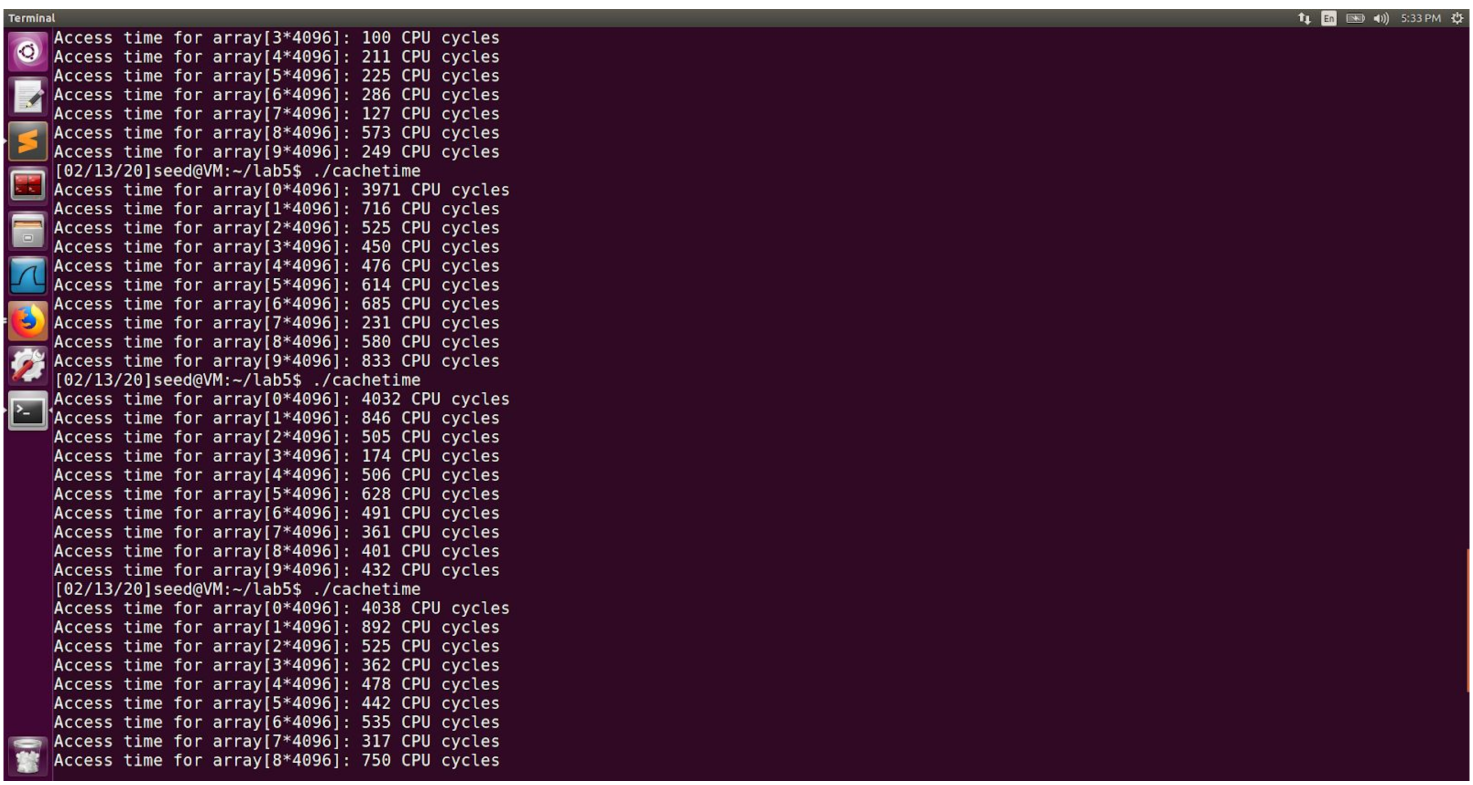
Lab 5 : Meltdown Attack Lab ***Darshan K (A20446137)***

Before we step in to the lab, we set the -march=native flag set, to the gcc. This tells the compiler to enable all instruction subsets supported by the local machine.

Task 1: Reading from Cache v/s from Memory:

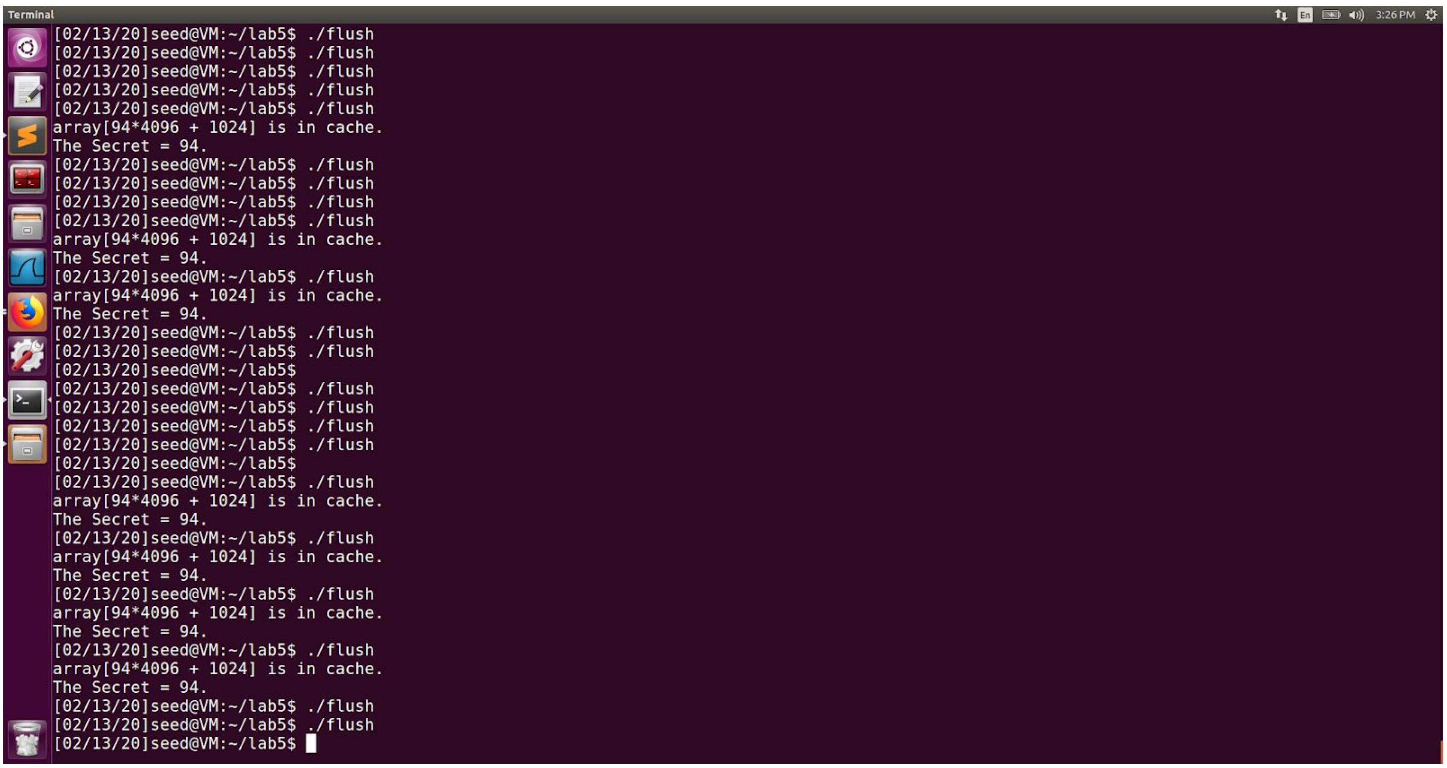
The CacheTime.c program is compiled with -march=native flag set, and the executable is run for 10 times and the observations are noted below. In this program, the data array[3*4096] and array[7*4096] are cached. The highest and the lowest values observed are 362 and 57. And for the non-cached data, the highest and the lowest observed are 4038 and 448. The difference between the highest cached and lowest non cached value is 86. So, we will choose, the threshold as 100 as it is close to the observed values from the few compilations made.



```
Access time for array[3*4096]: 100 CPU cycles
Access time for array[4*4096]: 211 CPU cycles
Access time for array[5*4096]: 225 CPU cycles
Access time for array[6*4096]: 286 CPU cycles
Access time for array[7*4096]: 127 CPU cycles
Access time for array[8*4096]: 573 CPU cycles
Access time for array[9*4096]: 249 CPU cycles
[02/13/20]seed@VM:~/lab5$ ./cachetime
Access time for array[0*4096]: 3971 CPU cycles
Access time for array[1*4096]: 716 CPU cycles
Access time for array[2*4096]: 525 CPU cycles
Access time for array[3*4096]: 450 CPU cycles
Access time for array[4*4096]: 476 CPU cycles
Access time for array[5*4096]: 614 CPU cycles
Access time for array[6*4096]: 685 CPU cycles
Access time for array[7*4096]: 231 CPU cycles
Access time for array[8*4096]: 580 CPU cycles
Access time for array[9*4096]: 833 CPU cycles
[02/13/20]seed@VM:~/lab5$ ./cachetime
Access time for array[0*4096]: 4032 CPU cycles
Access time for array[1*4096]: 846 CPU cycles
Access time for array[2*4096]: 505 CPU cycles
Access time for array[3*4096]: 174 CPU cycles
Access time for array[4*4096]: 506 CPU cycles
Access time for array[5*4096]: 628 CPU cycles
Access time for array[6*4096]: 491 CPU cycles
Access time for array[7*4096]: 361 CPU cycles
Access time for array[8*4096]: 401 CPU cycles
Access time for array[9*4096]: 432 CPU cycles
[02/13/20]seed@VM:~/lab5$ ./cachetime
Access time for array[0*4096]: 4038 CPU cycles
Access time for array[1*4096]: 892 CPU cycles
Access time for array[2*4096]: 525 CPU cycles
Access time for array[3*4096]: 362 CPU cycles
Access time for array[4*4096]: 478 CPU cycles
Access time for array[5*4096]: 442 CPU cycles
Access time for array[6*4096]: 535 CPU cycles
Access time for array[7*4096]: 317 CPU cycles
Access time for array[8*4096]: 750 CPU cycles
```

Task 2: Using Cache as a Side Channel:

In this task it is mentioned to run compile and run this program from more than 20 times. The CACHE_HIT_THRESHOLD defined in this program is 80 which is very close to what we achieved in the task 1. Therefore even after executing the program for more than 20 times, we see no change in the target value. Which is 94, as the difference in the values we observed in the previous step between the max cached and min non cached values is close to the CACHE_HIT_THRESHOLD value set in the program.



Task 3: Place Secret data in Kernel Space:

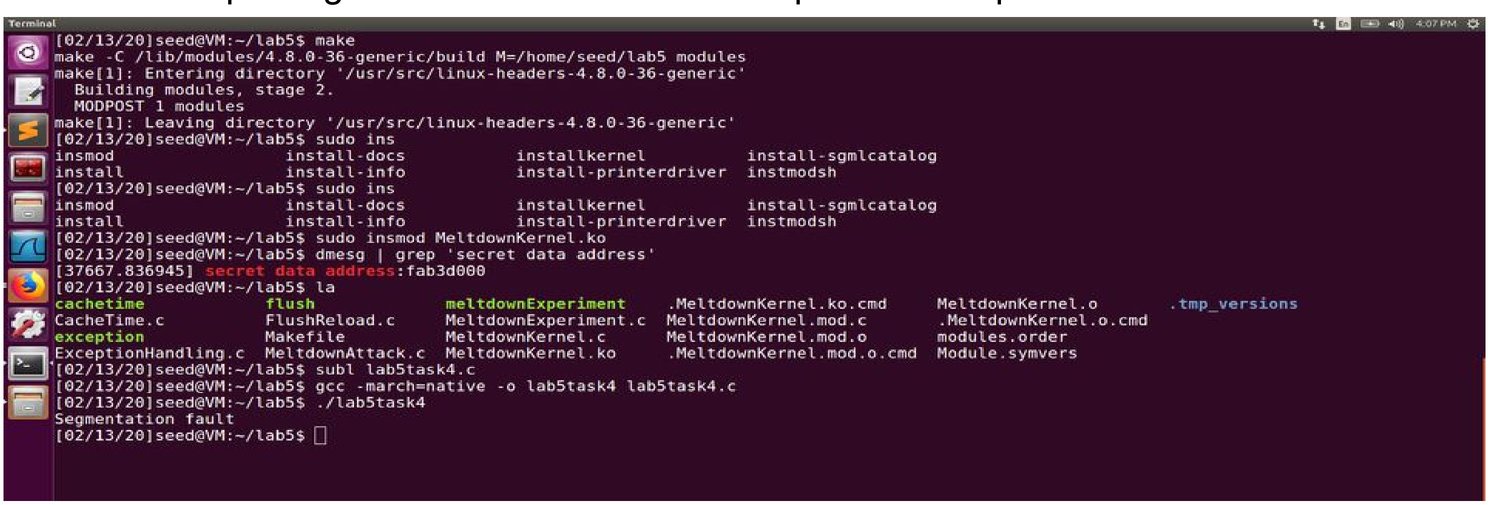
In this task, we copy the code from the labs website and run make and install the .ko file using insmod. This will install this kernel module. We then make use od dmesg command and extract the secret data address using grep and see the address returned as **0xfab3d000.** Here we have stored the data in kernel and we can see that the code can be accessed by the normal user.

Task 4: Access Kernel memory from user space:

In this task, after replacing the address fetched in the previous task, the updated code looks like this:

```
#include<stdio.h>
int main()
{
    char *kernel_data_addr = (char*)0xfab3d000;
    char kernel_data = *kernel_data_addr;
    printf("I have reached here.\n");
    return 0;
}
```

After running this code as a normal user we'll be trying to access the kernel memory. We cannot access the kernel memory as we get segmentation fault error after replacing the address fetched in the previous step.



Task 5: Handle Error/ Exceptions in C:

In this section, the usual exceptions that occur while executing the codes are handled. This handling of exception stops the program from crashing. The SIGSEV signal handled by the operating system is handled by us in the C code, so that the application doesn't crash and continue running by displaying the appropriate message for the exception that is caught.

Task 6: Out of Order Execution by CPU:

The array value 7 is cached that we previously accessed in the meltdown(). In this task we make use of a side channel to check how it is cached. Since we have made use of the exception handler in our C code, the utilisation of the SEGFAULT allows the program to run without crashing. And we can see the value is already cached.

Task 7.1: A Naive Approach (Basic Meltdown Attack):

In this task, we will be making use of FLUSH+RELOAD technique. In this we make use of array[kerneldata*4096] instead of array[7*4096] which brings it to the CPU cache. We fail to launch a successful attack as we get Memory access Violation every time we execute it.

```
[02/13/20]seed@VM:~/lab5$ gcc -march=native -o mE MeltdownExperiment.c
[02/13/20]seed@VM:~/lab5$ ./mE
Memory access violation!
[02/13/20]seed@VM:~/lab5$ ./mE
Memory access violation!
[02/13/20]seed@VM:~/lab5$ ./mE
Memory access violation!
[02/13/20]seed@VM:~/lab5$ ./mE
Memory access violation!
[02/13/20]seed@VM:~/lab5$ ./mE
Memory access violation!
[02/13/20]seed@VM:~/lab5$ ./mE
Memory access violation!
[02/13/20]seed@VM:~/lab5$ ./mE
Memory access violation!
[02/13/20]seed@VM:~/lab5$
[02/13/20]seed@VM:~/lab5$
[02/13/20]seed@VM:~/lab5$
[02/13/20]seed@VM:~/lab5$
```

Task 7.2: Improve the attack by Getting the Secret data Cached (Basic Meltdown Attack):

In this attack, we're still not successful in attacking even after upgrading our attack by adding the secret data to the cache. We're still not in a position to find out whether kernel data is the one that is accessed in the lease amount of time. Even with the secret data cached by CPU, we are still not successful in launching the attack.

```
Terminal
[02/13/20]seed@VM:~/lab5$ gcc -march=native -o meltdownExp MeltdownExperiment.c
[02/13/20]seed@VM:~/lab5$ ./meltdownExp
Memory access violation!
[02/13/20]seed@VM:~/lab5$ ./meltdownExp
Memory access violation!
[02/13/20]seed@VM:~/lab5$ ./meltdownExp
Memory access violation!
[02/13/20]seed@VM:~/lab5$ ./meltdownExp
Memory access violation!
[02/13/20]seed@VM:~/lab5$
```

Task 7.3: Using Assembly code to trigger Meltdown (Basic Meltdown Attack):

To the previous task, we make some improvements in the code, by adding a few lines of assemble instructions before the kernel memory access. The code loops by 400 times. After making the necessary changes we're now able to print the secret value which is still 0.

Task 8: Make the attack more practical:

For each possible secret values we map elements to an array of size 256 and make use of statistical technique. Running the code for multiple times fetch us the secret value of 83. The necessary changes to the code are made and the result is displayed below. It also shows the number of hits.