ASLR on the Line：利用Cache Side Channel来反随机化ASLR

2018-03-27 09:00:01

背景介绍

该文要介绍的工作是利用cache的side channel来实现ASLR的derandomization（即获得在ASLR下的内存地址）。参考的是发表在NDSS17年上的论文ASLR on the Line: Practical Cache Attacks on the MMU，下面首先介绍下背景知识

ASLR

ASLR是防御与内存有关安全漏洞的一个非常有用的措施，由于其overhead比较低，因此被广泛部署在现在的计算机系统中。如图1所示，ASLR是在程序运行时选择将数据和代码放到一个随机的位置，在64位机器上，其有效的虚拟地址位数一般为48位(256TB)，地址的熵比较大，可以有效的防止暴力破解ASLR。

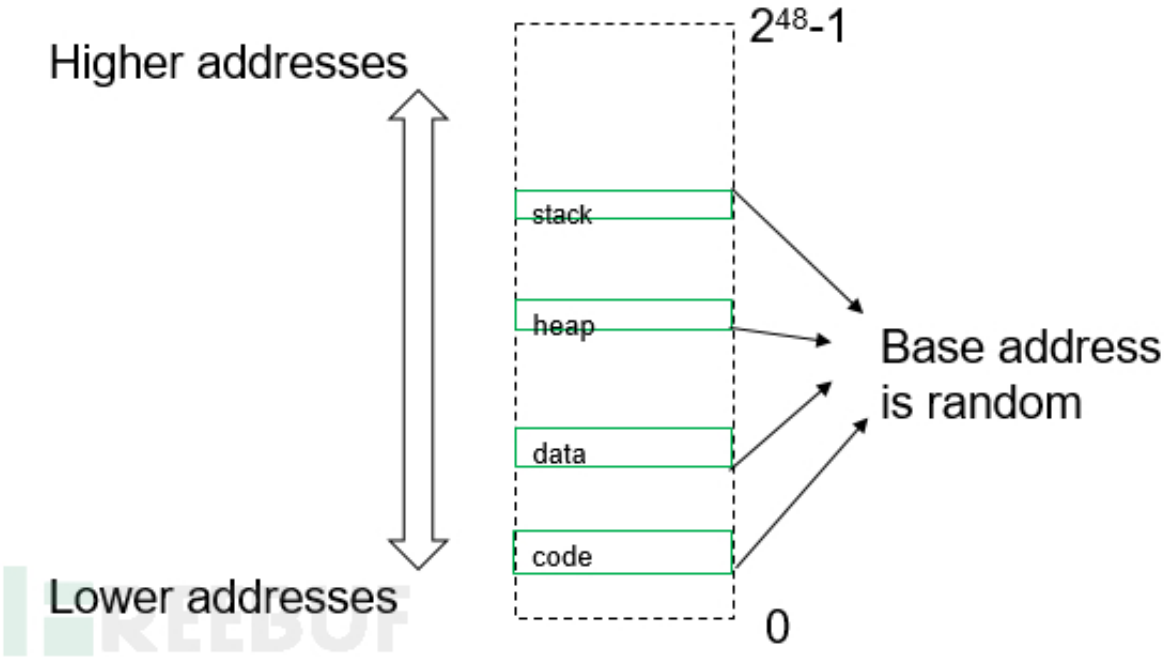

图1 虚拟地址空间

由于ASLR的部署，对程序的code reuse攻击(ret-to-libc, rop等)就变的比较困难。所以一般在进行code reuse攻击之前，会通过一定的手段(内存泄露，buffer over read)来泄露数据段或代码段的地

址，从而有效的实施攻击。但是如果没有buffer over read等漏洞，则很难泄露相关段的地址，从而很难破解ASLR防御。此文[1]利用cache的side channel来实现ASLR的derandomization，此攻击利用了硬件特性来对ASLR进行破解，因此不需要程序有buffer over read等漏洞。

Memory Organization

图2是Intel处理器的内存组织结构，当执行单元（取指令）或者Load/Store单元（访问内存）时，其访问到的地址是虚拟地址，需要将虚拟地址转换成物理地址，在计算机系统中，将虚拟地址转换为物理地址的单元为MMU，其首先检查TLB(Translation Lookaside Buffer)表中是否存在需要查找的Virt Addr, 如果TLB中不存在Virt Addr到其对应的物理地址的映射，则需要进行PT Walk，具体的PT Walk会在下面具体介绍，此时只需要知道其需要查找Page Table表，而Page Table表的内容则可作为数据可能会存储在cache中。当Page Table Walk之后，就找到了Virt Addr对应的物理地址Phys Addr。首先查找cache，看cache中是否存在phys addr地址对应的内容，如果cache中不存在，则从DRAM内存中取得数据。

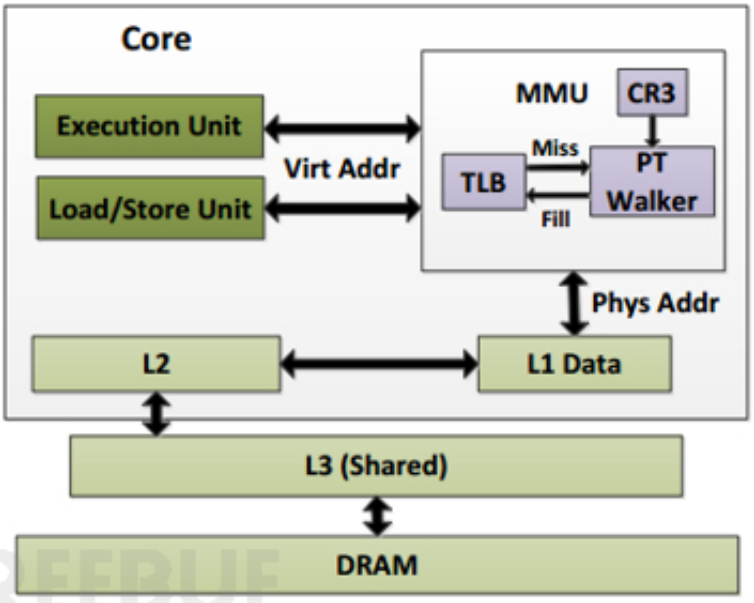

图2 Memory orgainzation in a recent Intel processor[1]

需要注意的是在现在的Intel处理器中（Skylake i7-6700K），cache是分为三层的，每一层都比其下一层小，但是访问速度较快，相当于每一层都是下一层cache的缓存。区中，在第一层L1中，又具体将数据Data和代码Code进行分开，分为L1 code和L1 data；在第二层L2和第三层L3中code和data是放在一起的。需要注意的是，L2中的内容不包含L1中的内容，即如果一个地址的数据或代码存放在L1中，其肯定不会存放在L2中，而L3中的内容即包含L1中的内容又包含L2中的内容，即如果一个地址的数据或代码在L1或者L2中，其一定存放在L3中，相反，如果一个地址的内容不在L3中，其一定也不在L1和L2中。通过此特性，可以使某个内容不在cache L3中，来确保其不在cache中，从而实现cache的side channel攻击。

Page Table Walk

在计算机系统中，程序所能看到的只能是虚拟地址，而真实存在的数据或者代码是存放在物理地址的，所以当程序访问某一虚拟地址va时，需要特定的硬件单元(MMU)来进行虚拟地址到物理地址pa的转换。该转换在计算机系统中是通过查找page table来完成的，该过程叫做Page Table Walk。

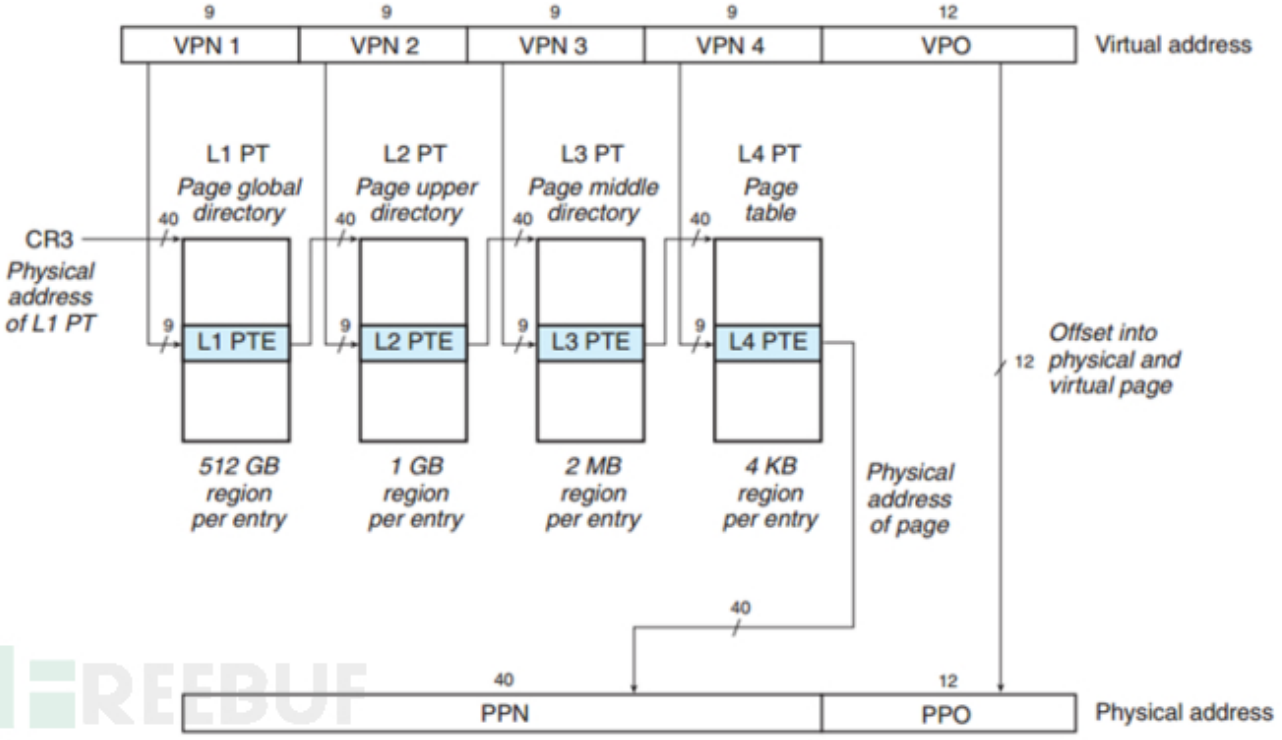

图3 Page Table Walk

图3是我从深入理解计算机系统[2]中截取的Intel i7的page table转换图，由图3可知，其虚拟地址是48位，由于每个页表是4KB(2^12B),所以低12位表示页内偏移。由于是在64位机器中，每一个PTE（page table entry）的大小是64位,即8B（2^3B），所以可用9位（2^12/2^3=2^9）表示页表中的偏移，因此可将虚拟地址划分为4级页表。首先通过CR3寄存器取出一级页表的基地址，再通过VPN1找到二级页表的基地址，如此往复，直到找到物理页的基地址pba, 是的pba+vpo即为要转换到的物理地址pa。

Cache

当从虚拟地址va转换成物理地址pa之后，就可以通过物理地址pa来访问真实的数据。由于CPU处理速度远远大于DRAM的访问速度，如果直接访问DRAM，则可能造成CPU处理能力的浪费，所以在现在的计算机系统中，在CPU和DRAM之间又添加缓存，用来存储CPU最近时间内访问的数据或指令。

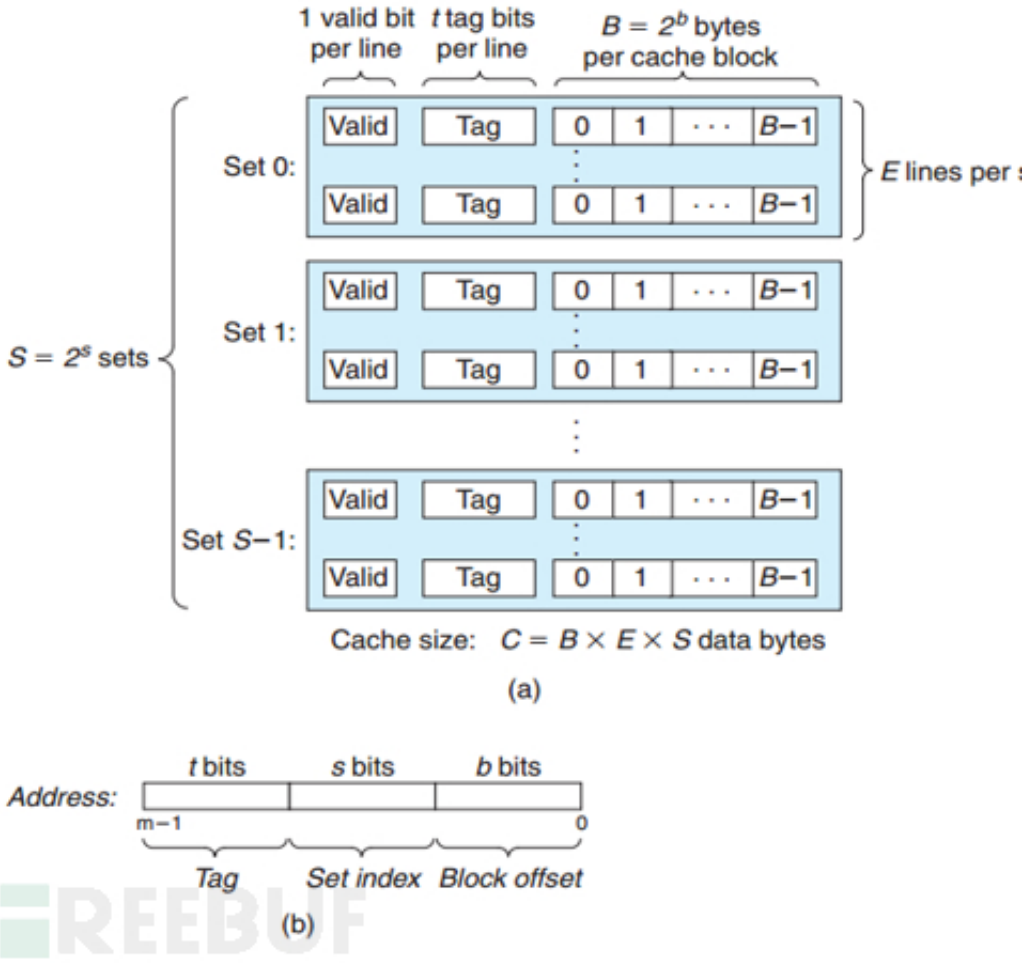

图4 cache结构[2]

图4是当今计算机系统中的cache结构，由图中可知，其将物理地址分成了3部分：块内偏移，组号和标签。每一个cache组可能由多个cache行组成，每个cache行所存储的数据大小是B=2^b字节，则块内偏移就是低b位；cache中存在S=2^s个cache组，则组号为中间s位，剩余的部分就是标签位。当要查找cache数据时，首先通过中间s为找到cache组号，在通过标签位找到其对应的cache行，再通过低b位找到对应数据。

Derandomizing ASLR

由前面一节介绍的基础知识我们可以知道，当MMU进行虚拟地址va到物理地址pa的转换时，首先查找TLB表，如果TLB表miss，则进行page table walk。在进行page table walk时，就是进行4级页表的访问操作。具体的例子如图5所示。

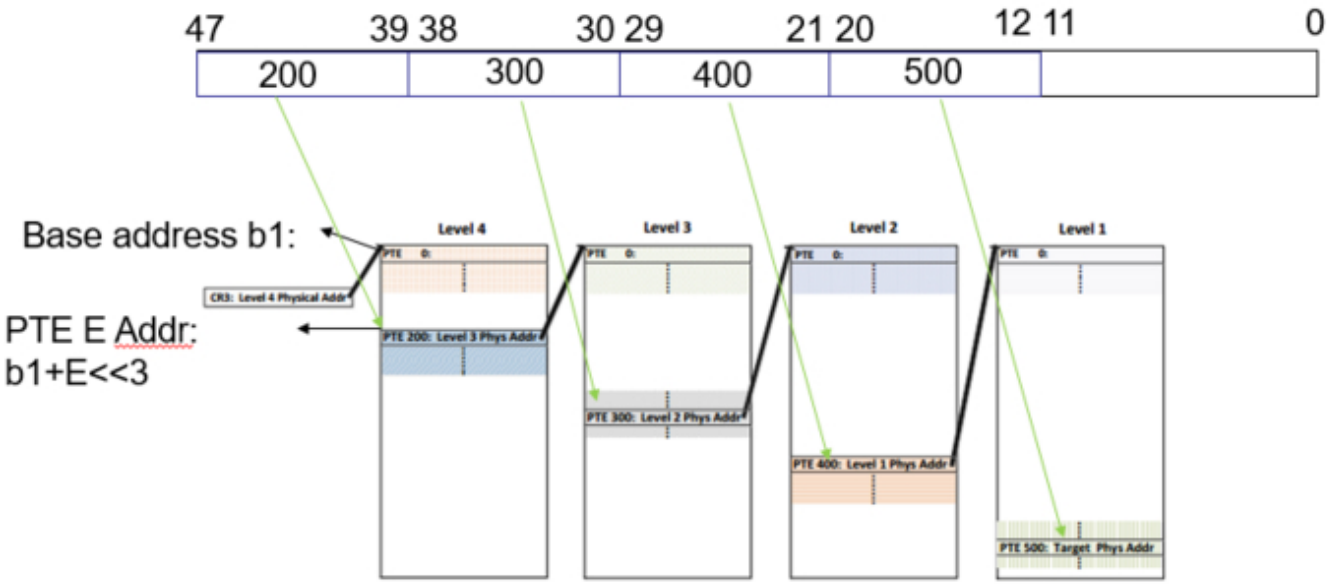

Fig. 1. MMU's page table walk to translate 0x644b321f4000 to its corresponding memory page on the x86_64 architecture.

图5 虚拟地址转换[1]

由图5可知，首先访问level 4页表，通过CR3得到level 4页表基址的物理地址b1，再获得页表项PTE 200（PTE 200物理地址pa1：b1+200<<3）上存的数据，而PTE 200上存的数据放在cache或者DRAM中。如果我们通过某种手段（cache side channel）得知PTE 200存放在cache中对应的组号和块内偏移，则相当于知道了pa1的低b+s位，而如果b+s>6+3，则可以通过cache side channel知道200这个值。同理可以通过cache side channel知道其它三层页表内的偏移值:300, 400, 500。

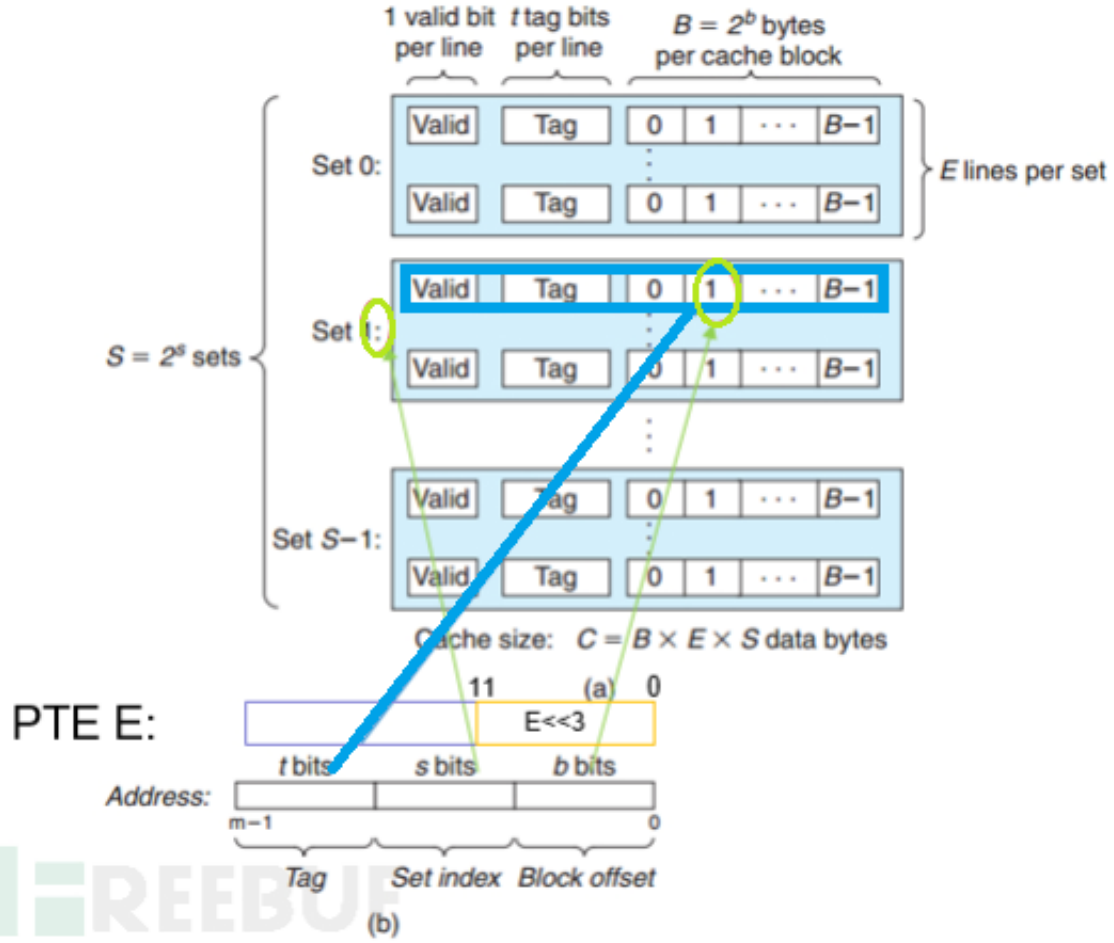

图6 derandomize ASLR

而在当今的计算机系统中，一般都能保证b+s>9。在Intel　Skylake处理器中，其对应的b=6，s=10。

攻击模型

该攻击模型是假设攻击者可以在被攻击者的浏览器中执行JavaScript代码（通过引导被攻击者访问恶意的网站或者伪装的网站）。

JavaScript Timing

如果想要使cache side channel实施成功，则需要比较精确的JavaScript计时器。通过JavaScript计时器可以告诉攻击者其访问的数据是存放在cache中还是存放在DRAM中，因为访问cache的速度远远快于访问DRAM的速度。在以往的攻击中，一般通过JavaScript的performance.now()函数来比较精确的获得访问DRAM和cache的时间差。而现在的浏览器一般都对side channel有了一定的防御措施，即将performance.now()函数的精度将为5us。

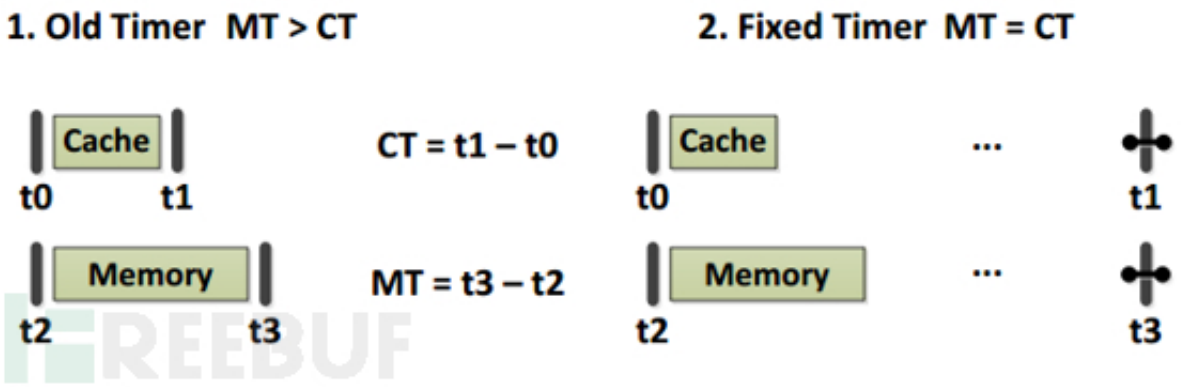

图7 performance.now()进行side channel示意图

由图7可知，当performance.now()精度很高时，可以在访问数据前记一下时t0，访问数据后记一下时t1，将其T=t1-t0就记为访问数据的时间，如果T与CT比较接近，则证明其访问的是cache，否则，其访问的是内存。而如果将performance.now()的精度降为5us之后，就如图7.2所示，无法分辨其访问的数据是在cache中，还是在内存中。

Shared Memory Counter

在当今浏览器中，采用了SharedArrayBuffer函数来申请一个共享的Buffer，使多个线程能够共享该Buffer的数据。

首先申请共享内存:

```
buf = new SharedArrayBuffer(1)
```

在线程1中，进行counter:

```
c=0;
while(buf[0] == 0);

while(buf[0] == 1){
   c++;
}
```

在线程2中，访问内存：

```
buf[0]=1;
operation();
buf[0]=0;
```

这种方法进行的timing与performance.now()的精度没有任何关系，因为它只依赖于CPU的运算速度。

Triggering MMU Page Table Walks

前面也提到过，如果想要进行Page Table Walks，则首先保证TLB缺失，即要操作的目标虚拟地址va不在TLB表中。所以需要先设计一个TLB eviction set，保证TLB中不存在目标虚拟地址va。由于TLB分为i-TLB和d-TLB，即数据和代码不是在一个TLB中，所以在此文[1]中，有两种方式来设计TLB eviction set。首先如果目标虚拟地址va在heap上，则通过ArrayBuffer类型来作为TLB-eviction set，由于ArrayBuffer是页对其的，因此可以有效的预测ArrayBuffer的索引与页内偏移的关系，能够比较有效的设计TLB eviction set。而如果目标虚拟地址va在代码段，则需要生成JITed代码区域。

EVICT+TIME Attack

一般常见的cache side channel有EVICT+TIME[3]，PRIME+PROBE[4]和FLUSH+RELOAD[5]。以上三种cache side channel一般都是在不同的场景下来分辨出某些操作(AES加密算法等)涉及到的某些数据或代码是否在cache中，通过此种方法来还原具体的操作过程。此文[1]采用了两种方法PRIME+PROBE和EVICT+TIME来分辨出目标地址在进行Page Table Walks的时候四个页表项 (有四级页表，每个页表根据offset来得到页表项)映射到了哪些cache set中。通过得到页表项映射的cache set编号，也就得到了物理地址的中间s位。由于PRIME+PROBE需要构造精确的LLC eviction set，在没有精确的计时器的情况下构造该set比较困难，所以在此主要介绍EVICT+TIME方法的攻击，如果想要了解PRIME+PROBE攻击，请参考[1]。

对于一个页表来说，其大小是4KB，每个页表项为8B（64位），假设cache line存储的数据大小是64B，则每一个cache line可以存储8个页表项，因此一个页表一共占了64个cache lines。所以可以将整个的LLC（第三层cache）按每页进行划分，一共可以划分为64种不同的颜色。

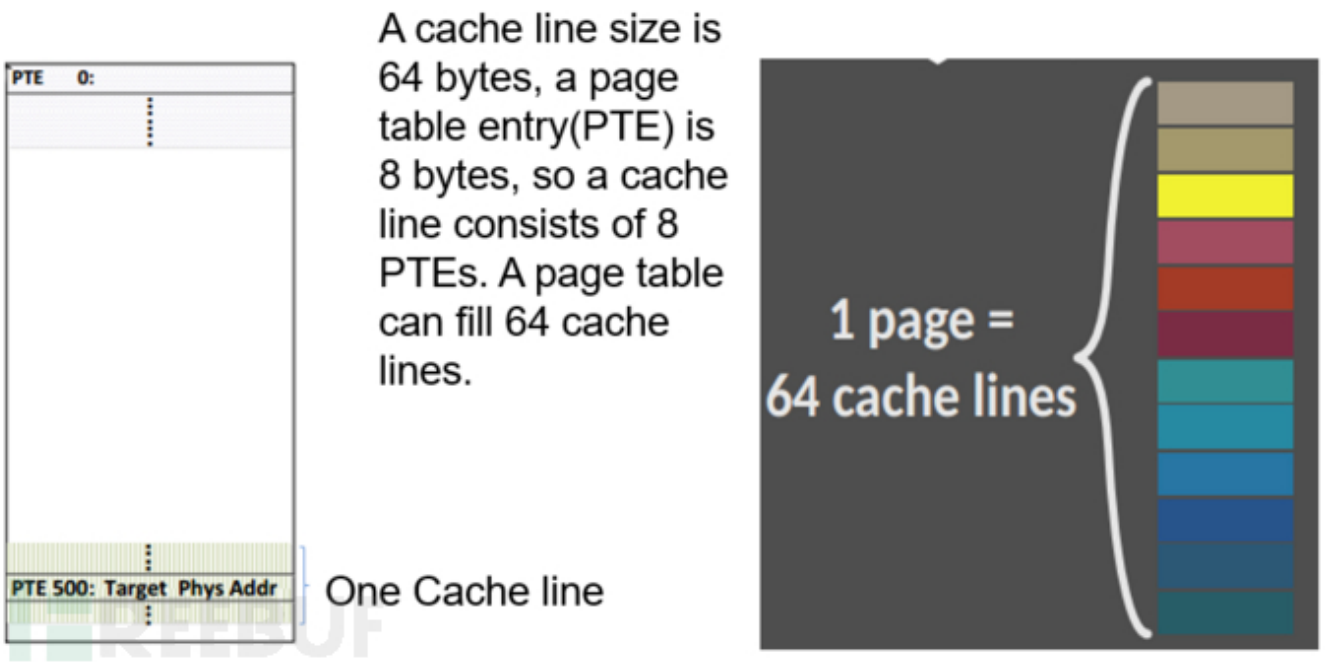

图8 page colors

具体攻击过程如下：

1): 首先选一些内存页来作为eviction set。这些eviction set必须保证能把某一个特定的page color对应的cache line都给驱逐走，保证目标页表项不在特定的page color对应的cache line中。
2): 对于目标页表项来说，如果它位于页表的第t个cache行（64个中的一个），则通过读取eviction set中那些内存中相应的offset（64个中的一个）来驱逐页表项。offset是从1到64进行遍历的。
3): 在此访问目标虚拟内存va，监控它的访问速度。如果它的访问速度比较慢，则说明t=offset，即offset对应的cache行与t对应的cache行一致。此时，就可得到一个页表项。使得offset从1遍历到64，则可得到4个页表项对应的cache offset。

Sliding PT Entries

此时已经得到了4个页表项对应的cache offset，即得到了四个页表项的物理地址的中间s位，然而现在面临着两个问题：

无法准确的知道每个页表项分别对应的cache offset，即无法建立四个页表项与得到的cache offset的一一映射关系。
无法知道cache行的页内偏移，也就无法知道物理地址的低b位。

此文[1]使用了滑动页表项这一方法有效的解决了此类问题。

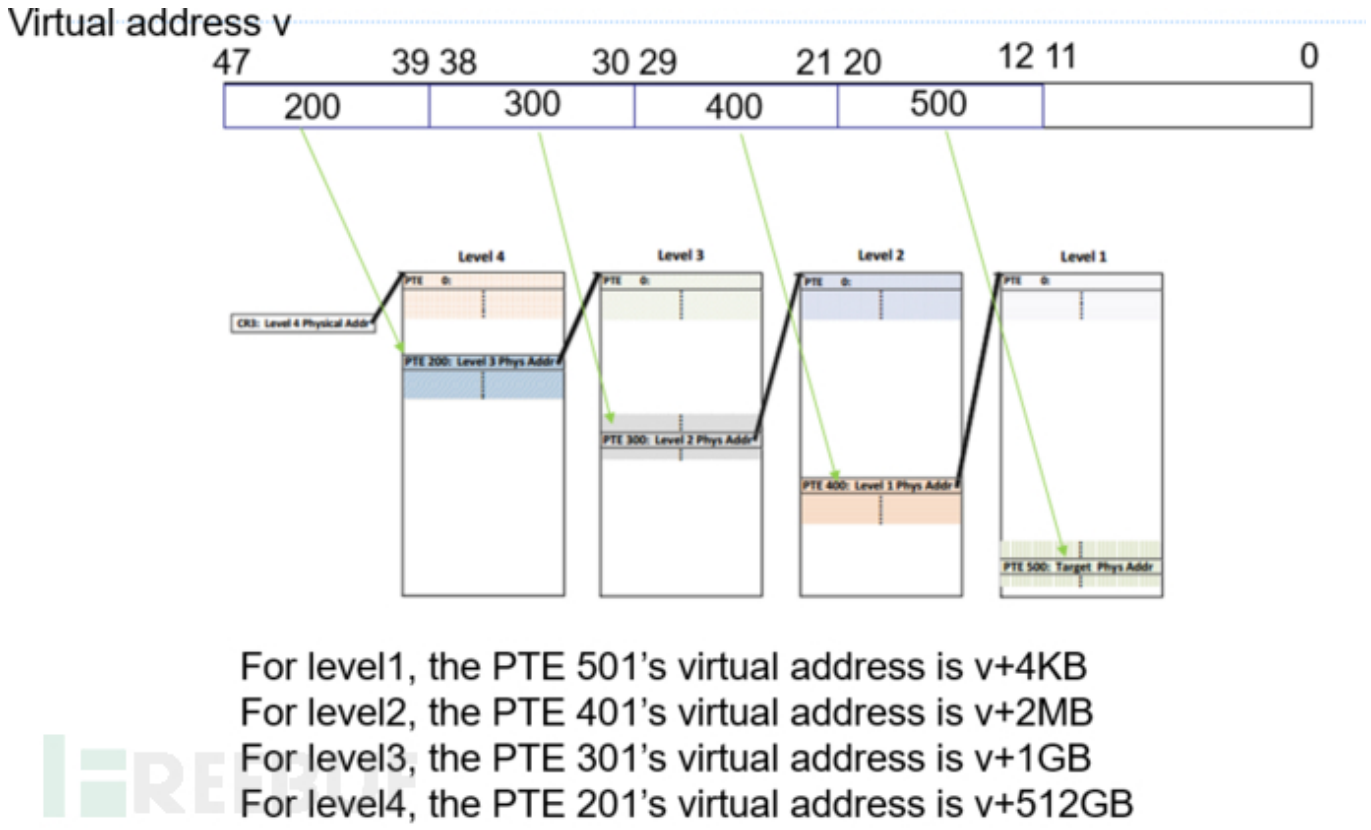

图9 sliding PT Entries

为了更清楚的解释这一方法，我们可以看一下图9这个例子。在图9中，设其此时的虚拟地址为v，我如果想要在Page Table Walks时访问Level 1的第501个页表项，则需要访问的虚拟地址为v+4KB；如果想要访问Level 2的第401个页表项，则需要访问的虚拟地址为2MB；如果想要访问Level 3的第301个页表项，则需要访问的虚拟地址为v+1GB；如果想要访问Level 4的第201个页表项，则需要访问的虚拟地址为v+512GB。

PTL1和PTL2

如果想要知道PTL1对应的cache offset，则将v+i*4KB(i=1,2,...,8)，如果检测到一个cache line在i的时候发生了变化，则可以得到两个信息：那个变化的cache line就是PTL1所在的cacahe line，因此可以知道level 1中的页表项对应的cache偏移；同时cache的块内偏移是8-i。因此也就知道了PTL1的物理地址的低(s+b)位。

同理对于PTL2来说，可以将v+i*2MB(i=1,2,...,8)，从而可以检测出PTL2对应的cache offset和cache的块内偏移。

PTL3和PTL4

由于对于PTL3和PTL4来说，相邻的页表项需要加的偏移较大（1GB和512GB），所以针对不同的浏览器内存分配策略有不同的方式进行处理。

Firefox

在Firefox中，[1]发现可以使JavaScript分配TBs级别的虚拟内存，只要对应的需要内存页不被访问(not touched)，就不会分配具体的物理内存页，因此，针对Firefox来说，可以采用反随机化PTL1和PTL2的方法来反随机化PTL3和PTL4。

Chrome

在Chrome浏览器中，当内存分配器使用mmap分配内存之后会将该内存初始化，因此分配多少虚拟内存就对应着相同的物理内存大小。因此不可能分配TBs级别的虚拟内存。[1]采取的措施是使JavaScript申请2GB的虚拟内存，一般2GB的虚拟内存对应3个Level 3的页表项（除一些特殊的情况对应2个页表项），因此在这3个页表项进行测试。如果v对应的Level 3的页表项在cache中的偏移是7或8，则能够检测出来，否则，检测不出来。

实验结果

该文[1]通过多个指标来衡量该攻击模型，在此仅从两方面进行分析：成功率和可行性。

成功率

图10显示的分别是在Chrome和Firefox下的成功率和假阴性和假阳性的比例。其中假阴性是指当在进行sliding PT entries时没有检测到cache line发生了变化，假阳性是报告出一个错误的地址（与真实的地址v不一致）。

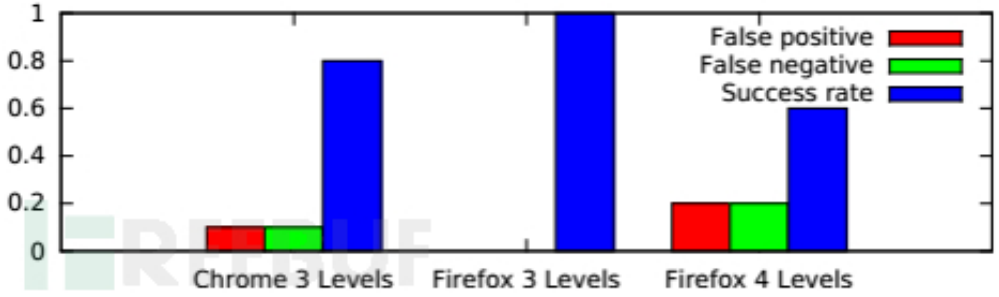

图10 Success Rate

可行性

图11显示的是随着时间的增加，剩下的还没有解出的虚拟地址的位数关系。

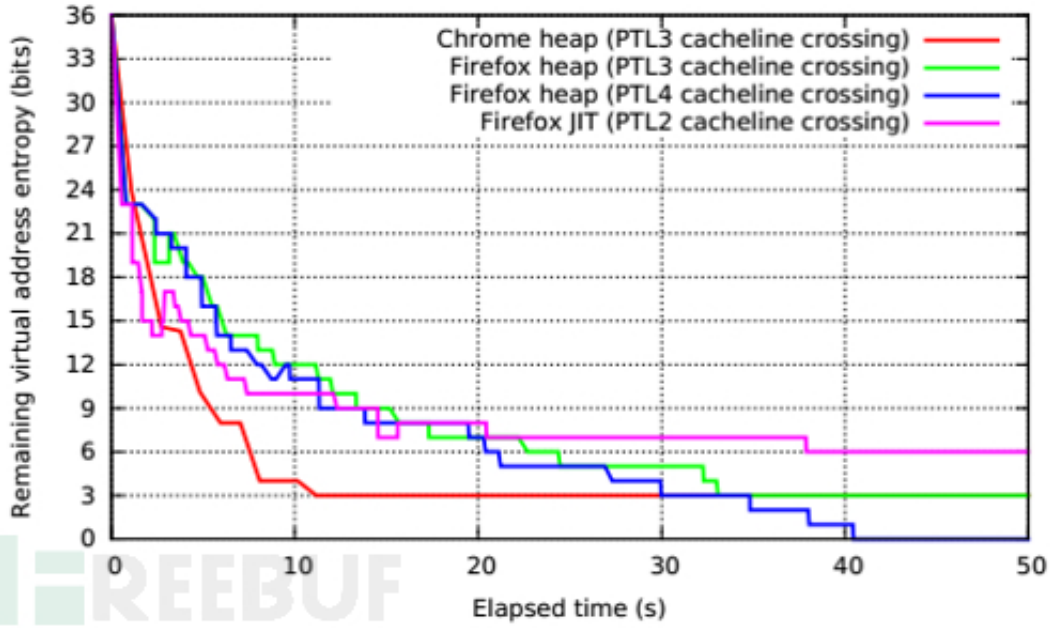

图11 Fesibility

由图11可知，其可以在较短的时间内解出大部分的虚拟地址位数。即使没有解出所有的位数，也降低了暴力破解的熵值，极大地提高了暴力破解成功的可能性。

另该项目的具体实现作者[1]已经发布在github revanc上，但是其发布的是针对C程序ASLR的反随机化。

缓解措施

最后，此文[1]也提供了一些缓解措施：

检测：由于实行该攻击需要进行大量的虚拟内存分配和cache的清空操作，在一定程度上影响了浏览器的性能，因此可以通过性能检测器来检查此类漏洞。但是此类检测被证明[1]存在假阴性和假阳性。
安全的计时器: 由于cache side channel需要准确的计时器，所以可能通过设计安全的计时器来有效的防止cache side channel。但是[6]也总结了其它有效的方式来实施cache side channel攻击
隔离的caches：可以设计一个专门的cache来存储页表项的数据。

引用

[1] Gras, B., Razavi, K., Bosman, E., Bos, H., & Giuffrida, C. (2017). ASLR on the Line: Practical Cache Attacks on the MMU. NDSS (Feb. 2017).

[2] 深入理解计算机系统(第三版 英文版)

[3] Tromer, E., Osvik, D. A., & Shamir, A. (2010). Efficient cache attacks on AES, and countermeasures. Journal of Cryptology, 23(1), 37-71.

[4] Liu, F., Yarom, Y., Ge, Q., Heiser, G., & Lee, R. B. (2015, May). Last-level cache side-channel attacks are practical. In Security and Privacy (SP), 2015 IEEE Symposium on (pp. 605-622). IEEE.

[5] Yarom, Y., & Falkner, K. (2014, August). FLUSH+ RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In USENIX Security Symposium (pp. 719-732).

[6] Cock, D., Ge, Q., Murray, T., & Heiser, G. (2014, November). The last mile: An empirical study of timing channels on sel4. In Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (pp. 570-581). ACM.