Meltdown & Spectre: Exploring and Expanding the PoCs

Juliana Furgala, Annie Chen
Meltdown focus, Spectre focus
Tufts University
juliana.furgala@eecs.tufts.edu, annie.chen@tufts.edu

Abstract

For years it was commonly assumed that hardware optimizations result in performance gains without introducing any security vulnerabilities. With the rise of Meltdown and Spectre this longheld assumption has been disproved. These key vulnerabilities rely on out of order execution and speculative execution respectively.

In this project, both Meltdown and Spectre exploits were examined and reproduced to better understand the extent of these vulnerabilities. Reproducing the Meltdown and Spectre exploits from the Proof of Concept (PoC) programs from their respective papers were successful after the correct machine set up, environments, and libraries [13, 16, 10]. After reproducing the code samples, additional enhancements were made.

For Meltdown, these enhancements were adapting the demos to allow users to run them successfully (primary) and combining a demo with new user activity to capture binary that resulted from download actions (secondary). Meltdown was both successfully reproduced and enhanced as planned. The key to enabling user run capabilities was adapting the kaslr demo, the results of which the rest of Meltdown relied upon.

For Spectre, the goal of the extension was to generate more accurate results when extracting data from a timing based cache side channel. This was achieved through implementing a dynamic cache hit threshold tool. The implemented tool was able to measure at runtime the cache hit latency for a machine that it is running on, and use that value in the Flush+Reload side channel. When this tool was integrated with the Spectre PoC code, it made the Spectre exploit more accurate by being able to extract data more precisely from the side channel.

1 Introduction

First we present background concepts underlying the Meltdown and Spectre vulnerabilities, our respective research focuses. Then we explore the setups required to recreate them. Finally we explain the enhancements made and the results uncovered in these experiments.

For both the Meltdown and Spectre exploits, the PoC scripts were successfully reproduced. To further understand the extent in which these exploits can be realistically deployed, we implemented additional enhancements to both exploits. To not further aid malicious attackers in conducting these exploits, the exact implemented programs are not posted anywhere, but rather talked about in a higher overview in this paper. Psuedo code and algorithms are described and shown, and the results are analyzed, but the exact programs remain private. We believe that by demonstrating the process, results, and conclusions, readers will be convinced that Meltdown and Spectre are indeed real threats to our systems, not just theoretical ideas of possible exploits.

2 Project Background

While the research has been split halfway for one individual focus on Meltdown and the other on Spectre, the motivation and goals of both topics remain similar. Our motivations for the projects were two-fold: to better understand the execution of these security exploits and how they are conducted based on core principles of computer architecture; to enhance the publicly available scripts into their full, sophisticated forms, thereby demonstrating the extent of the dangers of Meltdown and Spectre.

Our goals stem from these motivations. For our respective security vulnerabilities, we sought to exploit them on an unpatched machine to better understand how an attack can be executed. Once this was accomplished we then expanded the functionalities of the public Proof of Concepts (PoCs). For Meltdown, this was modifying the PoC scripts to run with user access rather than administrative access. With Spectre it was incorporating additional software to optimize the PoCs for improved accuracy.

In the below sections, details of Meltdown and Spectre are provided to better understand the concepts underlying these security vulnerabilities.

2.1 Meltdown

Discovered in parallel by three sets of researchers[16, 11], Meltdown shook up the computer science computer when it was announced. Meltdown takes advantage of a security vulnerability accidentally introduced in the name of performance. As the instruction pipeline became a bottleneck, hardware optimizations were developed to further improve processor performance. The key optimizations that enable Meltdown are prefetching and out of order instruction execution. The assumed threat model is that of remote execution and no administrative access.

In essence Meltdown uses the early, out of order execution of an instruction before the permission check to extract information out of the machine. Despite the cache flushing that occurs upon a thrown permission error, there is a microarchitectural footprint left behind by the access attempt. The presence of this footprint strips away the processor security promise of memory isolation, allowing any user to access sensitive information from other users or the kernel.

The second part of this exploit is a timing-based side channel attack on the cache. Upon scanning the memory with a tactic known as Flush+Reload, the formerly accessed page will hit more quickly than the other pages in memory, based on the hit-miss cache threshold. Then the attacker can extract information contained within that page. This process can be repeated to dump as much of the memory as desired.

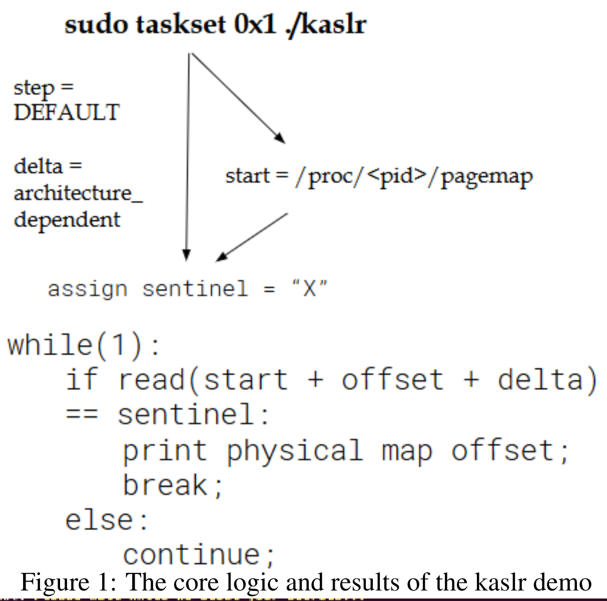
For information on common questions and company-specific responses to Meltdown see [8].

2.1.1 *Core Concepts*

- **Prefetching** is the advance fetching of future instructions into the instruction pipeline based on predictions made from local and/or global branch history structures or from other techniques.
- **Out of order execution** is premised on pipelining, where instructions are being handled simultaneously on different subpaths and stages of the processor. As some stages or operations may take longer to execute, such as division, this can lead to instructions being executed (though not committed) out of order.
- **KASLR**, short for kernel address space layout randomization, changes the location of the kernel each time the computer is turned on, theoretically ensuring that an attacker could not reliably hope to read the kernel. As Meltdown indicates, this assumption does not hold.
- **The hit-miss threshold** is the estimated and fluctuating boundary between which a memory access is a hit or a miss. This threshold can be determined by averaging many accesses and is instrumental in carrying out the Meltdown attack. For a code sample, see Section 3.2.3.
- **Flush+Reload** is one tactic by which an attacker can attempt to determine if a page falls crosses the hit-miss cache threshold. For a code sample, see Listing 1.

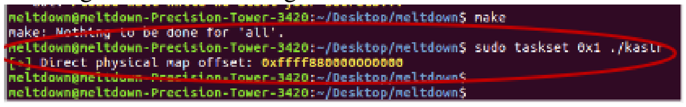
  Listing 1: Abbreviated Flush+Reload logic from the Meltdown PoC repository [10]

```
static int flush_reload(void *ptr) {
  uint64_t start = 0, end = 0;

  start = rdtsc();
  maccess(ptr);
  end = rdtsc();

  flush(ptr);

  delta = end - start;
  if(delta < cache_miss_threshold)
  {
    return 1;
  }
  return 0;
}
```



```
assign sentinel = "X"

while(1):
    if read(start + offset + delta)
    == sentinel:
        print physical map offset;
        break;
    else:
        continue;
```

Figure 1: The core logic and results of the kaslr demo



```
meltdown@meltdown-Precision-Tower-3420:~/Desktop/meltdown$ make
make: Nothing to be done for 'all'.
meltdown@meltdown-Precision-Tower-3420:~/Desktop/meltdown$ sudo taskset 0x1 ./kaslr
[+] Direct physical map offset: 0xffff880000000000
meltdown@meltdown-Precision-Tower-3420:~/Desktop/meltdown$
meltdown@meltdown-Precision-Tower-3420:~/Desktop/meltdown$
```

2.1.2 *Proof of Concept Demonstrations (Demos)*

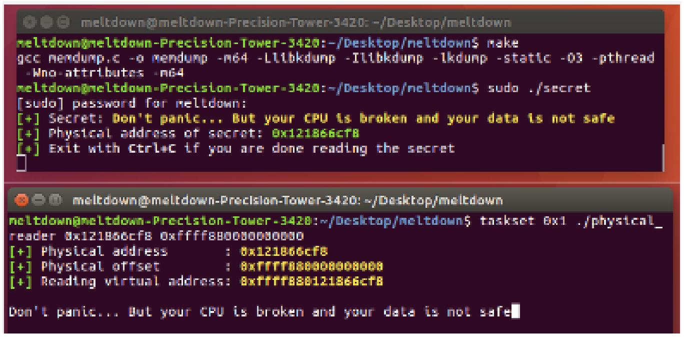
The following demos are all available at the IAIK meltdown PoC public repository[10]. Their purposes are summarized below.

- **Checking for machine support**
  This demo checks if memory can be read by the libkdump sublibrary on the machine being used.
- **Breaking KASLR**
  The kaslr demo 'breaks' KASLR by getting the start location of memory from the /proc/<pid>/pagemap file. It then calculates the physical map offset with this start location and common, default machine values for the offset and delta. For further detail on the core logic of the kaslr demo see Figure 1. Note that this demo requires administrative privileges and will not run without sudo.
- **Testing read reliability**
  Having the physical memory map offset located in the kaslr demo, the script attempts to read the memory from that point onwards, recording its overall percentage of successful accesses. Note that this goes through all readable memory on the machine.
- **Reading physical memory**
  For this demo the user artificially works with a secret so that the attacker can detect it in real time. In a real scenario this could look like a user logging into a service with their credentials. It also requires the physical memory map offset. For further detail on the core logic of the physical memory reader demo see Figure 2.
- **Dumping memory**
  The demo iterates through the entirety of memory or until a

User	Attacker
sudo ./secret	sudo taskset 0x1 ./physical_reader 0x121866cf8 0xffff880000000000

```
size_t vaddr =
libkdump_phys_to_virt(phys);

while(1):
    print(read(vaddr);
    fflush(stdout);
    vaddr++;
```

Figure 2: The core logic and results of the physical memory reader demo



```
meltdown@meltdown-Precision-Tower-3420:~/Desktop/meltdown$ make
gcc memdump.c -o memdump -m64 -Llibkdump -Ilibkdump -lkdump -static -O3 -pthread
 -Wno-attributes -m64
meltdown@meltdown-Precision-Tower-3420:~/Desktop/meltdown$ sudo ./secret
[sudo] password for meltdown:
[+] Secret: Don't panic... But your CPU is broken and your data is not safe
[+] Physical address of secret: 0x121866cf8
[+] Exit with Ctrl+C if you are done reading the secret
```

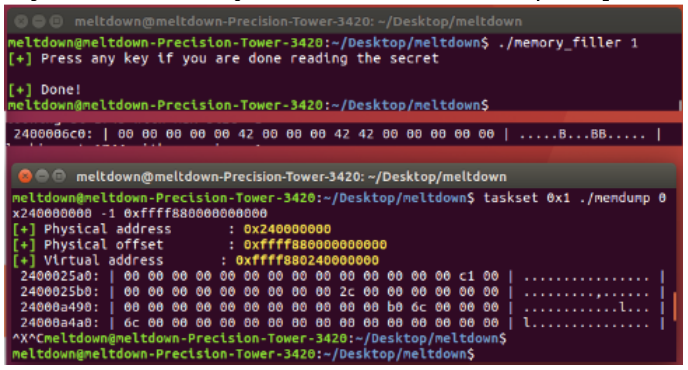
```
meltdown@meltdown-Precision-Tower-3420:~/Desktop/meltdown$ taskset 0x1 ./physical_
reader 0x121866cf8 0xffff880000000000
[+] Physical address       : 0x121866cf8
[+] Physical offset        : 0xffff880000000000
[+] Reading virtual address: 0xffff880121866cf8

Don't panic... But your CPU is broken and your data is not safe
```

sudo taskset 0x1 ./memdump
0x240000000 -1 0xffff880000000000

↓

```
assign char* buffer = malloc(width);

while(running and delta < size):
    buffer[delta % width] = read(vaddr +
                                     delta);
    if buffer is full:
            print buffer contents;
```

Figure 3: The core logic and results of the memory dump demo



```
meltdown@meltdown-Precision-Tower-3420:~/Desktop/meltdown$ ./memory_filler 1
[+] Press any key if you are done reading the secret

[+] Done!
meltdown@meltdown-Precision-Tower-3420:~/Desktop/meltdown$
```

```
 2400006c0: | 00 00 00 00 00 42 00 00 00 42 42 00 00 00 00 00 | .....B...BB..... |
```

```
meltdown@meltdown-Precision-Tower-3420:~/Desktop/meltdown$ taskset 0x1 ./memdump 0
x240000000 -1 0xffff880000000000
[+] Physical address       : 0x240000000
[+] Physical offset        : 0xffff880000000000
[+] Virtual address       : 0xffff880240000000
 2400025a0: | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 c1 00 | ................ |
 2400025b0: | 00 00 00 00 00 00 00 00 00 2c 00 00 00 00 00 00 | .........,...... |
 24000a490: | 00 00 00 00 00 00 00 00 00 00 00 b0 6c 00 00 00 | ............l... |
 24000a4a0: | 6c 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | l............... |
^X^Cmeltdown@meltdown-Precision-Tower-3420:~/Desktop/meltdown$
meltdown@meltdown-Precision-Tower-3420:~/Desktop/meltdown$
```

given end address and prints all memory in that range. Similarly to the reliability and physical memory reader demos this demo requires the physical memory map offset from the kaslr demo.

For further detail on the core logic of the memory dump demo see Figure 3.

2.1.3 Variations

The IAIK repository PoC presents three main variations of Meltdown: regular/null, no null, and fast. These three versions of Meltdown are very similar with only one or two assembly lines different between each. null uses an additional argument with a null value, which no null does not, giving it its name. Fast is simply a shorter version of the no null with one less instruction.

All three variations exploit the same vulnerability. However, some are more effective with certain machines than others, depending on the specific machine instruction set and configuration.

2.2 Spectre

Spectre is a hardware security vulnerability that was discovered around the same time as Meltdown (2018). The threat model of this exploit is one that can be executed remotely and requires no admin access. Unlike Meltdown, where typically kernel memory is attacked, Spectre generally aims at exploiting browser memory.

Spectre's exploit heavily relies on branch prediction and speculative execution. Branch prediction and speculative execution are techniques that modern processors use to improve performance. Essentially, if a branch is about to be executed and is dependent on a memory value that is in the process of being read, the processor will attempt to predict the direction of the branch (true or false) and start loading instructions based on this prediction. When the memory value has arrived, the processor will either discard and flush the instructions if the prediction was incorrect, or commit the speculative computation if correct. A spectre attack involves "inducing a victim to speculatively perform operations that would not occur during correct program execution and which leak the victims confidential information via a side channel to the adversary" [13].

From exploiting branch prediction and speculative execution, Spectre is able to violate much of the security assumptions of operating systems, containerization, just-in-time (JIT) compilation, and countermeasures to cache timing and side-channel attacks [13]. Furthermore, due to the nature of Spectre being a hardware vulnerability that relies on performance optimization techniques that is found in almost all modern processors, Spectre, like Meltdown, poses as a serious threat for billions of devices with microprocessors from Intel, AMD, and ARM.

There are two general variations of Spectre that differ in the method of achieving speculative execution and the method used to leak the information.

2.2.1 *Variant 1: Exploiting Conditional Branches*

The first variant of Spectre involves a bounds check bypass through exploiting conditional branches [13]. We demonstrate in detail how this variant works as the Spectre extension in this project is focused on this variant. Consider the example in Listing 2. In the first line of the code, a bounds check is made to ensure that the value x is within the size of the array. This check prevents the processor from reading sensitive data in memory outside of array1. If x is a value that is larger or equal to the size of array1, then the correct execution would be for the body of the if-statement to never be run at all. However, before the result of the bounds check is known, the processor can speculatively load the code in the body of the if-statement if it predicts the branch to be true. Thus, an adversary can simply train the branch predictor to keep predicting true by first running this code segment with valid values of x (within the size of array1) multiple times. This leads the branch predictor to believe the if-conditional will likely to be true. Then, the adversary can supply a value for x, with x = (address of a secret byte to read) - (base address of array1). During the speculative execution, array1[x], which resolves to the secret byte k in the victim's memory, will be computed to find the address of array2[k * 4096].

When the bounds check is finally made but discovered to be mispredicted, the speculatively loaded instructions would be flushed from the pipeline, but the the read of array2 at an address dependent on k is still within the cache state. To complete the attack, the adversary is able to measure which location in array2 is in the cache from a side channel measure, such as Flush+Reload. This reveals the value of secret byte k, since the victims speculative execution cached array2[k*4096]. This example clearly shows that "speculative execution and branch prediction may allow unauthorized disclosure of information to an attacker with local user access via a side-channel analysis" [5].

Listing 2: Conditional Branch Example from the Spectre Paper [13]

```
if (x < array1_size) {
    y = array2[array1[x] * 4096];
}
```

2.2.2 *Variant 2: Exploiting Indirect Branches*

The second common variant of Spectre relies on the attack mistraining the branch predictor in one context such that in another context (i.e. process), the branch predictor will go to a malicious destination chosen by the attacker [4]. This destination would be a specific code fragment in the victim's address space that would be able to transfer the victim's sensitive information into a covert channel upon speculative execution. In other words, the adversary can train branch predictors to go in a direction such that the program will speculatively execute to locations that would never have originally occurred during legitimate program execution. More details and examples of Spectre variant can be read about in the original Spectre paper [13].

For both variants, a commonly used side channel to extract the information is through the L1 data cache timing. Though there are other forms of side channels from other microarchitectural components, including the instruction cache [1], lower level caches[9], and branch history[2], the project focuses on using the L1 data cache. The L1 data cache is chosen as the side channel to reproduce the Spectre attack as it is one of the most common effective side channels that can be ported on different machines, regardless of the variations in microarchitecture. This was discovered and explored as the main research of the Spectre focus, which will be discussed later on in detail.

3 Experimental Methodology

3.1 Meltdown

3.1.1 *Initial Attempt*

Originally the machine used had an i5 processor chip with the Ubuntu 14.04 operating system. This proved to be problematic on two counts.

The i5 chip lacked the new optimization capabilities of the i7 processor chips, especially TSX. TSX, or Transactional Synchronization Extensions, allows for greatly improved performance and pipelining. Since Meltdown relies on a timing-based side channel attack (using the hit-miss threshold) to determine the memory accessed from the pagemap file, this improvement makes Meltdown more reliable and repeatable. In contrast the i5 chip lacks this capability and would require a potentially more sophisticated script to maintain the same repeatability. Note that even on an i7 chip Meltdown is not always repeatable.

As well the first operating system used was Ubuntu 14.04 which is different than its successor Ubuntu 16.04. This means that the Ubuntu 14.04 OS did not react fully to the regular Meltdown variation like Ubuntu 16.04 did. While the i5 chip difference is significant, the two OS versions seemed to require different variations of Meltdown. Likely there are many OS-related factors that go into this but they were not further explored at this juncture.

3.1.2 Final Setup

With these discoveries in mind I switched out the i5 machine with one that had an i7 chip, specifically the Intel(R) Core(TM) i7-6700K. The operating system run on this machine was the more updated, yet still unpatched, Ubuntu 16.04 with an x86_64 instruction set. The results pictured above in Figures 1, 2, and 3 are from these testing conditions.

To get the original scripts running, first all dependencies needed to be satisfied, mainly taskset, glibc_static, and linux-tools-4.4.0-31-generic. Now that the environment was setup the scripts were able to run.

3.1.3 Modifications

Having successfully run the demos on the machine I turned to extending its use cases.

One shorter case I tried was of recording memory accesses. The original authors of the Meltdown paper[16] recorded log-in credentials. Instead I looked at memory accesses via apt-get upgrade. These accesses, which took place during the active run of the scripts, were observed and the binary was recorded in terminal. An attacker could use this to spy on a user's download activity at any point in time, assuming that their machine remained unpatched and vulnerable.

The main modification I focused on was changing the script for user use. With kaslr as the only demo that requires the use of sudo, as it accesses a protected file for page information, the lack of user run privileges hinges on this demo. By adapting this demo to allow for an unauthorized user to break KASLR the rest of the demos can be run in the exact same way as they were before, with the physical memory map offset from the kaslr demo. Therefore this was the demo that I targeted. The new threat model with the user modification assumes remote execution with an attacker that has only user permissions (versus the administrative privileges needed for the original demo).

Since there are accountability concerns with script kiddies the original, public demo lacks the key feature of user usability. The modified code from this research will not be posted for the same reason. In place of code for the modification I offer the following steps:

- **Fork** and have the child process access the kernel memory.
- **Catch** the resulting exception.
- Use the cache as a covert channel through **Flush + Reload**.
  - Determine the **hit/miss threshold** of the cache.
  - **Iterate** through the pages and check the timing of each access.
  - Considerably less **access time below threshold** indicates the cached page.
  - **Dump** the cached value.
- Continue this as many times as desired.

3.2 Spectre

3.2.1 Reproducing Spectre Proof of Concept

The Spectre attack outlined in Appendix C of the paper Spectre attacks: Exploiting speculative execution (2018) was successfully reproduced after some modification [13]. The code was running on a machine with the Darwin Kernel Version 14.5.0: x86_64 operating system, and a processor of Intel(R) Core(TM) i5-5250U CPU @ 1.60GHz. Note that this architecture and OS version was appropriate to reproduce this attack because it is on a processor that is known to be vulnerable to Meltdown and Spectre because of its architecture, and an OS version early enough that the patches for these vulnerabilities have not been installed. The program can be run on GNU Make and GCC.

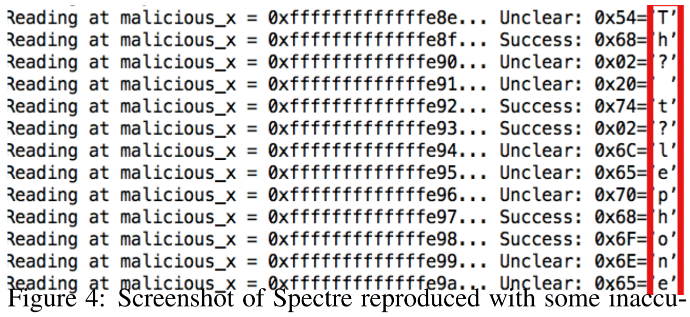
3.2.2 Initial Setup Modifications

First, in order to reproduce the PoC code, the code had to be modified to include the correct libraries and system calls that translate to the corresponding instructions that the CPU is able to handle. Depending on the version of the CPU, certain instructions will need to be disabled in order for the program to run correctly. This is because certain newer instructions that the code uses were not yet introduced for older CPUs. If required, alternative functions need to be implemented to run the code that can do similar functionalities. For example, the `clflush` function is used to flush out specific data from the cache, but was only introduced with the SSE2 instruction set [17]. This means that most CPUS pre-Pentium 4 (prior to the SSE2 being integrated) will need to disable this instruction, and find a means to mimic the same functionality. The `clflush` function is used to complete the Flush+Reload side channel technique to flush a specific cache line from all levels of the processor's cache hierarchy. For processors without the SSE2 instruction set, an alternative is to implement a function that iteratively reads memory addresses at cache block size intervals. The iterations will run a number of times until the processor cache is filled, depending on the size of the cache. This works as a method of "flushing" out the cache by loading arbitrary data into it.

After these checks of the code were made and the necessary functions rewritten to ensure that the PoC code can successfully run on the machine, the attack was able to be reproduced, though not at 100% accuracy. Figure 4 shows a part of a screenshot of the program output when the code ran. The full image is not shown due to space constraints. However, as one can see, the characters that are outputted are "Th? t?lephone" when in fact the actual victim bytes to be read are "The telephone". The "?" characters that is seen shows that the program was not able to accurately and clearly deduce the exact byte that was loaded. The same kind of pattern goes on for the rest of the string (not pictured in the screenshot) in which out of 40 bytes, 13 bytes were unclear and "?". The next section goes into detail on why the results were not completely accurate and successful, and how improvements can be made to more accurately read the data.

3.2.3 Creating a Dynamic Cache Hit Threshold Measurement Tool

After thoroughly examining the code, it was discovered that during the cache hit threshold value was hardcoded as part of



```
Reading at malicious_x = 0xffffffffffffe8e... Unclear: 0x54='T'
Reading at malicious_x = 0xffffffffffffe8f... Success: 0x68='h'
Reading at malicious_x = 0xffffffffffffe90... Unclear: 0x02='?'
Reading at malicious_x = 0xffffffffffffe91... Unclear: 0x20=' '
Reading at malicious_x = 0xffffffffffffe92... Success: 0x74='t'
Reading at malicious_x = 0xffffffffffffe93... Success: 0x02='?'
Reading at malicious_x = 0xffffffffffffe94... Unclear: 0x6C='l'
Reading at malicious_x = 0xffffffffffffe95... Unclear: 0x65='e'
Reading at malicious_x = 0xffffffffffffe96... Unclear: 0x70='p'
Reading at malicious_x = 0xffffffffffffe97... Success: 0x68='h'
Reading at malicious_x = 0xffffffffffffe98... Success: 0x6F='o'
Reading at malicious_x = 0xffffffffffffe99... Unclear: 0x6E='n'
Reading at malicious_x = 0xffffffffffffe9a... Unclear: 0x65='e'
```

Figure 4: Screenshot of Spectre reproduced with some inaccuracy

the program. This cache hit threshold is important, as the Flush+Reload technique heavily depends on this threshold as a way of determining whether a memory access was a cache hit or cache miss. I predicted that it was because this cache hit threshold was not accurate for the machine that the code was run on that resulted in inaccurate readings for the byte. The reasoning behind this is that cache thresholds vary for each machine with a different computer architecture, as a cache hit time on each machine would vary, depending on properties like cycles per instruction, memory hierarchy, physical processor attributes. In other words, one can imagine that a processor from 2018 would have a much lower cache hit time than a processor from 2005, as technological developments would have improved the cache hit time by a significant margin.

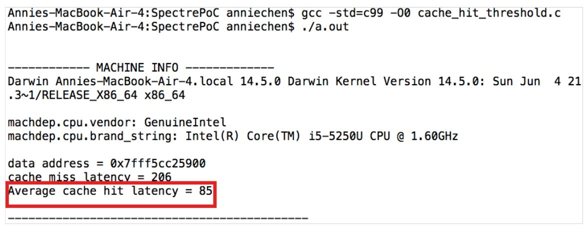
To address this issue, I built an additional program that can measure a machine's cache hit threshold dynamically. Due to the same dangers as noted for the Meltdown modification, the exact code of this program will not be posted. Rather, only the algorithm will be shown. The main area of interest is in the following:

Listing 3: Main algorithm to measure the cache hit threshold of a machine

```
temp = data[0]; // Data access: cache miss
for (int i = 0; i < RUNS; i++) {
      mm_mfence(); // Start serialize
      time1 = rdtsc(); // Timer
      temp = data[0]; // Data access: hit
      total_time += rdtsc() - time1; // Timer
      mm_mfence(); // End serialize
}
cache_threshold = total_time / RUNS;
```

Listing 3 shows the pseudocode that demonstrates how to compute the cache hit threshold of a machine. In essence, the time for a data access is measured across multiple runs and averaged out, to provide a more accurate measurement of the cache hit time. The function `rdtsc` is used as a timing mechanism, and is a system call that returns the processor time stamp in terms of clock cycles [15]. Thus, the result of the cache hit threshold is in terms of clock cycles, rather than in the unit of seconds. This is important because the Spectre PoC code expects the cache hit threshold to be in terms of clock cycles too.

An important function that was missing in the initial development of the code was `mfence`. Without it, when the program



```
Annies-MacBook-Air-4:SpectrePoC anniechen$ gcc -std=c99 -O0 cache_hit_threshold.c
Annies-MacBook-Air-4:SpectrePoC anniechen$ ./a.out


------------ MACHINE INFO -------------
Darwin Annies-MacBook-Air-4.local 14.5.0 Darwin Kernel Version 14.5.0: Sun Jun  4 21
.3~1/RELEASE_X86_64 x86_64

machdep.cpu.vendor: GenuineIntel
machdep.cpu.brand_string: Intel(R) Core(TM) i5-5250U CPU @ 1.60GHz

data address = 0x7fff5cc25900
cache miss latency = 206
Average cache hit latency = 85

--------------------------------------------
```

Figure 5: Output of running the implemented cache hit threshold tool. The red box shows the number of clock cycles a L1 cache hit took. The results are further analyzed in Section 4.2

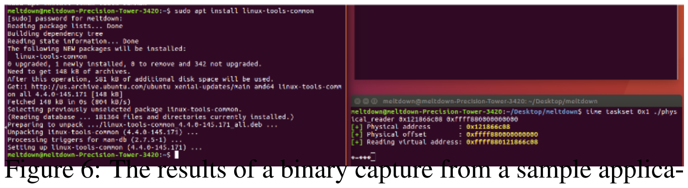
was run multiple times, different cache threshold outputs would occur, varying from 30-150. As one can tell, this range is far too wide to accurately determine the cache threshold. Upon further examination, because of the nature of modern processors to use out of order execution as a means of performance improvement when running any program, the `rdtsc` timing functions were likely being executed out of order, thereby giving a wide, unpredictable variation of results on each run. This is because when `rdtsc` instructions are executed out of order, they lose their value as a timing mechanism.

After this analysis, the `mfence` function was added before the first `rdtsc()` call and another after the last `rdtsc()` call to ensure that the block of code in between keeps its serialization. After this was added, the program started outputting results with a much smaller range of variation between different runs. Figure 5 shows a run of the program. The OS version and CPU chip information are first printed out, as this is helpful in analyzing and understanding the results. Then, the data address of the memory access is printed, as well as the calculated cache miss and cache hit latency. For our purposes, the most important result is the cache hit latency, as that acts as the cache hit threshold for the Spectre PoC program. The tool for calculating the cache hit threshold is now ready to be used and integrated into the Spectre PoC code. This means that in the PoC program, before the exploit happens, the "add-on" tool would have calculated the cache hit threshold at runtime, and then assign that as the cache hit threshold for the Flush+Reload, instead of a single hardcoded number as seen previously,

4 Results and Analysis

4.1 Meltdown

The experiment was generally successful though not as reliable as was desired. Meltdown was indeed capable of capturing binary, which while not printable via terminal can be collected and recorded by an attacker to gain information about a user's activity. To see this in action see Figure 6. This binary could



```
meltdown@meltdown-Precision-Tower-3420:~$ sudo apt install linux-tools-common
[sudo] password for meltdown:
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following NEW packages will be installed:
  linux-tools-common
0 upgraded, 1 newly installed, 0 to remove and 342 not upgraded.
Need to get 148 kB of archives.
After this operation, 581 kB of additional disk space will be used.
Get:1 http://us.archive.ubuntu.com/ubuntu xenial-updates/main amd64 linux-tools-comm
on all 4.4.0-145.171 [148 kB]
Fetched 148 kB in 0s (804 kB/s)
Selecting previously unselected package linux-tools-common.
(Reading database ... 181364 files and directories currently installed.)
Preparing to unpack .../linux-tools-common_4.4.0-145.171_all.deb ...
Unpacking linux-tools-common (4.4.0-145.171) ...
Processing triggers for man-db (2.7.5-1) ...
Setting up linux-tools-common (4.4.0-145.171) ...
meltdown@meltdown-Precision-Tower-3420:~$
```

```
meltdown@meltdown-Precision-Tower-3420: ~/Desktop/meltdown
meltdown@meltdown-Precision-Tower-3420:~/Desktop/meltdown$ time taskset 0x1 ./phys
ical_reader 0x121866c08 0xffff880000000000
[+] Physical address       : 0x121866c08
[+] Physical offset        : 0xffff880000000000
[+] Reading virtual address: 0xffff880121866c08

+-+++
```

Figure 6: The results of a binary capture from a sample application download via terminal

represent programs, images, etc.

Bringing user access to the existing scripts came down to focusing on the kaslr demo. With modifications the kaslr script can then be successfully run without sudo. Modifying this single demo allows a user to derive the physical map offset, which the other demos require as input. This location is a secret that is protected by an administrative access check. Once this information is discovered by exploiting out of order execution then the Meltdown scripts know where to look for kernel and user information. Therefore, changing this single source of input information used by the rest of the demos transforms Meltdown into its regular, more dangerous form.

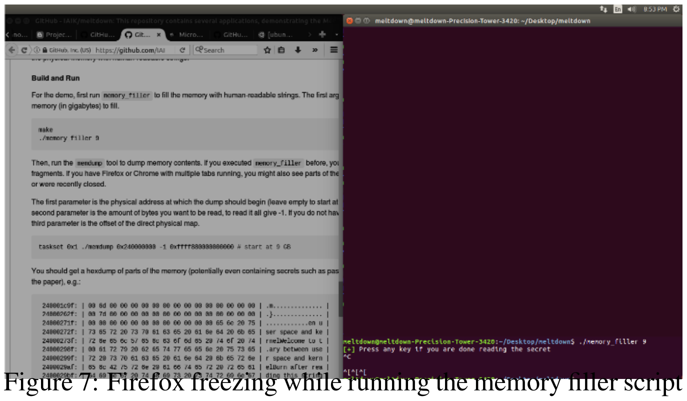
There were some difficulties even with the i7, Ubuntu 16.04 arrangement with the original script. The replacement machine had 1 TB of space, making for long run times for the reliability and memdump demos as well resulting in occasionally wonky behavior. This disk space issue could be resolved through use of partitioning to trick the operating system into thinking there is less memory available than there really is. However, considering that the attacker would not be partitioning a victim's computer as part of their attack, the overt, unexpected behaviors would be observable to the user.

When the memory_filler script that accompanied the memdump demo filled more memory than swap space available (in this case 8 GB) then applications froze (e.g Firefox) and terminals broke, resulting in scrambled inputs, letter swaps, and terminal prompt splicing and duplication. For one example of this see Figure 7. While the user would not likely be running the memory filler script presented by the authors of the Meltdown paper there are similar, realistic scenarios. This is analogous to a user downloading a multi-gigabyte application or running a majorly memory-intensive application of a similar scale.

Progress could still be made for the modified Meltdown script in this research, such as preventing the unintended behaviors and improving repeatability. However, the missing core features of the full Meltdown exploit were implemented.

4.2 Spectre

Once the cache hit threshold tool was integrated, every machine that runs the modified PoC program will now run with its own the cache hit threshold appropriate for its machine. For newer processors, this number can range from around 20-90, depending on the architecture of the machine. For older processors, this number can reach up to 200. To understand what this value means, we need to consider how this number is calculated. As noted in the earlier section, the `rdtsc` instruction in Listing 3 measures the number of cycles, and so the timing measures the number of cycles that occurred in between the two `rdtsc` functions. As intended, only the data access instruction is between



GitHub - IAIK/meltdown: This repository contains several applications, demonstrating the M
no...
Projec...
GitHu...
Git...
Micro...
GitHu...
[ubun...
GitHub, Inc. (US) https://github.com/IAI
Search
8:53 PM

Build and Run

For the demo, first run `memory_filler` to fill the memory with human-readable strings. The first arg
memory (in gigabytes) to fill.

```
make
./memory_filler 9
```

Then, run the `memdump` tool to dump memory contents. If you executed `memory_filler` before, you
fragments. If you have Firefox or Chrome with multiple tabs running, you might also see parts of the
or were recently closed.

The first parameter is the physical address at which the dump should begin (leave empty to start at
second parameter is the amount of bytes you want to be read, to read it all give -1. If you do not hav
third parameter is the offset of the direct physical map.

```
taskset 0x1 ./memdump 0x240000000 -1 0xffff880000000000 # start at 9 GB
```

You should get a hexdump of parts of the memory (potentially even containing secrets such as pas
the paper), e.g.:

```
240001c9f: | 00 6d 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .m.............. |
24000262f: | 00 7d 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .}.............. |
24000271f: | 00 00 00 00 00 00 00 00 00 00 00 00 65 6e 20 75 | ............en u |
24000272f: | 73 65 72 20 73 70 61 63 65 20 61 6e 64 20 6b 65 | ser space and ke |
24000273f: | 72 6e 65 6c 57 65 6c 63 6f 6d 65 20 74 6f 20 74 | rnelWelcome to t |
24000296f: | 00 61 72 79 20 62 65 74 77 65 65 6e 20 75 73 65 | .ary between use |
24000299f: | 72 20 73 70 61 63 65 20 61 6e 64 20 6b 65 72 6e | r space and kern |
2400029af: | 65 6c 42 75 72 6e 20 61 66 74 65 72 20 72 65 61 | elBurn after rea |
2400029bf: | 64 69 6e 67 20 74 68 69 73 20 73 74 72 69 6e 67 | ding this string |
```

```
meltdown@meltdown-Precision-Tower-3420: ~/Desktop/meltdown
meltdown@meltdown-Precision-Tower-3420:~/Desktop/meltdown$ ./memory_filler 9
[+] Press any key if you are done reading the secret
^C

^[^[^[
```

Figure 7: Firefox freezing while running the memory filler script

the two timing functions, as the goal is to time the number of cycles for a cache hit.

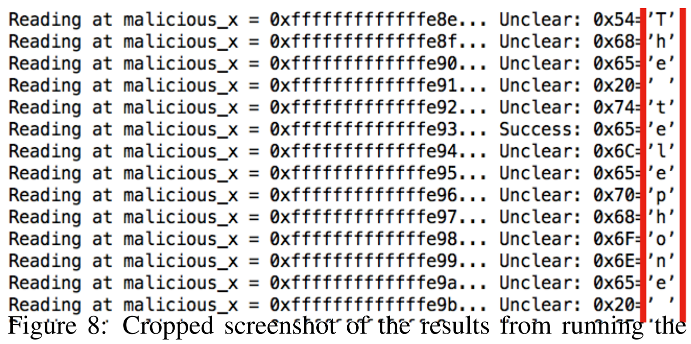
However, even on new processors, a measurement of 20 cycles for a L1 cache access seems extreme, as processors like the Intel Core i7 and Intel Xeon 5500 have an access latency of the L1 data cache of 4 cycles [7]. The reason behind this is that there is overhead for calling the `rdtsc` function itself, and possibly additional overhead when the compiler inserts instructions for the loop. However, this is not an issue when integrating in the Spectre PoC program, because the original program already accounts for this. In other words, during the Flush+Reload code in the program, the code measures a memory access in the same way as Listing 3, thereby also including the overhead given by the loop and `rdtsc` function call.

After running the modified Spectre PoC program, the new results are shown in Figure 8. A cropped screenshot of the results are shown due to space constraints. As we can see from Figure 8, the original "?" from Figure 4 have all been removed. This means that the program, when extracting the secret victim data from the Flush+Reload side channel, was able to more accurately find the right bytes that was cached. However, it is noted that it is not 100% successful on all runs, as there are still a few instances when running the modified Spectre PoC program repeatedly, that a few "?" will still appear in the results. Overall, we see a significant improvement from the initial results in Figure 4. This confirms that the implemented cache hit threshold program did indeed enhance the Spectre PoC. It also shows the importance of having an appropriate cache hit threshold for a given machine. This is logical, as the only way the Flush+Reload technique will work is if it can tell the difference between a cache hit and a cache miss, which relies solely on the cache hit threshold.

Though an improvement in accuracy is generally positively regarded, it also creates a worrisome picture for the future of unpatched machines in a bigger picture. These presented results and analysis shows that the Spectre attack can be tuned and further enhanced to increase its accuracy, and presents a very real threat to a multitude of users.

5 Current Work and Solutions

While there have been a number of OS patches and microcode updates from numerous technology companies like Microsoft, Intel, Apple, a general result from applying these patches and



```
Reading at malicious_x = 0xfffffffffffffe8e... Unclear: 0x54='T'
Reading at malicious_x = 0xfffffffffffffe8f... Unclear: 0x68='h'
Reading at malicious_x = 0xfffffffffffffe90... Unclear: 0x65='e'
Reading at malicious_x = 0xfffffffffffffe91... Unclear: 0x20=' '
Reading at malicious_x = 0xfffffffffffffe92... Unclear: 0x74='t'
Reading at malicious_x = 0xfffffffffffffe93... Success: 0x65='e'
Reading at malicious_x = 0xfffffffffffffe94... Unclear: 0x6C='l'
Reading at malicious_x = 0xfffffffffffffe95... Unclear: 0x65='e'
Reading at malicious_x = 0xfffffffffffffe96... Unclear: 0x70='p'
Reading at malicious_x = 0xfffffffffffffe97... Unclear: 0x68='h'
Reading at malicious_x = 0xfffffffffffffe98... Unclear: 0x6F='o'
Reading at malicious_x = 0xfffffffffffffe99... Unclear: 0x6E='n'
Reading at malicious_x = 0xfffffffffffffe9a... Unclear: 0x65='e'
Reading at malicious_x = 0xfffffffffffffe9b... Unclear: 0x20=' '
```

Figure 8: Cropped screenshot of the results from running the modified Spectre PoC proram with the integration of the dynamic cache hit threshold measurement tool.

updates have been a slowdown in the computer system. These patches have often resulted in additional overhead and decrease in performance that is noticeable for users. Some of the patches actually caused further vulnerabilities, such as the initial Windows Meltdown patch, which opened up arbitrary write access and had to be rolled back[18]. As well, these patches are truly only short-term solutions for a hardware-based problem. The long-term solution will come from chip makers who will need to account for these vulnerabilities in future designs. Until then software patches are the most viable, universal solution.

5.1 Meltdown

Windows[19], Unix-based systems such as Debian[6], and Apple Inc.[3] all introduced patches for Meltdown that led to double-digit performance losses. The reason for this was that many of them limited or disabled speculative execution as a preventative measure. Most of these initial attempts were repeatedly revised and the Windows patch was even rolled back after it was revealed that it caused further problems by allowing for arbitrary write access[18]. In the face of these quick fixes, researchers began to look at longer term solutions.

Research has already been conducted for potential hardware mitigations of the security problem that out of order execution introduces. Note that these solutions are not widely implemented but are rather explorations for future hardware development. One such attempt is InvisiSpec, which ensures safer speculation by storing handled information in a separate buffer that is only committed if the permissions check out [14]. This way speculative and out of order execution is still supported but there is no microarchitectural footprint to follow and therefore no read leak threat. Similarly, SafeSpec created another intermediate structure that is only committed after checks[12]. Both of these solutions rely on the assumption that while memory reading potential is dangerous, speculative execution is a major contributor to machine performance and cannot simply be scrapped.

5.2 Spectre

A likely exploitation of Spectre is through a web-based attack using JavaScript, for example in a malicious ad, to leak private information, session keys, and such, that are cached in the browser. As such, companies with browser products like Google, Mozilla, Apple, and Microsoft have issued new updates for their browsers to reduce the risk of being under a Spectre attack. The success of these patches remain unclear, as the companies often do not provide the technical details of the fix, so researchers are unable to analyze how accurate these patches were able to mitigate Spectre.

Not only have action been taken on browser platforms, but OS companies have also released patches addressing Spectre. For example, Windows have released a few Spectre patches for the two different Spectre variants, but the details of the patches for the Windows OS also remain unclear as they are not fully disclosed to the public [19]. There also appears to be multiple different patches released on different dates that address the same issue (ex/ Spectre Variant 2 [4]). However, due to the lack of details on how this vulnerability was patched, it is still unclear to what extent the patches protect the machines. It should also be noted that a number of patches had to be rolled back due to a negative performance impact.

6 Conclusion

Meltdown and Spectre, which rely on out of order and speculative execution respectively, disproved the former notion that hardware optimizations were relatively independent of security considerations. In this experiment each of us focused on one of these major vulnerabilities, Juliana on Meltdown and Annie on Spectre, reproducing the limited PoCs and extending their functionality. For Meltdown this meant modifying the exploit code for user use. With Spectre, it meant enhancing the existing PoC to be able to produce more accurate results on any machine by dynamically measuring an appropriate cache hit threshold. This paper's contributions are these modifications, an indication that while these attacks require a level of technical experience and sophistication, they are indeed feasible to reenact.

Performing the Meltdown attack on a system often left visible and unintended side-effects that would be observable by the user. Such behavior included frozen applications and broken terminals. Therefore it is likely that a victim of a Meltdown attack would notice some consequences, whether or not the attack successfully dumped secret information. This, however, was not a clear result from running a Spectre attack, thereby making Spectre an even more concerning vulnerability.

Future adaptations or extensions of the Meltdown script would include improved handling of memory accesses. Error throwing associated with memory accesses made for inconsistent and difficult to repeat results. This issue could be more effectively handled in the next iteration of the script.

As for the Spectre research, future work can be done on testing the modified Spectre PoC program on more machines of different architectures. This can produce more concrete and quantitative data on the exact % increase in accuracy from using the integrated tool on each machine. It would also be worthwhile to analyze the differences in cache threshold for both newer and older processors.

All in all, successfully reproducing the Meltdown and Spectre exploits demonstrated how reasonably these attacks can be executed. Then, further enhancing these programs showed the different possibilities of either creating more potential areas of attack or the potential of improving the accuracy of extracting secret victim data.

References

[1] O. Acicmez. Yet another microarchitectural attack: Exploiting i-cache. In *CSAW*, 2007.

[2] O. Acicmez, S. Gueron, and J.-P. Seifert. New branch prediction vulnerabilities in openssl and necessary software countermeasures. In *International Conference on Cryptography and Coding (IMA)*, 2007.

[3] Apple Inc. About speculative execution vulnerabilities in arm-based and intel cpus. https://support.apple.com/en-us/HT208394.

[4] Common Vulnerabilities and Exposures. Cve-2017-5715. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-5715.

[5] Common Vulnerabilities and Exposures. Cve-2017-5753. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-5753.

[6] Debian. Debian security: Spectre meltdown. https://wiki.debian.org/DebianSecurity/SpectreMeltdown.

[7] Dr David Levinthal. Performance analysis guide for intel core i7 processor and intel xeon 5500 processors. https://software.intel.com/sites/products/collateral/hpc/vtune/performance_analysis_guide.pdf.

[8] Graz University of Technology. Meltdown and spectre. https://meltdownattack.com/.

[9] D. Gullasch, E. Bangerter, and S. Krenn. Cache games - bringing access-based cache attacks on aes to practice. In *S&P*, 2011.

[10] IAIK. Meltdown proof-of-concept. https://github.com/IAIK/meltdown.

[11] Jann Horn. Reading privileged memory with a side-channel. https://googleprojectzero.blogspot.com/2018/01/reading-privileged-memory-with-side.html, 1 2018.

[12] C. S. D. E. D. P. N. A.-G. Khaled N. Khasawneh, Esmaeil Mohammadian Koruyeh. Safespec: Banishing the spectre of a meltdown with leakage-free speculation. Technical report, University of California, Riverside; College of William and Mary; Binghamton University, June 2018.

[13] P. Koche, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre attacks: Exploiting speculative execution. Jan 2018.

[14] D. S. A. M. C. W. F. Mengjia Yan, Jiho Choi and J. Torrellas. Invisispec: Making speculative execution invisible in the cache hierarch. In *51st Annual IEEE/ACM International Symposium on Microarchitecture*, 2018.

[15] Microsoft Docs. rdtsc. https://docs.microsoft.com/en-us/cpp/intrinsics/rdtsc?view=vs-2019.

[16] D. G. T. P. W. H. A. F. J. H.-S. M. P. K. D. G. Y. Y. M. H. Moritz Lipp, Michael Schwarz. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium*, 2018.

[17] Oracle. x86 assembly language reference manual sse2 instructions. https://docs.oracle.com/cd/E18752_01/html/817-5477/epmpv.html.

[18] Ulf Frisk. Total meltdown? http://blog.frizk.net/2018/03/total-meltdown.html, 3 2018.

[19] Windows Support. Protect your windows devices against spectre and meltdown. https://support.microsoft.com/en-us/help/4073757/protect-your-windows-devices-against-spectre-meltdown.