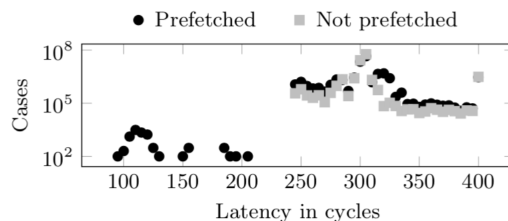

Figure 2.1: A visual example from [Gruss et al. [2016]] that shows the access latency when prefetched or not through the kernel memory.

Chapter 3

Cache-Based Side Channel Attacks

When performing a microarchitectural attack, there are two main steps that the attacker must take. First, the attacker must configure the cache so that they know what the cache's state is before the victim interacts with it. Second, the attacker will collect timing data and use it to gain information about the victim's operation. Many of the attacks will either be a direct attack to obtain a secret key, or they will try to obtain information about the virtual address and physical address that map to processor-reserved memory in order to allow other types of attacks to be used. For instance, defenses like address space layout randomization (ASLR), supervisor mode execution prevention (SMEP), and supervisor mode access prevention (SMAP) have been implemented to prevent virtual address and physical address based attacks [Gruss et al. [2016]].

3.1 Prime+Probe

There are two steps an attacker takes when doing a Prime+Probe attack. First the attacker primes the cache by occupying a specific cache set. This attack does not require the attacker to have a set knowledge of the cache's state. Once the attacker is occupying the cache set, continuous calls are made that access memory which maps to the occupied set. The attacker also keeps track of the time it took for the victim's operation to finish executing.

In an experiment, the authors discuss how Prime+Probe measurements are made to attack Advanced Encryption Standard (AES) cryptosystems. In the example, the attacker tries to discover which set of memory blocks were read during the encryption process. This is done by first allocating an array named A and then *priming* the cache by reading a value from every block within array A, thus filling up all the cache sets with the attacker's data. The attacker then forces an encryption of a plain text P. If the encryption accesses memory that is mapped to one of the cache sets that the attacker has filled, then the data in the set is evicted and replaced with the encryption-accessed data. Once the encryption has occurred, the attacker may then go and *probe* the cache sets to discover which ones were evicted, and the evicted ones would have had a longer

access time. The key difference between this timing and the Evict+Time attack is that Prime+Probe is timing a simple operation and is lass sensitive to time variations. By the end of the experiment, the attacker is able to recover the full 128-bit AES key [Tromer [2010]].

When a time spike is detected, a cache miss has occurred. Because the attacker is continually accessing the cache set with their own data, any time that the user needs to access memory that would map to the cache set, the user will have a cache miss.

This attack has been successfully launched against sandboxing, across virtual memory boarders, across cores [Gruss et al. [2016], Gruss et al. [2017]], and within cloud environments where the processor, OS, and hypervisor are trusted but other cloud tenants are not [Gruss et al. [2017]].

3.2 Flush+Reload

The Flush+Reload technique, which is a variant of the Prime+Probe attack, requires that both cache hierarchy and memory pages are shared between the attacking and victim processes [Falkner and Yarom [2014]] as well as libraries. Otherwise the attack is rendered ineffective [Gruss et al. [2017]]. Flush+Reload utilizes the fact that the last level cache (LLC) is shared across processes. The feature of the LLC that Flush+Reload exploits is the fact that, in modern Intel processors, the LLC is an inclusive cache. This inclusively means that when data is evicted from the LLC, it is also evicted in all other cache levels.

When attempting this attack, the attacker will first set the cache to a known state by calling the `clflush` command in order to flush the memory out of a particular cache line that the attacker is occupying. Since this memory is flushed from all cache levels, any attempt to access memory which is mapped to that cache line will result in a cache miss (see Figure 3.1). Afterwards, the attacker will use one of the two methods of timing that was described earlier. Figure 3.2 gives a visual representation of when the attacker chooses the second method of timing. This figure depicts how the attacker needs to stagger the time between a clflush and when the attacker checks the cache line for a cache miss. If the attacker checks too soon, then line (A) occurs where the victim may have not had time to access the cache line. But if the attacker waits too long, then line (E) can occur, where the victim access the line multiple times and the granularity of the information the attacker can extract is less fine.

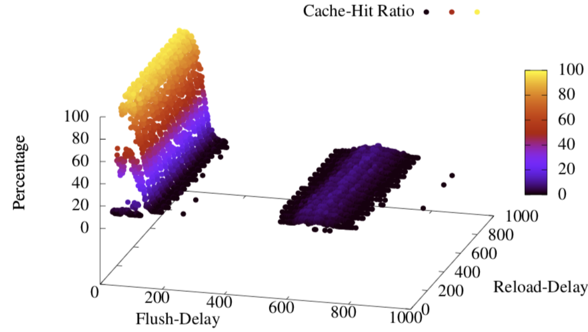

Figure 3.1: “Cache hits observed by a Flush+Reload attacker with the ability to overlap the attack with different segments of the victim’s transaction” Gruss et al. [2017]

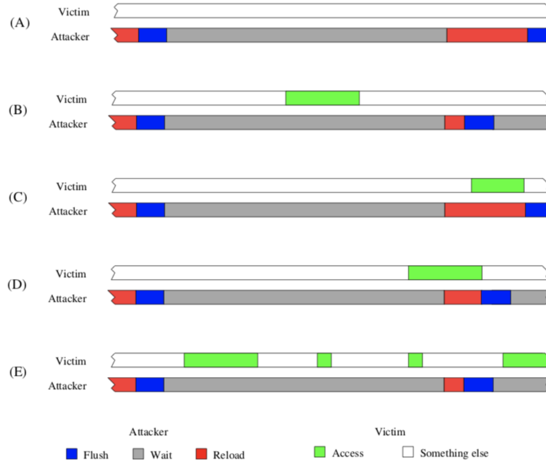

Figure 3.2: “Timing of FLUSH+RELOAD. (A) No Victim Access (B) With Victim Access (C) Victim Access Over- lap (D) Partial Overlap (E) Multiple Victim Accesses” Falkner and Yarom [2014]

A unique dangers of this attack is that it allows the attacker to identify specific cache lines that are being accessed thanks to the fact that memory pages are shared between the attacker and victim processes. Another unique danger is that the attack uses the LLC, which enables the attack to be launched across cores [Falkner and Yarom [2014]], as proven by [Gruss et al. [2017]], who also detail how this attack can be launched across the cloud. One weakness of this attack is that most of the memory activity occurs at the L1 cache level, thus less information will be available for extraction from the LLC activity [Falkner and Yarom [2014]].

An example of this attack is where researchers created the Address Translation Oracle, which flushes one virtual address that maps to a physical address, prefetches another virtual address that potentially maps to the same physical address, and then

reloads the first virtual address. This oracle allows the attacker to check whether two virtual addresses map to the same physical address, which can then be used to bypass stack protections like SMAP [Gruss et al. [2016]].

3.3 Evict+Time

There are three steps an attacker will take for an Evict+Time attack, also referred to as the Evict+Reload attack. First, the attacker measures the time it takes for a victim's process to execute an operation. Then the attacker will make an access call to data that is mapped to a specific cache set or line, ensuring that whatever data was previously stored within is evicted and replaced with the attacker's data. Afterwards, they use the first method of timing where they measure the execution time of the victim's process. By comparing this time to the previously measured time, the attacker is able to tell through time spikes whether the evicted cache was accessed by the victim or not.

For instance, researchers created a Translation Level Oracle (TLO) to bypass protections like ASLR. The oracle enables the attacker to determine the translation table a cache page is from and whether a specific virtual address maps to a specific PA [Gruss et al. [2016]]. In the Translation Level Recovery Attack (TLRA), the attacker performs the Evict+Time attack, and after the eviction occurs and the timing spike has shown that the user had a cache miss, the TLRA is able to use the ATO to learn the precise addresses of the memory from the cache miss. This allows the attacker to circumvent the ASLR protection. The second attack is a Address Translation Attack (ATA) which brute force searches for kernel addresses using ATO. The third attack is the Kernel ASLR Exploit that utilizes the fact that the offset randomization protection for cache pages does not occur on a sub page level. The attack uses TLRA to find the start of the virtual address, bypassing KASLR protection.

Evict+Time has also been successfully executed to access the bits of a secret cipher key instead of the addresses. The attack is able to recover bytes of the plain text and of the cipher key, P_i and K_i, because the lookup table's indices of the accesses is dependent upon the different bytes P_i and K_i. When the two indices are equal for the plaintext, then the second access will result in a cache hit. The indices are equal when $P_1 \oplus K_1 = P_2 \oplus K_2$ [Acıiçmez and Koç [2009]].

The attacker goes through rounds of table lookups of $P_i \oplus K_i, i \in \{0...n\}$ and times their execution. When either $P_1 \oplus K_1 = P_2 \oplus K_2$ or $P_1 \oplus P_2 = K_1 \oplus K_2$, a cache hit occurs. The number of bits recovered depends on the number of elements in a cache line. This type of attack has also been successfully launched against the kernel ASLR [Gruss et al. [2016]] and bypassing supervisor mode execution protection (SMEP) [Acıiçmez et al. [2007]].

Chapter 4

Spectre Attack

At the beginning of 2018, two independent parties discovered and reported three new speculative-execution based side channel attacks. The attack names were Spectre Variant 1 [Kocher et al. [2018]], Spectre Variant 2 [Horn [January 3, 2018]], and Meltdown (or Spectre Variant 3)[Lipp et al. [2018]]. The attacks have been shown to bypass many of the previous side channel attack defenses [Kocher et al. [2018]]. Spectre Variants 1 and 2 are especially dangerous because they are able to be executed with user privileges and have the capacity to go deep into arbitrary memory and leak it to the attacker. While there has been a proven patch for Meltdown, the Spectre attacks, especially the Variant 1, have been reported to be much more difficult in patching [Lipp et al. [2018], Kocher et al. [2018]].

4.1 Spectre Variant 1

Spectre Variant 1 is a *bounds check bypass attack*, which is an attack that exploit how branch predictors perform speculation in order to access protected memory. A branch predictor is exploitable through the asynchronous timing between when the condition from a conditional statement is evaluated and when the speculation is evaluating the array access within the loop. The attacker sets up this exploitation by first executing the code multiple times with valid inputs, training the branch predictor to think that the next input will also be valid. Once the branch predictor has been trained, the attacker is then able to make a call with a malicious input that would be rejected by the conditional. If the timing is right, then the speculation will have executed the code within the loop before the conditional has fully evaluated.

An example code of this is given in Figure 4.1, where the `if` statement condition checks if the input `x` from the user is safe. With speculation, the code `exploited &= array2[array1[x]*10]` is executed prematurely and may accidentally access the memory where `key` is stored instead of the memory where `array2` is stored [Kocher et al. [2018]]. An attacker, with user permissions, can use speculation with a malicious input that, through speculation, will access the key.

```
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
/*******************
Spectre Variant 1 Exploitable Code
*******************/
int exploited = 0;
int array1Size = 8;
int array1[64]={1,2,3,4,5,6,7,8};
int array2[300];

char *key = ``Top secret info";

void exploitableFunction(int x){
    if(x < array1Size){
        exploited &= array2[array1[x]*10];
    }
}
```

Figure 4.1: An example of C code that is vulnerable to the Spectre Variant 1 attack. In the code `array2` is accessed within the conditional if statement. This condition and array access are vulnerable to the speculative execution.

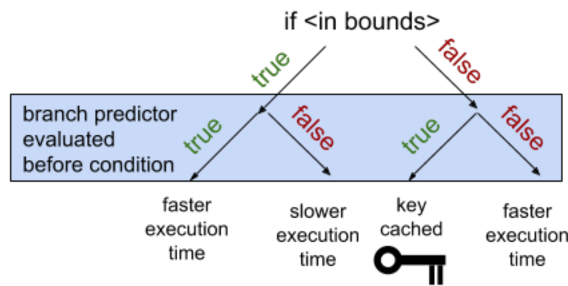

Figure 4.2: A visual representation of the possible paths that can be taken in an if statement.

Figure 4.2 demonstrates the four cases that can occur as the bounds check and speculation are both executed using the code listed in Figure 4.1. In the first, second, and fourth cases, either the input was correct when the speculation was faster, or the speculation occurred after the conditional was evaluated, and all had no unexpected side effects. It is in the third case that demonstrates how a faster speculation will be able to obtain and cache the information before the conditional evaluated false and the process rolled back its states to pre-speculation.

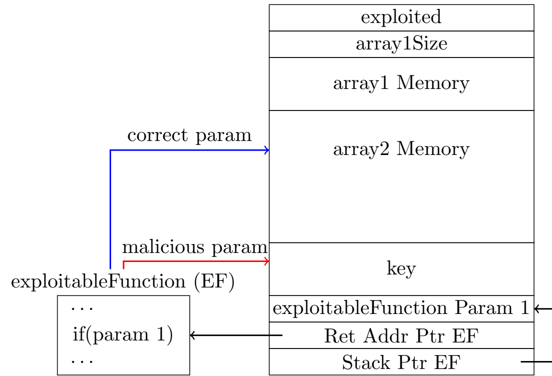

Figure 4.3: Memory stack configuration of C code vulnerable to Spectre Variant 1. The two colored arrows depict where in the memory a correct and an incorrect user input will try and access.

Figure 4.3 shows the memory layout for the example C code in Figure 4.1. At the middle is the memory allocated for `array2`'s data. The green arrow shows where in memory a valid user input will access. The red arrow shows how a launched timing attack with an invalid user input, which fails the bounds check at `if(x < array1Size)` in Figure 4.1, would try an access memory outside of `array2`'s memory. In the Figure 4.3 example, if speculation occurred with the red arrow's attacking input, then data from `key` would have been read and loaded into the cache.

Once the data has been loaded into the cache, the attacker is able to use all three of the previously described microarchitecture attacks to leak the data. For instance, the attacker might use Flush+Reload or Prime+Probe to find the location in `array2` the data came from, and then can use `array2[k*10]` to find the key. The attacker might also use Evict+Time by calling the function again with an in bounds input to find the location in `array2`.

A proof of concept for Variant 1 utilized two extended Berkeley Packet Filters (eBPF) to make the attack [Horn [January 3, 2018]]. An eBPF captures and filters network packages that match specified parameters set by the user. eBPF are able to be attached to a code path in the kernel, and when the code is run, the eBPF is executed [Fleming [2017]]. The eBPF were chosen instead of Berkeley Packet Filters (BPF) because eBPF has arrays and array pointers that allow the branch prediction exploit to occur in the kernel. One eBPF is used to conduct the attack by checking different offsets from an array to the userspace address. The other utilizes the Flush+Reload technique to leak the information by being repeatedly called by the program to point to the userspace memory area [Horn [January 3, 2018]].

4.2 Spectre Variant 2

Variant 2 is also known as *Branch Target Injection* [Horn [January 3, 2018]], *Exploiting Indirect Branches*, or *Poisoning Indirect Branches*. The attack method uses the fact

```
adc   edi, dword ptr   [ebx+edx+13BE13BDh]
adc   dl,byte ptr [edi]
```

Figure 4.4: Example of a disassembly code that acts as the Spectre gadget

that the execution of indirect branches of one process will result in the process making many calls to the cache and main memory. The state of the cache shared between processes means that if these indirect branches fill cache lines with their data, then the user branch will be forced to have cache misses when it accesses data that maps to the same cache lines [Kocher et al. [2018]].

Variant 2's proof of concept utilized Flush+Reload as the means to create the covert channel, but the authors state that the Prime+Probe method is theoretically also able to work. The Prime+Probe method was discussed as being easier to use because it does not require the attacker to know the user-kernel virtual address of a guest page [Horn [January 3, 2018]].

To perform this attack, the attacker's first step is to train the target's branch predictor into speculating incorrectly. This is done by taking advantage of the fact that the branch predictor utilizes the branch history buffer (BHB) in order to better predict indirect calls that can have multiple targets. The BHB keeps track of a set number of last taken branches. By continually executing the code with malicious inputs, the attacker fills the BHB with erroneous branches and thus change the predictions made by the target's branch predictor [Horn [January 3, 2018]].

Once the target's branch predictor has been trained, the attacker then utilizes a *Spectre gadget* that will act as its covert channel in leaking the information found by the incorrect branch predictor's speculation. A Spectre gadget is a fragment of code that, when speculatively executed, will transfer the user's data to the covert channel. Figure 4.4 gives an example of a speculative gadget. When speculatively executed, this gadget will read a 32-bit value from the address m, where `m=ebx+edx+13BE13BDh`, into the register `edi`. The second instruction then fetches the m into the cache. The attacker may then use Flush+Reload to infer the values obtained from the victim process. In this example, the attacker must have control over the `ebx` and `edi` registers in order to create the covert channel.

4.3 Spectre Variant 3: Meltdown

Spectre Variants 1 and 2 were discovered and reported together while Variant 3 was discovered and reported separately. However, all three variants are often discussed and analyzed together due to their similar natures and because they were found around the same time frame. They all take advantage of microarchitecture vulnerabilities. Meltdown is further reaching than the first two variants because it is able to attack with escalated user privileges. The attack circumvents the hardware encoded isolation protection that CPUs uses by taking advantage of asynchronous instruction execution that is used to optimize performance. The attack is able to execute code on a victim's

process with escalated privileges when the attacker has escalated privileges on their own process. This is achieved due to the fact that manipulating one indirect branch will affect another.

When Meltdown is executed, the attacker has access to the kernel memory from user space. Once the memory has been accessed, the attacker will use the Flush+Reload technique to create a covert channel. This covert channel is capable of recovering a full byte at a time, and by the end of the attack the entire kernel memory can be dumped.

Meltdown is more limited than Spectre Variant 1 and Spectre Variant 2 because it is specific to many Intel and some ARM processors and the KAISER patch has been proven to stop Meltdown[Lipp et al. [2018]].

Chapter 5

Spectre Attack Defenses

Of the three Spectre Variants, the mitigations available for the Variant 1 and Variant 3 have a minimal impact upon the system performance while the impact of the mitigation for Variant 2 is more noticeable [Myerson [January 9 2018]].

5.1 Emergency Patches

Soon after the Spectre Variants were reported, major computer and CPU developers, including Microsoft and Intel, released a series of emergency patches. The emergency patches prevented attacks by untrusted code, such as downloaded apps or web browsing, and were released through Windows Updates and silicon microcode updates. These updates isolated an applications binary or code, ensuring that the application cannot access unauthorized memory within the users Windows Server [Myerson [January 9 2018]]. The emergency patch worked by running all of kernel mode code with branch speculation restricted [Lyigun [March 1 2019]]. These patches can be applied to physical servers and virtual machines (VM)[Myerson [January 9 2018]].

It was noted by Microsoft that Windows systems manufactured 2015 or earlier would have a more noticeable decrease in system performance after the emergency patch was installed [Wycislik-Wilson [2018]]. Benchmarking by both Microsoft and Intel on the performance impact caused by the incorporation of their emergency patches has shown that while the Windows 10 computer with the latest Intel CPUs (2016-era PCs with Skylake, Kabylake, or newer CPUs) are minimally affected, older models (Windows 8 or older with silicon 2015-era PCs or older) have a noticeable decrease in system performance [Myerson [January 9 2018]]. Intel used the SYSmark 2014 SE benchmark on 8th Generation Core processors and solid state storage. The benchmark scores showed an average performance impact of 6% with the individual tests ranging from 2% to 14% impact [int [January 9 2018]]. Developers at Intel state that average computer users who perform common tasks like accessing photos or writing documents should not face any significant performance impact [int [January 9 2018]].

5.2 Microsoft BIOS Update

Microsoft later released an updated Spectre Variant 2 patch using Googles retpoline and this patch is said to reduce the performance impact of Spectre Variant 2 significantly [Hruska [October 22 2018]]. *Googles Retpoline* prevents unsafe speculation by replacing all indirect call or jumps in kernel-mode binaries with an indirect branch sequence that has safe speculation behavior. However, the use of retpoline requires that the hardware and OS support for branch target injection is both present and enabled, so Skylake and any later generations of Intel processors are unable to utilize this defense and must use the emergency patches to defend against Spectre Variant 2 [Lyigun [March 1 2019]]. Retpoline is able to be used on any processor where the speculation is not on the contents of the indirect branch predictor. Any processor with that property as well as all AMD processors and any Intel processor with the codename Broadwell and earlier are able to use the retpoline defense.

Microsoft and third parties, such as Phoronix and Techspot, used multiple benchmarks in order to determine whether or not the new retpoline-based defense has a lower performance impact than the first patch. Microsoft ran the Diskspd (storage) and the NTttcp (networking) benchmarks upon Broadwell CPUs. The benchmarks showed about a 25% speedup with Office app launch times and between a 1.5-2x improved throughput [Lyigun [March 1 2019]].

The Techspot team benchmarked the retpoline defense at the start of January, 2018. When they began testing, the Asus Z370 series motherboards were the only motherboard manufacturers to have released an update incorporating the Windows BIOS update for Spectre Variant 2 defense. The Techspot team benchmarked the Core i3-8100 on a Asus TUF Z370-Plus Gaming motherboard. The Techspot benchmarks run are Cinebench R15, measuring the performance of single thread and multi threads; Corona 1.3, measuring image rendering time; Excel 2016 workload; Blender [Ryzen Graphic], another graphic rendering test; and VeraCrypt 1.2.1, an AES Encryption and Decryption test. In the Cinebench R15, and Corona 1.3 Benchmark, there was a slight reduction of performance with -1% for the single thread test, -2% for the multi thread test, and a -3% for the render test (See Figure 5.1 for the render test bar graph). All of these performance reductions are well within the margin of error, and all of the other benchmark tests resulted in almost the exact same score for pre and post update [Walton [January 7 2018]].

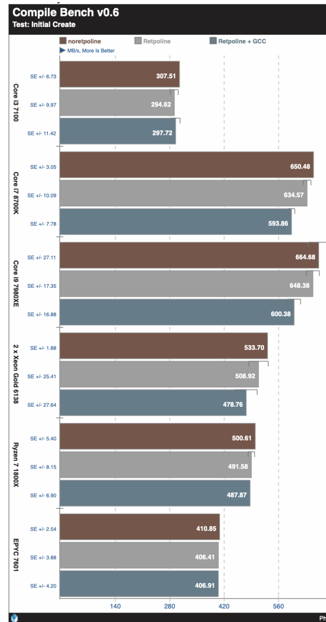

Figure 5.1: One of the benchmarks run by Techspot to test the performance of the retpoline Microsoft patch

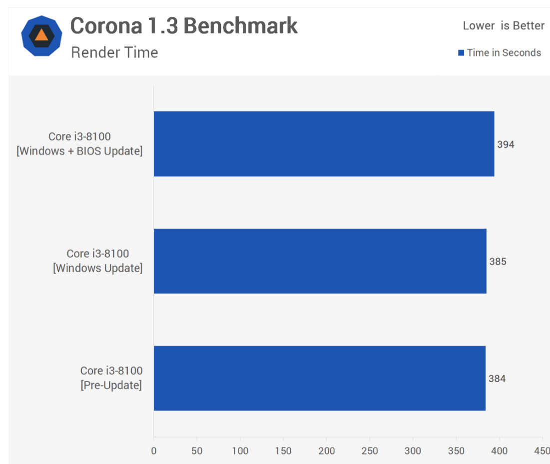

Figure 5.2: A benchmark run by Phoronix when testing the retpoline Microsoft patch

Phoronix, a company centered around enriching the Linux hardware experiment, ran a series of benchmarks on a variety of AMD and Intel processors to see the performance change. The benchmarks used were all part of the Phoronix test suite 7.4.0. The benchmark Flexible IO Tester v2.1.13 tests the random read, random write, and sequential write performance while the benchmark Compile Bench v0.6 tests the initial create (See Figure 5.2 for the initial create benchmark's bar graph). These four tests were performed upon the CPUs Core i3 7100, Core i7 8700k, Core i9 7980XE, 2 x Xeon Gold 6138, Ryzen 7 1800x, and EPYC 7601. Of the CPUs tested, the core i7 8700k

and 2 x Xeon Gold 6138 had a negative performance impact on all of the tests, ranging from a -1% change (random writes with 2 x Xeon Gold 6138) to a -10% change (random writes with core i7 8700k). The other CPUs had an almost or completely zero percent impact [Larabel [January 8 2018]].

5.3 Independent Benchmarking

I conducted a series of benchmarking experiments to observe how the Microsoft BIOS Spectre Variant 2 defense impacts the computational speed of the computer. The three benchmarks chosen were NovaBench, PCMark10, and Geekbench 4. All three of these benchmarks were chosen because they are free, thus easily used by the average Windows computer user, and are a representation of the different sized benchmark suites. All three benchmarks are able to run on Windows 7 or later generations.

The PCMark10 benchmark, as of March 2019, is the latest version of the UL industry standard PC benchmarks. The tests that it comprises are divided into three sections: essentials, covering web browsing and app startup time; productivity, testing read and write speeds of office applications like spreadsheets; and digital content creation, covering image and video editing and rendering [pcm [April 1 2019]]. This benchmark was chosen because it is a a large benchmark suite that tailored towards testing Windows computers. It is a fully documented and has continual maintenance.

NovaBench is a small, quick benchmark that tests the machine's disk read and write speeds, memory transfer speeds, and the machine's CPU and GPU. It is useful because it gives the tester a quick look at the general performance of the computer while not taking up a large amount of memory or taking a long time to run.

Geekbench 4 is a simple and straightforward benchmark suite of tests that performs direct memory accessing, testing the machines performance when using a single core and when using multiple cores. The four tests performed are in encryption/decryption, memory accessing, and computational speed with both integers and floating points. Geekbench 4 is a good benchmark that is a balance between the other two benchmarks in terms of size, execution time, and performance analysis. Geekbench is also a good choice as a medium because it overlaps in some tests with PCMark10: web browsing, image editors, and developer tools.

To disable mitigations for CVE-2017-5715 (Spectre Variant 2):

reg add "HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management" /v FeatureSettingsOverride /t REG_DWORD /d 1 /f

reg add "HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management" /v FeatureSettingsOverrideMask /t REG_DWORD /d 3 /f

Restart the computer for the changes to take effect.

To enable default mitigations for CVE-2017-5715 (Spectre Variant 2) and CVE-2017-5754 (Meltdown):

reg add "HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management" /v FeatureSettingsOverride /t REG_DWORD /d 0 /f

reg add "HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management" /v FeatureSettingsOverrideMask /t REG_DWORD /d 3 /f

Restart the computer for the changes to take effect.

Figure 5.3: The instructions that a user may run on the Windows Powershell in order to turn the Microsoft BIOS Spectre Variant 2 Defense on or off

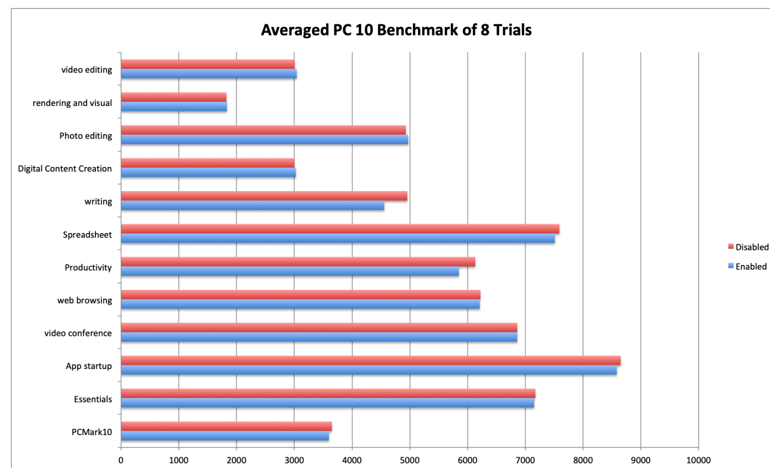

Figure 5.4: Bar graph showing the PC Mark 10 benchmark results.

For the experimental setup, the benchmarks were performed on a 64-bit Windows 10 PC that had a x86 Intel Core i7-6820HQ CPU at 2.70GHz. The machine had 8GB of RAM, 4 cores, and a Dell Int. 06YF8N motherboard. To ensure that the experimentation environment was as static as possible, the following steps were taken. All applications except the benchmarks were completely shut off, the computer was continually plugged in an at max battery, and all of the software necessary to run the benchmarks was downloaded and compiled in order to prevent any memory shifts between testing (See Figure 5.3 for the instructions used to turn the Spectre Variant 2 on and off). Refer to Appendix A for the full data set of the three benchmarks.

Figure 5.4 depicts a bar chart that shows the aggregate scores of the three main test sections of the PCMark10 benchmark suite, the scores of the smaller tests that comprise

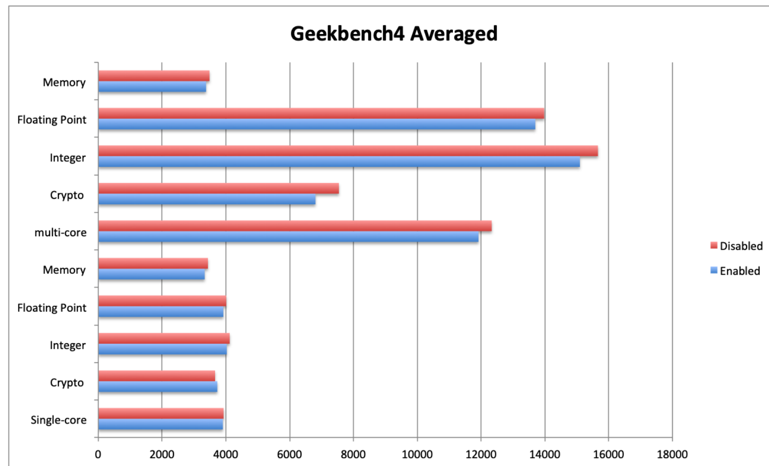

Figure 5.5: Bar graph showing the Geekbench4 benchmark results.

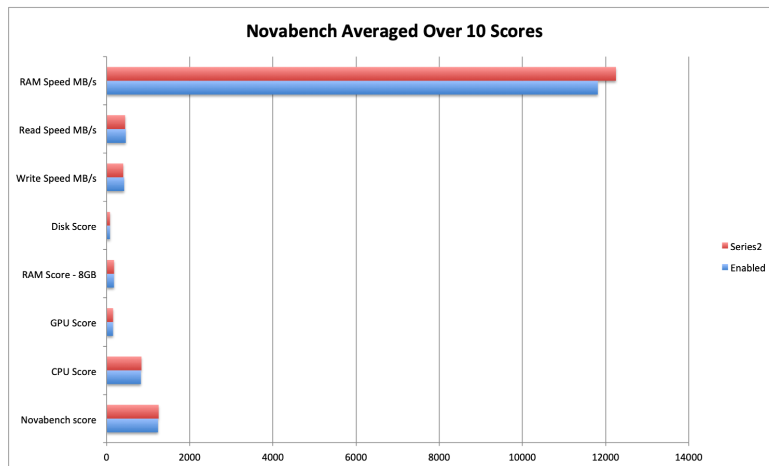

Figure 5.6: Bar graph showing the NovaBench benchmark results.

each section, and the final averaged score of all the scores, labeled PCMark10.

The PCMark10 benchmark program was run eight times when the Microsoft BIOS defense was enabled and eight times when it was disabled. Averaged out, there was a -1.39% decrease in performance with the individual scores, once averaged, ranging from -8.06% to 0.85% Both the essentials and the productivity tests had a slight decrease in performance while the digital content creation section had a slight increase.

In the benchmark testing, Geekbench 4 was run twelve times with the defense enabled and eleven times when it was disabled. On average, there was a -0.58% impact for the single core tests and a -3.35% impact for the multiple cores tests, see Figure 5.5. Between the single core and the multiple core tests, the individual tests have a slowdown that ranges from a 1.93% down to a -9.74%. While the single core crypto results were almost 2% better than before, the multi core crypto results were almost -10% worst. The Geekbench 4 benchmark results indicate that programs run on multiple cores will

suffer more of a performance impact than those on single cores, which lines up with the benchmarks from Phoronix.

For testing, NovaBench was run ten times when the defense was enabled and ten when disabled. The averaged total score shows a slight decrease in performance at -1.17%, with the individual scores ranging from a 5.58% increase to a -3.56% decrease (see Figure 5.6). The disk read and write tests were the only ones to have an increase in performance while the other tests decreased in performance.

Chapter 6

Conclusion

The Spectre attacks are a new series of side channel that pose a serious danger to computer users since they target an integral component of modern CPUs. While different microarchitectures, such as those of ARM and AMD CPUs, provide some defense against the Spectre Variant 3, modern CPUs are still vulnerable to Spectre Variant 1 and Spectre Variant 2. While research has been done in protections against Spectre attacks [Lipp et al. [2018], Kocher et al. [2018], Krzanich [2018], Sloss [January 11, 2018], Miller [March 15 2018]], the current defenses available do not cover all CPUs and cause a negative impact in performance. This impact is more easily seen in older computer generations.

When the defenses were first released, there was a wave of preliminary benchmarks that support Microsoft's claim that their retpoline-based BIOS update has a low performance impact, especially compared to their emergency patch [Walton [January 7 2018]]. Unfortunately the retpoline defense also has compatibility issues with some of the current hardware security improvements, like Intel's Control Flow Enforcement Technology, and may have compatibility issues with future hardware and software [Miller [March 15 2018]]. The current benchmarks do not give the full picture of the impact caused by the defenses. Many of the benchmarks were run in February and before Microsoft released available Intel microcode updates [Miller [March 15 2018]].

I strongly recommend that a lot more research is devoted towards development of defenses against the Spectre Variants and towards documenting how these defenses as well as previous defenses fare in performance impacts. More benchmark testing with a wider coverage in both tests and computer systems that can run them are needed, especially in older generations of computers.

Appendix A

Geekbench4		GB/sec	1/sec[1]	1/sec[2]	GB/sec and ns		GB/sec	1/sec[1]	1/sec[2]	GB/sec and ns
pre	Single-core	Crypto	Integer	Floating Point	Memory	multi-core	Crypto	Integer	Floating Point	Memory
1	4001	3928	4215	4024	3502	12186	7214	15375	14029	3491
2	3993	3915	4246	4004	3428	12197	6996	15442	13998	3494
3	4003	3935	4212	4074	3442	12282	6805	15607	14026	3552
4	3977	3880	4181	4036	3456	12275	7432	15488	14104	3515
5	3986	3745	4208	4001	3523	12073	7503	15370	13636	3471
6	3996	3851	4207	4031	3506	12161	7332	15345	13957	3510
7	3986	3767	4245	3974	3479	12205	7309	15417	14001	3511
8	3978	3792	4178	4014	3521	12248	7419	15479	14025	3520
9	3877	3388	3820	3667	2944	11047	5649	14015	12868	2987
10	3573	3366	3830	3652	2926	11051	5321	14058	12856	3013
11	3530	3326	2792	3601	2886	11135	5602	14172	12941	2975
12	3957	3868	4189	3962	3405	12169	7107	15399	13951	3493
Average	3904.75	3730.08	4026.92	3920	3334.83	11919.08	6807.42	15097.25	13699.33	3377.67
1	3972	3891	4194	4002	3445	12365	7578	15711	14039	3526
2	3985	3783	4220	4017	3458	12370	7452	15768	13967	3558
3	4001	3828	4239	4035	3461	12322	7670	15562	14115	3508
4	3972	3848	4246	3953	3417	12353	7597	15674	14035	3547
5	4009	4053	4201	4050	3505	11995	7628	15553	13082	3453
6	4013	3914	4276	4020	3434	12429	7512	15578	14131	3522
7	3373	1439	3128	4055	3383	12398	7418	15789	14069	3058
8	4009	3955	4234	4046	3462	12424	7580	15766	14138	3545
9	3975	3834	4224	4003	3409	12364	7637	15740	13958	3558
10	4006	3934	4251	4030	3436	12359	7471	15657	14102	3547
11	3888	3774	4091	3907	3403	12279	7416	15492	14109	3522
Average	3927.55	3659.36	4118.55	4010.73	3437.55	12332.55	7541.73	15662.73	13976.82	3485.82
Disabled - Enabled	22.79	-70.72	91.63	90.73	102.72	413.46	734.31	565.48	277.48	108.15
Productivity Impact	-0.58%	1.93%	-2.22%	-2.26%	-2.99%	-3.35%	-9.74%	-3.61%	-1.99%	-3.10%

Pre	NovaBench score	CPU Score	GPU Score	RAM Score - 8GB	Disk Score	Write Speed MB/s	Read Speed MB/s	RAM Speed MB/s
1	1136	727	155	173	81	439	449	10600
2	1284	878	155	181	70	261	471	12840
3	1152	758	147	168	79	421	446	9135
4	1246	832	155	178	81	437	454	12003
5	1271	856	155	179	81	439	464	12383
6	1253	837	155	179	82	440	469	12336
7	1245	830	155	178	82	446	465	12070
8	1265	849	155	179	82	446	466	12301
9	1252	837	155	178	82	445	463	12138
10	1261	845	155	179	82	447	462	12318
Average	1236.5	824.9	154.2	177.2	80.2	422.1	460.9	11812.4
Post	NovaBench score	CPU Score	GPU Score	RAM Score - 8GB	Disk Score	Write Speed MB/s	Read Speed MB/s	RAM Speed MB/s
1	1202	809	155	176	62	367	256	11645
2	1261	845	155	179	82	49	466	12485
3	1257	840	155	180	82	446	466	12558
4	1263	846	155	180	82	450	463	12608
5	1251	835	155	179	82	445	465	12485
6	1259	843	155	179	82	449	466	12437
7	1256	839	155	179	83	451	471	12413
8	1254	838	155	179	82	449	467	12479
9	1254	838	155	179	82	447	469	12336
10	1254	843	155	174	82	445	464	11041
Average	1251.1	837.6	155	178.4	80.1	399.8	445.3	12248.7
Disabled - Enabled	14.6	12.7	0.8	1.2	-0.1	-22.3	-15.6	436.3
Productivity Impact	-1.17%	-1.52%	-0.52%	-0.67%	0.12%	5.58%	3.50%	-3.56%

1.[MB, Mpixel, Krows, function]/sec

2. ns,FBS,Gflops, [Kpixels,Mpixels, Words, Mpairs,GB]/sec

		FPS and 1/sec	sec	FPS	sec
enabled	PCMark10	Essentials	App startup	video conference	web browsing
1	3626	7246	8914	6851	6230
2	3628	7108	8303	6828	6336
3	3592	7136	8599	6853	6168
4	3581	7175	8603	6886	6194
5	3595	7128	8523	6859	6196
6	3592	7136	8599	6853	6168
7	3581	7157	8603	6882	6194
8	3595	7128	8523	6859	6196
Average	3598.75	7151.75	8583.375	6858.875	6210.25
disabled					
1	3650	7178	8696	6854	6207
2	3662	7169	8654	6854	6213
3	3648	7169	8649	6840	6230
4	3649	7163	8558	6873	6249
5	3659	7171	8656	6845	6225
6	3653	7197	8679	6887	6239
7	3668	7181	8635	6859	6254
8	3607	7151	8667	6859	6152
Average	3649.5	7172.375	8649.25	6858.875	6221.125
Disabled - Enabled	50.75	20.625	65.875	0	10.875
Productivity Impact	-1.39%	-0.29%	-0.76%	0	-0.17%

	1/sec	1/sec	1/sec	FPS and 1/sec	1/sec	FPS	FPS and 1/sec
enabled	Productivity	Spreadsheet	writing	Digital Content Creation	Photo editing	rendering and visual	video editing
1	5802	7656	4398	3079	5047	1876	3085
2	5982	7650	4679	3047	5019	1820	3097
3	5846	7476	4572	3015	4947	1828	3032
4	5781	7445	4489	3013	4968	1826	3017
5	5876	7469	4624	3012	4940	1829	3026
6	5846	7476	4572	3016	4947	1828	3032
7	5781	7445	4489	3013	4968	1826	3017
8	5876	7469	4624	3012	4940	1829	3026
Average	5848.75	7510.75	4555.875	3025.875	4972	1832.75	3041.5
disabled							
1	6114	7617	4908	3008	4923	1827	3028
2	6184	7636	5009	3007	4950	1830	3004
3	6130	7563	4970	3000	4911	1827	3010
4	6152	7557	5009	2994	4929	1822	2989
5	6160	7670	4948	3011	4945	1827	3022
6	6115	7549	4955	3007	4949	1820	3019
7	6227	7629	5083	2998	4927	1826	2998
8	5973	7492	4762	2982	4906	1810	2988
Average	6131.875	7589.125	4955.5	3000.875	4930	1823.625	3007.25
Disabled - Enabled	283.125	78.375	399.625	-25	-42	-9.125	-34.25
Productivity Impact	-4.62%	-1.03%	-8.06%	0.83%	0.83%	0.50%	1.14%

Bibliography

PCMark10 Technical Guide. *Futuremark Corporation*, pages 1–83, April 1 2019. `https://s3.amazonaws.com/download-aws.futuremark.com/pcmark10-technical-guide.pdf`.

Intel Offers Security Issue Update . *Intel Newsroom*, January 9 2018.

Onur Acıiçmez and Cetin Kaya Koç. Microarchitectural Attacks and Countermeasures. In *Cryptographic Engineering*, pages 475–504. Springer, 2009.

Onur Acıiçmez, Cetin Kaya Koç, and Jean-Pierre Seifert. Predicting Secret Keys Via Branch Prediction. *Topics in Cryptology - CT-RSA*, pages 225–242, 2007.

Daniel J. Bernstein. Cache-timing Attacks on AES. *Prinston*.

David Brumley and Dan Boneh. Remote Timing Attacks are Practical. *Computer Networks*, 48(5):701–716, 2005.

Katrina Falkner and Yuval Yarom. Flush+reload: a high resolution, low noise, l3 cache side-channel attack. In *23rd Security Symposium (Security 14)*, pages 719–732, 2014.

Matt Fleming. A Thorough Introduction to eBPF. *Contegix*, 2017.

Daniel Gruss, Clementine Maurice, and Anders Fogh. Prefetch Side-Channel Attacks: Bypassing SMAP and Kernal ASLR. *ACM SIGSAC Conference on Computer and Communications Security*, pages 368–379, 10 2016.

Daniel Gruss, Julian Lettner, Felix Schuster, Olya Ohrimenko, Istvan Haller, and Manuel Costa. Strong and Efficient Cache Side-channel Protection Using Hardware Transactional Memory. In *USENIX Security Symposium*, pages 217–233, 2017.

Jann Horn. Reading Privileged Memory with a Side-channel. *Project Zero*, 3, January 3, 2018.

Joel Hruska. New Google Patch Reduces Spectre Performance Impact to Noise. *ExtremeTech*, October 22 2018.

Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. *40th IEEE Symposium on Security and Privacy (S&P'19)*, abs/1801.01203, 2018.

Paul C. Kocher. Timing Attacks on Implementations of Die-Hellman, RSA, DSS and Other Systems. In *Anual International Cryptology Conference*, pages 104–113. Springer, 1996.

Brian Krzanich. Advancing Security at the Silicon Level. *Intel Newsroom*, 2018.

Michael Larabel. Benchmarking Linux With the Retpoline Patches for Spectre. *Phoronix Media*, January 8 2018.

Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, et al. Meltdown: Reading Kernel Memory from User Space. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, 2018.

Mehmet Lyigun. Mitigating Spectre Variant 2 with Retpoline on Windows. *Microsoft*, March 1 2019.

Christopher Vinckier Marko Aho. Computer System Performance Analysis and Benchmarking.

Matt Miller. Mitigating Speculative Execution Side Channel Hardware Vulnerabilities. *Microsoft Security Response Center (MSRC)*, March 15 2018.

Terry Myerson. Understanding the Performance Impact of Spectre and Meltdown Mitigations on Windows Systems. *Microsoft Security*, January 9 2018.

Colin Percival. Cache Missing For Fun and Profit. BSDCan, 2005.

Treynor Sloss. Protecting our Google Cloud Customers from New Vulnerabilities Without Impacting Performance. *Google Cloud*, January 11, 2018.

Eran Tromer. Efficient Cache Attacks on AES, and Countermeasures. *Journal of Cryptology*, 23(1):37–71, 2010. Springer.

Steven Walton. Patched Desktop PC: Meltdown & Spectre Benchmarked. *Techspot*, January 7 2018.

Mark Wycislik-Wilson. Microsoft and Intel Reveal Just How Much Meltdown and Spectre Patches Slow PCs. *Betanews*, 2018.

YongBin Zhou and DengGuo Feng. Side-Channel Attacks: Ten Years After Its Publication and the Impacts on Cryptographic Module Security Testing. *IACR Cryptology ePrint Archive*, 2005:388, 2005.