# INTRODUCTION

随着移动互联网近些年突飞猛进的发展，手机等移动设备已经成为我们生活必不可少的一部分。它在给我们生活带来便利的同时，也给我们到来了潜在的威胁，尤其是在手机能方便管理资产，存储着各种隐私的今天。因此，随着科技的进步，安全问题也越发得到人们的重视。虽然手机等设备的开发厂商以及Android系统的开发公司采取了各种措施来保护用户的隐私安全，包括可信执行环境，虚拟内存管理，权限管理等。然而，由于电脑，手机等设备结构设计的缺陷，并随着安全领域不断的研究，一些手机设备的安全漏洞和攻击方式逐渐的显露出来，其中包括cache旁路攻击。Cache旁路攻击是通过探测程序执行时cache的hit或miss，访存时间的多少等信息来进行攻击的方法。其中攻击程序与被攻击程序没有直接的交互，他们并行执行在同一个核或不同的核上，访问各自的地址空间，因此攻击时不需要多余的权限。

在最近10年中，基于x86平台的CPU cache攻击得到了人们越来越多的重视。Kocher[19]和Kelsey[18]等人提出了通过分析高速缓冲存储器在运行时泄露出的旁路信息来破解计算机中的加密算法的方法。这一思想在计算机安全相关人员的重视下得到快速的发展。在最近几年不断提出了在x86平台上的cache攻击方法，并在监测用户键盘输入，AES T-table加密破解方面得到了实验，也证明了cache攻击的有效性。比如，Page等人提出了一种针对DES加密技术的cache模型攻击的方法，将DES密钥的搜索空间从56位降低到32位。Tsunoo[34]通过分析DES查表索引以及Cache的访问特征首次实现了针对DES的Cache 攻击，并在一台600-MHz Pentium III 的个人计算机上，通过223个明文样本成功获取DES 全部密钥。Bernstein[38]在强制消除网络传输时延条件下实现了一种针对OpenSSL[27]中AES 的远程时序驱动计时攻击，其中加密服务端采集AES 加密时间。Bonneau[5]提出了一种利用密码程序在加密中的内部数据访问冲突导致的Cache 命中和失效信息来进行密码分析的方法。Percival[30]提出多线程间共享Cache 存储器访问引发的漏洞，它不仅提供了线程间一个简单、高带宽的隐秘通道，也给恶意线程监视其他线程提供了入口，使得恶意线程能够窃取加密密钥，他依次设计实现了一种针对RSA 的计时攻击。Osvik[29]借鉴Percival信息采集方法实现了多例针对AES 的Cache 计时攻击。Neve[24][25][26]将Osvik攻击[33]切入点转移到最后一轮，提出了一种新的AES 最后一轮访问驱动的Cache计时攻击方法。可见，通过这些漏洞，攻击者不仅可以获得用户键盘输入等的隐秘信息，还能获得价值更高的密钥等信息，进而威胁到用户的财产安全。

然而，由于Android等移动设备的体系结构与Intel x86结构有很大的区别，在指令集合cache组织方式以及cache替换策略等与cache攻击相关结构也有很大的不同。因此，直到最近为止，才有人提出在非root的手机上的有效的跨核 cache攻击方法。Moritz Lipp et al. [21] 提出了通过prime + probe， flush + reload， evict + reload以及flush + flush对ARM处理器的跨核攻击模型，并且不需要root权限。这些模型能够有效的探测到在被攻击程序运行时cache无意间泄露出来的信息，通过对这些信息进行统计分析，并将其作用于cache攻击的模型即可提取用户的私密信息。然而当前并没有在android端完整实现AES攻击的例子。由于基于Android系统的移动设备和x86设备的cache结构不同，在将x86平台的攻击方式应用到移动设备时会出现一些问题。比如cache的替换策略，在x86平台上，因为cache使用的是LRU替换策略，可以很简单的将cache中指定set中的数据驱逐到内存中。然而由于android使用的是伪随机替换策略，在将cache中指定set中的数据驱逐到内存时需要使用额外的方法。此外，为了得到稳定的数据访问时间，通常需要对访存或访cache操作进行预热。而之前的攻击方式通常直接使用第一次访问cache或访问内存的时间来度量cache命中与否，从而容易引进误差。

# Background

## Determining the Best Eviction Strategy

为了将地址从cache中驱逐到主存中，在Intel x86平台可以使用非特权的clflush指令。虽然ARM平台也提供了类似的cache操作工具，但在非特权模式下不允许使用。驱逐的第二种方式为相关地址的访问，主要原理为读取大量的能够映射到制定cache set的地址，以此来将该cache set中之前存储的数据替换到主存中。虽然读取大量的地址能够大概率的保证将关联set中的数据都驱逐出内存，但大量的访存操作不仅仅会增加驱逐所花的时间，存储相关地址的内存也会增大，而且由于cache伪随机替换策略的影响，驱逐干净cache set后很难了解cache set中存储哪些相关地址，会对probe阶段的探测结果产生影响，从而影响攻击结果。

除此之外，还需将L1 cache中相关set中的数据也驱逐到内存中。因此找到快速且可靠的驱逐方式是至关重要的。驱逐是否成功可以通过探测待驱逐的地址是否仍在cache中来判断。

Gruss et al. 发现了有三个因素对驱逐的成功率有影响，并将其作为可调整的驱逐策略参数：

1. 只有在能够映射到同一个cache set中的地址的cache hits和cache misses会对驱逐的成功率有非负的影响。这可以通过在相关地址中添加能够映射到其他cache set的地址来验证，并可以发现随机的非关联地址不会对平均的成功率产生影响。因此驱逐策略的有效性依赖于驱逐set的大小。
2. 此外，对于cache来说地址是不可区分的，因此访问模式被定义为一个地址系列，比如，其中每个标号代表一个不同的地址，这个系列定义了地址访问的一个先后次序，先访问等等。如果这个模式定义在一个循环中，则每个循环中访问的不同地址数会对驱逐策略的有效性产生影响。
3. ARM平台的cache替换策略倾向于驱逐最近添加到cache line中的数据，因此需要重复的访问相同的地址来保证地址被保存在cache中。比如，将驱逐序列从到缩短了超过33%的执行时间，并且增加了驱逐率。此外，在一定次数的重复之外，再增加访问次数不会增加驱逐率，或许还会更差。

基于这些观察，Gruss et al. 定义了依赖于cache以及cache的替换策略的三个可调整参数的驱逐模式，以供不同设备的调整以得到最佳的驱逐策略。

|  |  |
| --- | --- |
| 驱逐算法 | |
| 输入：  N：待驱逐set中可以存放的不同地址数  D：每个循环访问的不同地址数  A：每个循环每个地址的访问次数 | |
| 输出：无 | |
| 1:  2:  3:  4:  5:  6: | for i = 0; i < N – D; i++ do  for j = 0; j < A; j++ do  for k = 0; k < D; k++ do  Access(i + k)访问第i + k个相关地址  end for  end for  end for |

因此，为了保证驱逐的成功率，需要对具体的设备做大量的实验以得到快速有效的驱逐模式。再此上才能保证行而有效的cache攻击。

## The Prime Probe Strategy

一个精确的计时方式是cache攻击的前提，它需要将cache hits和cache misses区分开来，供攻击者获取待攻击程序运行导致的cache状态的变化。并针对不同的攻击对象获取不同的相关信息，比如对共享库的攻击需要获取共享库相关地址在被攻击程序运行期间的cache缓存状况。然而获取这些状况的前提都是拥有能够准确区分cache hits和cache misses的能力。Moritz Lipp et al.[24]虽然提出了几种非特权的计时方式，包括perf\_event\_open、POSIX的clock\_gettime函数以及专用的线程计时器。但这些接口并非对所有Android版本，对所有处理器都开放，因此需要对待攻击机型确定能精确且能稳定测量的时间源，以此提供对cache攻击的支持。

除了通过读取寄存器获取CPU周期来度量时间外，还有其他3种可选方式来度量访存或访问cache的时间。

* clock\_gettime系统调用

"clock\_gettime"是基于Linux C语言的时间函数，他可以精确到纳秒。

#include<time.h>

int clock\_gettime(clockid\_t clk\_id， struct timespec \*tp);

clk\_id : 检索和设置的clk\_id指定的时钟时间。

CLOCK\_REALTIME:系统实时时间，随系统实时时间改变而改变，即从UTC1970-1-1 0:0:0开始计时，中间时刻如果系统时间被用户改成其他，则对应的时间相应改变

CLOCK\_MONOTONIC:从系统启动这一刻起开始计时，不受系统时间被用户改变的影响

CLOCK\_PROCESS\_CPUTIME\_ID:本进程到当前代码系统CPU花费的时间

CLOCK\_THREAD\_CPUTIME\_ID:本线程到当前代码系统CPU花费的时间

struct timespec

{

time\_t tv\_sec; /\* 秒\*/

long tv\_nsec; /\* 纳秒\*/

};

* Perf性能分析工具

Perf是内置于Linux内核源码树中的性能剖析(profiling)工具。它基于事件采样原理，以性能事件为基础，支持针对处理器相关性能指标与操作系统相关性能指标的性能剖析。常用于性能瓶颈的查找与热点代码的定位。

从Linux Kernel2.6.31版本开始，Linux内核开始提供\_\_NR\_perf\_event\_open的系统调用。使用这个系统调用我们可以像使用文件一样打开一个Performance counter，通过设置不同的参数让这个Performance Counter统计不同的软件或硬件事件，然后就可以向读文件一样来读取这些事件的统计结果。

static struct perf\_event\_attr attr;

attr.type = PERF\_TYPE\_HARDWARE;

attr.config = PERF\_COUNT\_HW\_CPU\_CYCLES;

fd = syscall(\_\_NR\_perf\_event\_open， &attr， 0， -1， -1， 0);

PERF\_COUNT\_HW\_CPU\_CYCLES：统计CPU周期数

* 线程计时模拟器

如果没有足够精确的计时接口可用，攻击者可以通过运行一个循环自增一个全局变量的线程来得到CPU周期的一个倍数估计值。因为一次循环自增操作可看做是由固定次数个时间周期组成的，所测时间的整数倍即为所对应的时钟周期，因此能够将cache hits和cache misses区分开来，从而也可以用于cache攻击。