**北京航空航天大学计算机学院**

**硕士学位论文文献综述**

**论文题目：**面向移动设备的cache攻击关键技术研究

**专 业：**软件工程

**研究方向：**软件工程

**研 究 生：**李勃

**学 号：**SY1506402

**指导教师：**姜博副教授

**北京航空航天大学计算机学院**

2016年12月10日

面向移动设备的cache攻击关键技术研究

**摘 要**

近年来，提出了一种新的密码分析或信息截取的一种分析方法，这种方法将cache的行为特征作为获取密码旁路信息，通过获取待攻击设备在执行加密算法过程中cache泄露出来的时间、cache命中率、页表加载信息等与加密内容不显式相关的信息来获得cache命中或cache缺失的情况，并以此结合加密算法的数学特征来推断出密钥的秘密信息，或是减少推断密钥的复杂度甚至可以得到部分或所有密钥。由于cache攻击强大的攻击能力，能在非特权模式下对重要的密钥等信息进行有效的攻击，在x86平台下得到了越来越多的关注。而ARM架构在指令集、物理结构、cache组织等方面与x86有较大的差异，一直以来没有有效的跨核cache攻击方式，直到2016年Moritz Lipp提出了在非特权模式下ARM体系的跨核攻击方法。通过该方法实现的包括Prime + Probe、Evict + Reload等攻击模式能够有效的获取cache的旁路信息，并以此实现包括Android手机的cache攻击工具。

本文主要分析了面向移动设备的cache攻击的关键问题，总结了近年来cache攻击的研究成果，指出了cache移动设备攻击时面临的挑战。

**关键词：** 高级加密标准 访问驱动 Cache计时攻击 远程攻击 OpenSSL

**Abstract**

In recent years, a new analysis method of cryptanalysis or information interception has been proposed. This method takes the behavioral characteristics of cache as the information of bypassing the cryptosystem, and obtains the information that the attacking device leaks in the process of executing the encryption algorithm. Time, cache hit rate, page table loading information and the encrypted content is not explicitly related to the information to get the cache hit or cache miss situation, and in combination with the mathematical features of the encryption algorithm to infer the secret key information, or Reducing the complexity of deducing keys can even get some or all of the keys. Due to the powerful attack ability of cache attack, it can effectively attack important information such as key in non-privileged mode, and more and more attention has been paid on x86 platform. The ARM architecture in the instruction set, physical structure, cache and other aspects of the organization has a large difference with the x86, there has been no effective cross-core cache attack until 2016 Moritz Lipp proposed in the non-privileged mode ARM system cross- Attack method. By this method, including Prime + Probe, Evict + Reload and other attack patterns can effectively access the cache bypass information, and in order to achieve, including Android mobile phone cache attack tool.

This paper mainly analyzes the key issues of cache attack for mobile devices, summarizes the research results of cache attacks in recent years, and points out the challenges of cache mobile devices attack.

**Keywords:** AES, access driven, Cache timing attack, remote attack, OpenSSL

目 录

[1 Cache结构概述 1](#_Toc469874315)

[1.1 Cache简介 1](#_Toc469874316)

[1.2 Cache内存映射方式-全相联映射 2](#_Toc469874317)

[1.3 Cache内存映射方式-直接相联映射 3](#_Toc469874318)

[1.4 Cache内存映射方式-组相联映射 5](#_Toc469874319)

[1.5 Cache替换策略 7](#_Toc469874320)

[2 Cache攻击概述 8](#_Toc469874321)

[3 ARM cache攻击技术 11](#_Toc469874322)

[3.1 ARM cache组织 11](#_Toc469874323)

[3.2 Cache驱逐 12](#_Toc469874324)

[3.3 Cache驱逐策略 12](#_Toc469874325)

[3.4 计时方式 14](#_Toc469874326)

[4 结论 16](#_Toc469874327)

[5 主要参考文献 18](#_Toc469874328)

# Cache结构概述

## Cache简介

如今cpu的性能不仅仅依赖于时钟频率，Cache的出现是为了解决cpu日益增长的时钟频率与主存相对落后的读取时间的矛盾。他利用了程序运行过程中的局部性原理，也就是在前一段时间中使用的代码及数据在接下来的时间中被再次使用的几率也会很高[47]。将经常使用的数据存储在介于主存与cpu之间的速度相较主存快速的cache中，并在cache中做相关标记表示相关的地址被缓存到cache中，在cpu再次使用到这些数据时就直接从cache中获取。由于通常从cache中获取数据比从内存中读取数据块很多，因此能在限制成本的情况下显著提高系统的性能。

由于高速存储器非常昂贵，因此系统存储结构被组织为多级架构，越接近cpu的存储单元容量越小，价格越高，存储速度越快。访问cache的速度比访问其他主存速度要快得多。

由于性能方面的考虑，第一级cache通常与cpu物理相连来提高cpu访问cache的速度。而系统大致可分为两种架构，一种是哈佛架构，他是一种将程序指令和数据存储分隔开的存储器架构。他的特点是将指令和数据存储在不同的存储空间中，这些存储空间独立编址，独立访问。与两个存储器相对应的是系统的4条总线：程序和数据的数据总线与地址总线。这种分离的程序总线和数据总线可允许在一个机器周期内同时获得指令字（来自程序存储器）和操作数（来自数据存储器），从而提高了执行速度，提高了数据的吞吐率。又由于程序和数据存储在两个分开的物理空间中，因此取地址和执行能完全重叠。中央处理器首先到程序指令存储器中读取程序指令内容，解码后得到数据地址，再到相应的数据存储器中读取数据，并进行下一步的操作（通常是执行）。程序指令存储和数据存储分开，可以使指令和数据有不同的数据宽度。另一种为非哈佛体系，通常也称为冯诺依曼体系结构。该结构的最大特点就是不加区分的将程序和数据存储到一个存储其中，统一编址，通过同一条总线来获取指令和数据。哈佛结构与冯·诺依曼结构处理器相比，处理器有两个明显的特点：使用两个独立的存储器模块，分别存储指令和数据，每个存储模块都不允许指令和数据并存；使用独立的两条总线，分别作为CPU与每个存储器之间的专用通信路径，而这两条总线之间毫无关联。

作为数据的缓存位置，cache的最小操作单位是一个cache line，每次从主存中获取一个line大小的数据缓存到cache中。也就是说不能单从内存中获取单个字节。当CPU要从缓存中获取数据时，首先需要比对tag，在确认所需访问的数据在cache中之后通过index找到数据并送至CPU。如果在cache中未能找到待访问的数据，则到内存中找到待访问的数据后将其送至CPU并在cache中缓存下相应区域的数据，以方便下次访问，从而通过程序的时间和空间局部性原理来提高系统的性能。

## Cache内存映射方式-全相联映射

地址映射规则：主存的任意一块可以映射到Cache中的任意一块[48]

(1) 主存与缓存分成相同大小的数据块。

(2) 主存的某一数据块可以装入缓存的任意一块空间中。

全相联方式的对应关系如图1所示。如果Cache的块数为，主存的块数为，则映射关系共有×种。

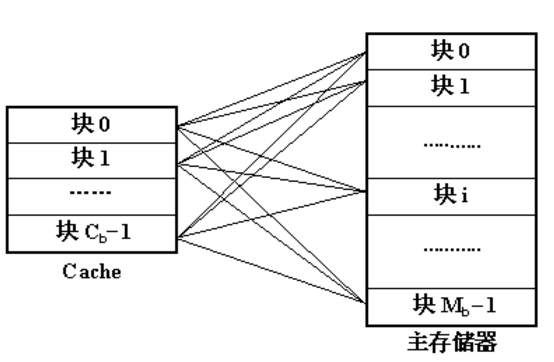


图1 全相联映射方式

图2展示出了目录表的格式及地址变换规则。 目录表存放在相关（联）存储器中，其中包括三部分：数据块在主存的块地址、存入缓存后的块地址、及有效位（也称装入位）。由于是全相联方式，因此，目录表的容量应当与缓存的块数相同。

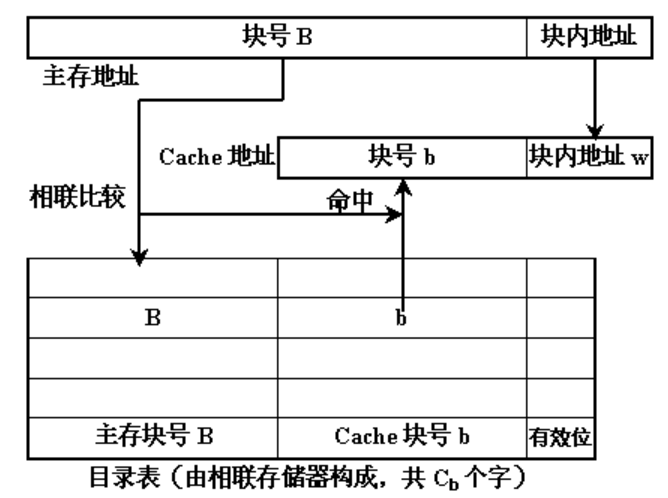


图2 全相联地址转换

举例：某机主存容量为1M，Cache的容量为32KB， 每块的大小为16个字（或字节）。主、缓存的地址格式、 目录表格式及其容量如图3所示。

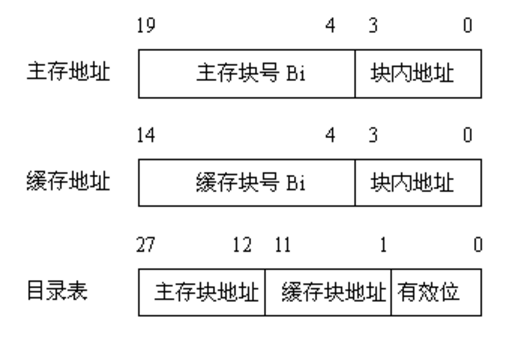


图3 地址结构

可得容量与缓冲块数量相同即＝2048（或32K/16＝2048）。全相联映射的主要优点为命中率比较高，Cache存储空间利用率高。但当访问相关存储器时，每次都要与全部内容比较，速度低，成本高，因而应用少。

## Cache内存映射方式-直接相联映射

直接相联映射的地址映射规则为：主存储器中一块只能映射到Cache的一个特定的块中[48]。主存与缓存分成相同大小的数据块，主存容量应是缓存容量的整数倍，将主存空间按缓存的容量分成区，主存中每一区的块数与缓存的总块数相等。并且主存中某区的一块存入缓存时只能存入缓存中块号相同的位置。

图4展示出了直接相联映射规则。 可见，主存中各区内相同块号的数据块都可以分别调入缓存中块号相同的地址中，但同时只能有一个区的块存入缓存。由于主、缓存块号相同，因此，目录登记时，只记录调入块的区号即可。

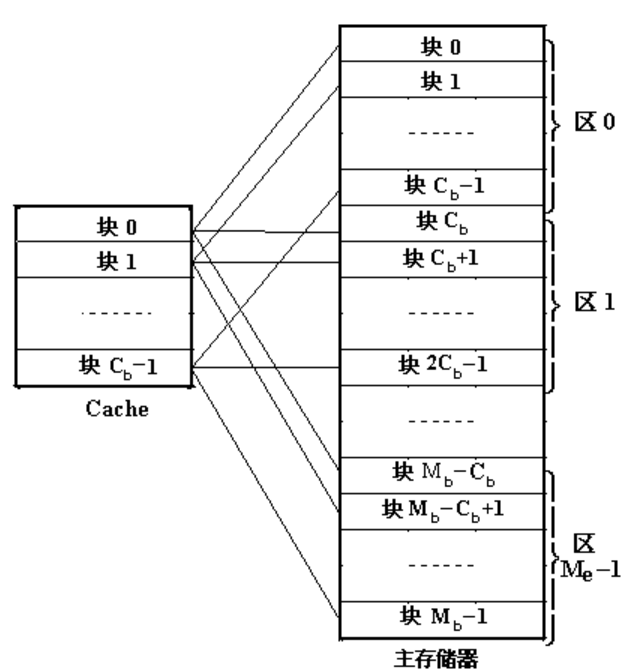


图4 直接相联映射方式

图5展示出了主、 缓冲地址格式、目录表的格式及地址变换规则。主、缓存块号及块内地址两个字段完全相同。目录表存放在高速小容量存储器中，其中包括二部分：数据块在主存的区号和有效位。目录表的容量与缓存的块数相同。

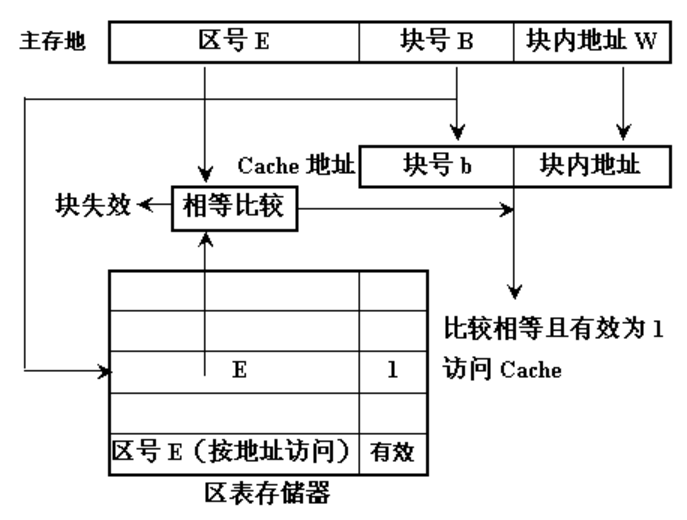


图5 直接相联地址转换

直接相联的地址变换过程：用主存地址中的块号B去访问目录存储器， 把读出来的区号与主存地址中的区号E进行比较， 比较结果相等，有效位为1，则Cache命中，可以直接用块号及块内地址组成的缓冲地址到缓存中取数；比较结果不相等，有效位为1， 可以进行替换，如果有效位为0，可以直接调入所需块。

直接相联映射的优点为:地址映象方式简单，数据访问时，只需检查区号是否相等即可，因而可以得到比较快的访问速度，硬件设备简单。但也有替换操作频繁，命中率比较低等缺点。

## Cache内存映射方式-组相联映射

另一种cache内存映射方式为组相联映射，他是直接相联以及全相联映射的折中方案[48]。组相联的映射规则为主存和Cache按同样大小划分成块，主存和Cache按同样大小划分成组。主存容量是缓存容量的整数倍，将主存空间按缓冲区的大小分成区，主存中每一区的组数与缓存的组数相同。当主存的数据调入缓存时，主存与缓存的组号应相等，也就是各区中的某一块只能存入缓存的同组号的空间内，但组内各块地址之间则可以任意存放， 即从主存的组到Cache的组之间采用直接映射方式；在两个对应的组内部采用全相联映射方式。

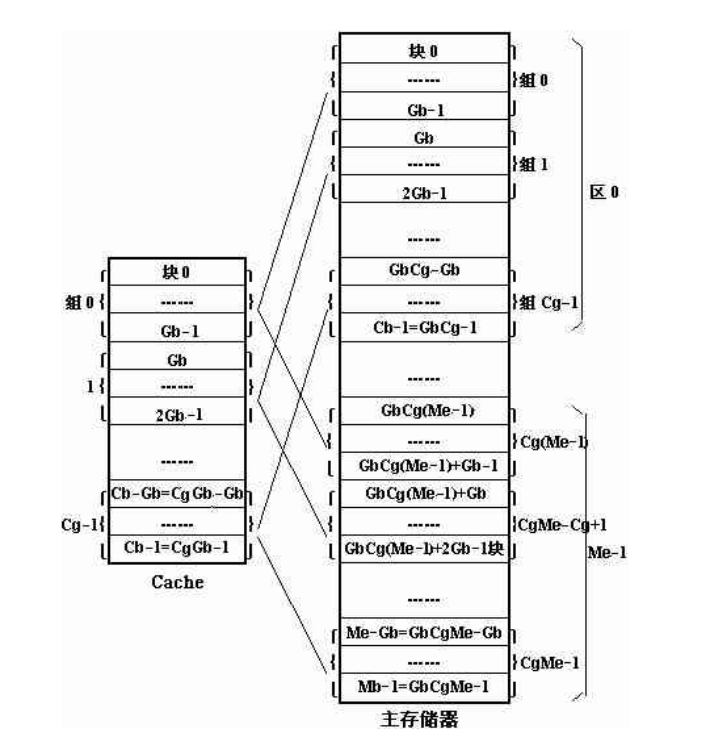


图6 组相联地址映射

图6展示出了组相联的映象关系， 图中缓存共分Cg个组，每组包含有Gb块；主存是缓存的Me倍，所以共分有Me个区， 每个区有Cg组，每组有Gb块。那么，主存地址格式中应包含4个字段：区号、区内组号、组内块号和块内地址。 而缓存中包含3个字段：组号、组内块号、块内地址。主存地址与缓存地址的转换有两部分，组地址是按直接映射方式，按地址进行访问，而块地址是采用全相联方式，按内容访问。组相联的地址转换部件也是采用相关存储器实现，如图7所示。

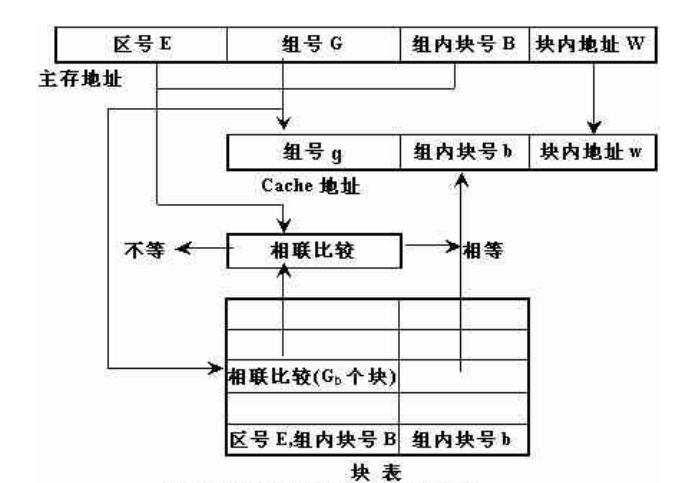


图7 组相联映射的地址转换

相关存储器中每个单元包含有： 主存地址中的区号E与组内块号B，两者结合在一起，其对应的字段是缓存块地址b。相关存储器的容量，应与缓存的块数相同。当进行数据访问时，先根据组号，在目录表中找到该组所包含的各块的目录，然后将被访数据的主存区号与组内块号，与本组内各块的目录同时进行比较。如果比较相等，而且有效位为“1”则命中。

将其对应的缓存块地址b送到缓存地址寄存器的块地址字段，与组号及块内地址组装即形成缓存地址。如果比较不相等，说明没命中，所访问的数据块尚没有进入缓存，则进行组内替换；如果有效位为0，则说明缓存的该块尚未利用，或是原来数据作废，可重新调入新块。

组相联映射地址转换能将块的冲突概率降低，块的利用率大幅度提高，块失效率明显降低，但实现难度和造价要比直接映射方式高。

## Cache替换策略

根据程序局部性规律可知：程序在运行中，总是频繁地使用那些最近被使用过的指令和数据[46]。这就提供了替换策略的理论依据。综合命中率、实现的难易及速度的快慢各种因素，替换策略可有随机法、先进先出法、最近最少使用法等。

随机法是随机地确定替换的存储块。设置一个随机数产生器，依据所产生的随机数，确定替换块。这种方法简单、易于实现，但命中率比较低。

先进先出法是选择那个最先调入的那个块进行替换。当最先调入并被多次命中的块，很可能被优先替换，因而不符合局部性规律。这种方法的命中率比随机法好些，但还不满足要求。先进先出方法易于实现，例如Solar－16/65机Cache采用组相联方式，每组4块，每块都设定一个两位的计数器，当某块被装 入或被替换时该块的计数器清为0，而同组的其它各块的计数器均加1，当需要替换时就选择计数值最大的块被替换掉。

LRU法是依据各块使用的情况，总是选择那个最近最少使用的块被替换。这种方法比较好地反映了程序局部性规律。

实现LRU策略的方法有多种。 下面简单介绍计数器法、寄存器栈法及硬件逻辑比较对法的设计思路。计数器方法：缓存的每一块都设置一个计数器，计数器的操作规则是：

(1) 被调入或者被替换的块， 其计数器清“0”，而其它的计数器则加“1”。

(2) 当访问命中时，所有块的计数值与命中块的计数值要进行比较，如果计数值小于命中块的计数值， 则该块的计数值加“1”；如果块的计数值大于命中块的计数值，则数值不变。最后将命中块的计数器清为0。

(3) 需要替换时，则选择计数值最大的块被替换。

# Cache攻击概述

如今的CPU性能不仅仅受时钟频率的影响，除此之外也收到指令、操作码获取以及与设备交互的因素的影响。为了能够提高系统的访存操作的速度，CPU通过cache缓存了经常使用的数据，其中cache是个小而快的内部存储单元。

当代的cache大多数实现组相联的映射方式，其中一个cache set分为多个cache line。每个内存地址映射到其中一个cache set中的一个cache line中。并将映射到相同cache set中的地址称为是相关地址。相关地址相互竞争着相同cache set中的cache line，并通过一个事先定义好的替换策略来实现cache line的替换。例如，通常使用的Intel CPUs使用一种称之为最近最少使用的替换算法（LRU）[1]。然而通常移动设备使用的是ARM处理器，它的第一级cache（L1）一般使用伪随机替换策略，第二级cache通常支持两种替换策略，分别为循环替换策略和伪随机替换策略。然而在实际使用中由于性能的原因只使用伪随机替换，要改变L2 cache的替换策略只能通过特权模式进行修改。

CPUs有多级cache，较低级的cache比高级的cache快。ARM处理器通常有两级cache，如果较低级cache中line所存储的数据也存储到高级的cache中，则称cache为inclusive。如果一个cache line仅仅只存在于某一级别的cache中，则称该cache为exclusive，而如果一个cache既不是inclusive的也不是exclusive的，则称该cache为non-inclusive。通常最高级的cache是核间共享的，以促进核间共享以及数据的传输。然而，通过利用这一特性，一个核能够影响其他核存放在cache中的内容。这也促成了最基本的核间cache攻击方式，Flush+Reload[2]。

为了保持多个CPUs在同一的状态，通常使用一致性协议来确保主存及cache内容的一致性。然而，一致性协议也能造成可利用的时间信息来进行攻击，例如Irazoqui et al. [3] on x86 CPUs。

只读共享内存被用来作为内存使用优化的一种方式，其减少共享库在内存中的保存数目，以通过减少cache中的内容来提升速度。操作系统将相同的物理内存映射到每个线程中。因为这种内存共享机制与文件打开方式、存储方式以及访问方式无关，因此，攻击者能够将这段只读的共享内存映射到攻击程序的地址空间中。

Android设备通常是用Java语言编写的。因此其包含了优化改变的代码和实时编译的代码，这些代码是非共享的。然而从Android 4.4版本开始，Dalvik VM逐渐被Android Runtime（ART）所取代。通过ART，Java字节码被编译为本地二进制代码，因此变为可共享的。

起初cache时间攻击针对的是加密算法[4][5][6][7][8][9][10]。例如，Bernstein[4]利用AES T-table实现的执行总时间来进行攻击。Percival[9]提出了能够更好的利用内存访问和cache访问时间差别的方法。此外，Osvik et al. 提出了Evict + Time以及Prime + Probe的概念来判断一个指定的cache set是否被待攻击程序占用。这两种方法由以下三个基本步骤组成：

Evict + Time：

1. 度量被攻击程序的执行时间
2. 驱逐一个指定的cache sets
3. 再次度量被攻击程序的执行时间

Prime + Probe：

1. 占用指定的cache sets
2. 被攻击程序执行
3. 判断哪些cache sets依然被占用

这两种方式均能让攻击者判断在待攻击程序运行期间哪些cache sets被占用了，并通过这些信息来攻击加密的实现[11,12,13,14]，还可以构建跨虚拟机的传输信道[15]。Yarom和Falkner[2]提出了Flush + Reload的方法，该方法利用了现代系统结构的三个基础概念来进行攻击。第一，攻击程序和被攻击程序之间共享内存的可利用性；第二，最后级cache在多个核之间是共享的；第三，Intel平台使用的是inclusive的最后一级cache，意味着从最后一级cache中驱逐的数据也会从其他核的低级cache中驱逐。因此允许任何程序将在其他核上的其他线程或进程的数据驱逐出cache。尽管这种攻击方式的基础是由Gullasch et al.[17]提出的，Yarom和Falkner将其扩展到共享的最后一级cache，以支持跨核攻击。Flush + Reload的操作过程如下：

Flush + Reload：

1. 将共享库映射到地址空间
2. 从cache中刷新一个cache line
3. 执行被攻击程序
4. 检查第二步相关的cache line是否被被攻击程序重新加载

因此，Flush + Reload允许攻击者判断具体哪条指令被执行或者哪些数据被被攻击程序访问。这种攻击方式能够获得更多更有用的信息，并已经被实现以攻击加密的实现[18][19][20]。此外，Gruss et al.[21]阐述了基于Flush + Reload的利用单通道cache信息进行自动化攻击的可能性。除了攻击AES T-table类似的加密实现外，他们展现了如何推测键盘敲击的信息甚至通过利用cache边信道来构建按键敲击的日志信息。与之类似，Oren et al.[22]提出了利用cache攻击在Intel平台上攻击JavaScript实现可能性，并展示了如何推测访问的网站信息以及如何跟踪用户的鼠标活动。

Gruss et al.[21] 提出了替换Flush + Reload中的flush操作的Evict + Reload技术。他将flush操作替换为等价的evict操作。尽管这种方式没有在x86 CPUs中实现，Moritz Lipp et al.[24]展示了其在ARM CPUs上的实现。最近，Flush + Flush[25]方法被提出，不像其他技术，该方式并不依赖访存的信息，而是依赖刷新指令的时间来判断一个cache line是否被被攻击程序加载。Moritz Lipp et al.展示了在ARMv8-A刷新指令的执行时间依赖于数据是否在缓存中，因此，这种方式也可以用于实现cache攻击。

因为上面所描述的攻击方式都是基于Intel处理器的，因此类似的攻击被认为不能应用到手机等移动设备上，因为这些设备通常有不同的指令集，cache的组织方式也不一样[2]。因此，之前只有关于手机的同核攻击方面的阐述。例如，WeiB et al.[27] 调查了Bernstein cache时间攻击[4]，在一个使用ARM Cortex-A8的微型处理器上。随后WeiB et al.[29]在多核情况下调查了这种时间攻击。就像WeiB et al.[27]所阐述的那样，在ARM处理器上，系统噪音会使得攻击困难。这一点Spreitzer和Plos[31]通过在不同的ARM Cortex-A8和ARM Cortex-A9手机上进行cache时间攻击也得到证明。他们的调查实验都验证了时间信息被泄露，但是需要花费大量的时间和大量的测量来进行cache攻击。

除了Bernstein攻击外，Bogdanov et al[32].提出了另一个针对AES T-table实现的cache攻击，其利用ARM9处理器上的被称之为宽冲突的特性。此外，有效的分析攻击[33]和电磁发散[34]被用来在ARM处理器上将cache访问可视化。Spreitzer和Plos[35]实现了在Android手机上攻击AES T-table实现的Evict + Time[13]方法。然而到目前为止，仅仅只有针对AES T-table实现的cache攻击。

# ARM cache攻击技术

## ARM cache组织

与Intel CPUs相比，ARM CPUs显得要多样化的多。例如，ARM处理器的第二级cache是否存在是由制造商决定的。此外，ARM设备上的最后一级cache通常是多核共享的，他可以拥有针对指令和数据不同的包含结构。由于cache的一致性原则，共享内存在多核CPUs上需要保持状态的一致性。这是度量访存和访cache时间差别的关键，因为远程核的cache访问比访存时间更快[37]。对于不遵循一致性原则的cache，跨核攻击也就不可行，但攻击者可以在所有核上运行攻击工具，因此等价于在相同核上的cache攻击。然而，大多数移动设备的cache都是满足cache一致性原则的。

为了执行跨核攻击，需要将足够多的书籍加载到cache中来驱逐相关的最后一级cache set。因此，通过利用将最后一级cache直接或间接的填充来实现驱逐。对于最后一级cache是inclusive包含式的情况，可以通过将本核的数据驱逐来实现其他核上相同数据的驱逐。图3.1.1展示了一个inclusive指令cache的驱逐过程：首先，一条指令被分配到最后一级cache，随后加载到某个CPU的第一级指令cache中；随后，一个处理器填充他的核的数据cache，导致最后一级cache set中line的驱逐；处理器通过填充数来来将其他核的指令驱逐出cache。

为了实现数据和指令在L1 cache和L2 cache之间的转换需要多次访问cache lines。因此越来越多的用于驱逐cache set的地址被缓存到L1或L2中。由于ARM CPUs的L1 cache的关联性很低，从L2 cache中驱逐出数据的可能性很高。因此执行在L1和L2之间频繁转换的驱逐策略能够进一步增加驱逐的概率。这种方法也支持其他组织方式的cache的跨核或跨CPU的攻击。由于CPU核之间的cache一致性协议[37][39],从远程核上取数据要比从内存中直接取数据速度要快，因此能够通过访问时间有效的区分开cache hit和cache miss的情况。

Moritz Lipp et al.[24]展示了在OnePlus One手机上跨核远程访问与访存时间区分的实验，如图3.1.2所示，跨核远程访问与本地核cache访问之间平均相差40个CPU周期，然而cache miss需要平均花费500个CPU周期。因此可以通过一个固定的时间阈值将cache hit和cache miss区分开来。

## Cache驱逐

Cache攻击的策略之一为获取被攻击程序运行时的cache状态信息，包括程序运行时间、cache set占用情况以及数据访问时间等，进而分析出对解密有帮助的信息，降低解密复杂度甚至得出部分或全部密钥。之前提到的各种攻击方式所用到的驱逐策略均需要将特定的cache set从cache中驱逐出去，而驱逐方式大致有两种：flush指令或使用内存中能映射到待驱逐set的相关地址。但由于flush指令并非在所有移动设备中都是可用的，大部分设备需要通过相关地址来实现对指定set的驱逐。Cortex-A53和Cortex-A57的L1 cache只有少数几路，并且使用最近最少使用替换策略[40]，对一次完整的驱逐来说，还需要将使用伪随机替换算法的L2 cache中的cache line驱逐出去。

## Cache驱逐策略

之前在Intel x86平台上的驱逐数据的方法需要大量的时间消耗[41]，并且只能应用于使用LRU替换策略的cache实现[12][15][22]。Spreitzer和Plos[35]提出了针对ARMv7-A CPU的驱逐策略，由于伪随机替换策略，需要对每个cache set访问更多的地址来实现cache set的驱逐。最近，Gruss et al.[25]阐述了如何在Intel x86平台上自动找出快速的驱逐策略的方法。Moritz Lipp et al.[24]将其应用到ARM CPUs上。通过这种方式可以自动的获取驱逐策略来减少系统消耗[35]。

因此，尽管cache使用伪随机替换策略，平均的驱逐率和平均的执行时间可以通过计算得来。其中驱逐的set可以由物理地址通过转换得到，而物理地址可以通过虚拟地址以及/proc/self/pagemap中的页表内容转换得来，改表在android 6.0.1之前可以在没有root权限的情况下访问，之后的版本则需要root权限才可访问该页表。因此，在该系统版本之前的Android设备上可以有效的得到驱逐sets。

对于使用伪随机替换策略的的ARM处理器来说，为Prime + Probe找到一个快速的驱逐策略没有Intel x86系统上直接。甚至是在使用最近最少使用替换策略的Intel x86平台上，也会出现cache污染的情况[14]。Cache污染出现在probe阶段加载相关地址的时候，因为加载的地址可能会驱逐出之后要访问的地址。Tromer et al. 通过使用双向链表的方式来减少cache污染的影响，在prime阶段以前向的方式来访问相关地址，而在probe阶段则反向访问这些地址。Moritz Lipp et al.[24]通过减少prime阶段访问的相关地址数目来减少cache污染对驱逐造成的影响。

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：

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。

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a\_1 a\_2 a\_3，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a\_1，然后访问a\_2，最后访问a\_3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。

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a\_1 a\_2…a\_17到a\_1 a\_1 a\_2 a\_2…a\_17 a\_17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：

参数：待驱逐set m的N个相关地址

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

N表示待驱逐set中可以存放的不同地址数

D表示每个循环访问的不同地址数

A表示每个循环每个地址的访问次数

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。

## 计时方式

Cache时间攻击主要依赖于访存与访cache的时间差别，因此需要在准确且高效的计时方式的基础上才能实现有效的cache攻击。然而在ARM处理器上精确到CPU周期的计时方式需要root权限，这造成cache攻击的局限性。

为了能够有效的将cache hits和cache misses区分开，可以使用时间源或是性能计数器。由于cache misses与cache hits相比有较大的时间延迟，并且时间源在Intel x86 CPUs上有大量的研究，在x86 CPUs平台上的cache攻击利用非特权指令rdtsc来获取纳秒级的时间戳。然而ARMv7-A架构没有提供实现该目标的指令，作为替代，ARMv7-A架构提供了一个性能监测单元来获取CPU的状态。其中的一个性能计数器-周期计数寄存器（PMCCNTR）可以被用来通过记录访存所花的CPU周期数来区分cache hits和cache misses。然而，这些性能计数器默认在用户态是无法访问的，并且需要root权限。

Moritz Lipp et al.[24]使用了几种不需要任何权限的几种时间源来替代性能计数器。主要分为3类：

非特权的系统调用：perf\_event\_open是通过内核来访问硬件无关的性能信息的一个抽象层面的系统调用，其参数如果设置为PERF\_COUNT\_HW\_CPU\_CYCLES则会返回一个精确的CPU周期。这个特性是否可用依赖于Android内核的配置。当该特性可用时，攻击者无需加载内核模型来访问性能信息，因为perf\_event\_open系统调用能够在非特权下调用。

POSIX函数：POSIX的clock\_gettime()函数能够获得精确到微秒到纳秒级别的系统时间。

专用的线程计时器：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。

Moritz Lipp et al.[24]展示了包括周期计数寄存器在内的4中计时方式在Alcatel One Touch Pop 2设备上的运行情况。如图3.3.1所示，尽管有噪音的影响，cache hits和cache misses能够通过4中方式明显的区别开来。因此所有的方法都能用于cache攻击。

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。

# 结论

驱逐策略关系着cache set能够快速高效的驱逐，不仅仅要将L2 cache相关set中的所有line中的数据驱逐，也要讲L1 cache中的相关数据驱逐到内存中。然而由于大多数ARM处理器使用的cache L2伪随机替换策略，要在连续的访存中估计cache set中的数据显得困难。虽然在不支持flush刷新的情况下可以使用相关地址来驱逐set，通过大量访问相关地址达到驱逐目的地址到内存的目的，但这不仅会增加时间和空间存储的开销，也会引入探测阶段的偏差（由于不清楚是哪些地址占用了set）。由此在确定机型上找到快速高效的驱逐策略是难点之一。Cache攻击的攻击程序与被攻击程序的交互方式是攻击能否成功的关键，然而在这个方向目前还没有太多相关的研究。Evict + Reload、Prime + Probe等方法只是探测cache set状态的工具，还需确定具体的攻击过程。由于在evict或prime过后需要被攻击程序执行待攻击的事件，之后再有reload或probe操作来获取cache的旁路信息，然而在这期间由于系统访存或攻击程序自身访存操作可能会对evict或prime阶段占用的cache set中的部分甚至全部line，导致探测阶段误认为待攻击程序访问了该cache set，因此如何处理系统噪音造成的影响也是攻击成功的关键。

# 主要参考文献

1. Gruss D, Maurice C, Mangard S. Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript[M]// Detection of Intrusions and Malware, and Vulnerability Assessment. 2016.
2. Yarom Y, Falkner K. FLUSH+RELOAD: a high resolution, low noise, L3 cache side-channel attack[C]// USENIX Security Symposium. 2014:719-732.
3. Irazoqui G, Eisenbarth T, Sunar B. Cross Processor Cache Attacks[C]// The, ACM. 2016:353-364.
4. Bernstein D J. Cache-timing attacks on AES[J]. Vlsi Design IEEE Computer Society, 2005, 51(2):218 - 221.
5. Kelsey J, Schneier B, Wagner D, et al. Side channel cryptanalysis of product ciphers[J]. Lecture Notes in Computer Science, 1998, 8(23):97-110.
6. Kocher P C. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems[M]// Advances in Cryptology — CRYPTO ’96. Springer Berlin Heidelberg, 1999:104--113.
7. Neve M. Cache-based Vulnerabilities and SPAM analysis[J]. Doctor Thesis Ucl, 2006.
8. Neve M, Seifert J P, Wang Z. A refined look at Bernstein's AES side-channel analysis[C]// ACM Symposium on Information, Computer and Communications Security. ACM, 2006:369-369.
9. Dan P. Theoretical Use of Cache Memory as a Cryptanalytic Side-Channel.[J]. Journal of Arid Environments, 2002, 2002(10):393-446.
10. Tsunoo Y, Saito T, Suzaki T, et al. Cryptanalysis of DES Implemented on Computers with Cache.[J]. Proc of Ches Springer Lncs, 2003, 2779:62-76.
11. Irazoqui G, Eisenbarth T, Sunar B. S$A: A Shared Cache Attack That Works across Cores and Defies VM Sandboxing -- and Its Application to AES[J]. 2015:591-604.
12. Liu F, Yarom Y, Ge Q, et al. Last-Level Cache Side-Channel Attacks are Practical[J]. IEEE Symposium on Security & Privacy, 2015:605-622.
13. Osvik D A, Shamir A, Tromer E. Cache Attacks and Countermeasures: The Case of AES[J]. Lecture Notes in Computer Science, 2005, 2005:1-20.
14. Tromer E, Osvik D A, Shamir A. Efficient Cache Attacks on AES, and Countermeasures[J]. Journal of Cryptology, 2010, 23(1):37-71.
15. Maurice C, Neumann C, Heen O, et al. C5: Cross-Cores Cache Covert Channel[M]// Detection of Intrusions and Malware, and Vulnerability Assessment. 2015:46-64.
16. Kocher P C. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems[C]// International Cryptology Conference on Advances in Cryptology. Springer-Verlag, 1996:104--113.
17. Gullasch D, Bangerter E, Krenn S. Cache Games -- Bringing Access-Based Cache Attacks on AES to Practice[J]. 2011, 2010(1):490-505.
18. Gülmezoğlu B, İnci M S, Irazoqui G, et al. A Faster and More Realistic Flush+Reload, Attack on AES[J]. 2015.
19. Irazoqui G, Inci M S, Eisenbarth T, et al. Know Thy Neighbor: Crypto Library Detection in Cloud[J]. Proceedings on Privacy Enhancing Technologies, 2015, 1(1):25-40.
20. Irazoqui G, Inci M S, Eisenbarth T, et al. Lucky 13 Strikes Back[C]// ACM Symposium on Information, Computer and Communications Security. ACM, 2015.
21. Gruss D, Spreitzer R, Mangard S. Cache template attacks: automating attacks on inclusive last-level caches[C]// Usenix Conference on Security Symposium. USENIX Association, 2015:897-912.
22. Oren Y, Kemerlis V P, Sethumadhavan S, et al. The Spy in the Sandbox: Practical Cache Attacks in JavaScript and their Implications[J]. Computer Science, 2015.
23. Kelsey J, Schneier B, Wagner D, et al. Side Channel Cryptanalysis of Product Ciphers.[C]// Computer Security - ESORICS 98, European Symposium on Research in Computer Security, Louvain-La-Neuve, Belgium, September 16-18, 1998, Proceedings. 1998:97--110.
24. Lipp M, Gruss D, Spreitzer R, et al. ARMageddon: Cache Attacks on Mobile Devices[J]. Mundo Electrónico, 2016, 6(1):págs. 60-65.
25. Gruss D, Maurice C, Wagner K, et al. Flush+Flush: A Fast and Stealthy Cache Attack[M]// Detection of Intrusions and Malware, and Vulnerability Assessment. Springer International Publishing, 2016.
26. Tsunoo Y, Saito T, Suzaki T, et al. Cryptanalysis of DES Implemented on Computers with Cache.[J]. Proc of Ches Springer Lncs, 2003, 2779:62-76.
27. Weiß M, Heinz B, Stumpf F. A Cache Timing Attack on AES in Virtualization Environments[M]// Financial Cryptography and Data Security. Springer Berlin Heidelberg, 2012:314-328.
28. Bernstein D J. Cache-timing attacks on AES[J]. Vlsi Design IEEE Computer Society, 2005, 51(2):218 - 221.
29. OpenSSL the open-source toolkit for SSL/TLS. 2005. <http://www.openssl.org/>
30. Bonneau J, Mironov I. Cache-collision timing attacks against AES[J]. 2006, 4249:201-215.
31. Spreitzer R, Plos T. On the Applicability of Time-Driven Cache Attacks on Mobile Devices[C]// Network and System Security. 2013:656-662.
32. Bogdanov A, Eisenbarth T, Paar C, et al. Differential Cache-Collision Timing Attacks on AES with Applications to Embedded CPUs[M]// Topics in Cryptology - CT-RSA 2010. Springer Berlin Heidelberg, 2010.
33. Gallais J F, Kizhvatov I, Tunstall M. Improved Trace-Driven Cache-Collision Attacks against Embedded AES Implementations[C]// Information Security Applications -, International Workshop, Wisa 2010, Jeju Island, Korea, August 24-26, 2010, Revised Selected Papers. 2010:243-257.
34. Gallais J F, Kizhvatov I. Error-Tolerance in Trace-Driven Cache Collision Attacks[J]. Cosade, 2011:222--232.
35. Spreitzer R, Plos T. Cache-Access Pattern Attack on Disaligned AES T-Tables[C]// Constructive Side-Channel Analysis and Secure Design. 2013:200-214.
36. Percival C. Cache missing for fun and profit[J]. Proc of Bsdcan, 2005.
37. Osvik D A, Shamir A, Tromer E. Cache Attacks and Countermeasures: The Case of AES[J]. Lecture Notes in Computer Science, 2005, 2005:1-20.
38. Neve M, Seifert J P. Advances on Access-Driven Cache Attacks on AES[C]// Selected Areas in Cryptography, International Workshop, SAC 2006, Montreal, Canada, August 17-18, 2006 Revised Selected Papers. 2006:147-162.
39. Aciicmez O. Advances in side-channel cryptanalysis : microarchitectural attacks[J]. 2006.
40. Acıiçmez O, Schindler W, Çetin K. Koç. Cache Based Remote Timing Attack on the AES[M]// Topics in Cryptology – CT-RSA 2007. Springer Berlin Heidelberg, 2007:271-286.
41. Dan P. Theoretical Use of Cache Memory as a Cryptanalytic Side-Channel.[J]. Journal of Arid Environments, 2002, 2002(10):393-446.
42. ARM LIMITED. ARM Cortex-A53 MPCore Processor Technical Reference Manual r0p3. ARM Limited, 2014.
43. LAL SHIMPI, ANANDTECH. Answered by the Experts: ARM’s Cortex A53 Lead Architect, Peter Greenhalgh. http://www.anandtech.com/show/7591/answeredby-the-experts-arms-cortex-a53-lead-architectpeter-greenhalgh, Dec. 2013. Retrieved on November 10,2015.
44. ARM LIMITED. ARM Cortex-A57 MPCore Processor Technical Reference Manual r1p0. ARM Limited, 2013.
45. Hund R, Willems C, Holz T. Practical Timing Side Channel Attacks against Kernel Space ASLR[J]. 2013, 363(9418):191-205.
46. 沈绪榜, 陆铁军. Cache设计的分析与决策[J]. 小型微型计算机系统, 1992(7):1-8.
47. 王志英, 张春元, 沈立. 计算机体系结构[M]. 清华大学出版社, 2015.
48. William Stallings. 计算机组成与体系结构[M]. 机械工业出版社, 2011.