中图分类号：TP311

论 文 编 号 ：10006SY1506402

北京航空航天大學

硕士学位论文

面向移动设备的 cache 攻击关键技术研究

作 者 姓 名　　李 勃

学 科 专 业　　软 件 工 程

指 导 教 师　　姜 博 副 教 授

培 养 院 系　　计算机学院

Research on Key Technologies of Cache Attack for Mobile Devices

A Dissertation Submitted for the Degree of Master

Candidate： Bo Li

Supervisor： Associate Prof. Bo Jiang

School of Computer Science and Engineering

Beihang University, Beijing, China

中图分类号： TP311

论 文 编 号 ： 10006SY106402

硕 士 学 位 论 文

面向移动设备的cache攻击关键技术研究

作者姓名	李勃	申请学位级别	学士硕士
指导教师姓名	姜博	职　　称	副教授
学科专业	计算机系统结构	研究方向	移动安全
学习时间自	2015 年 9 月 10 日	起至	2018 年 2 月　　日止
论文提交日期	2018 年　　月　　日	论文答辩日期	2018 年 3 月 10 日
学位授予单位	北京航空航天大学	学位授予日期	年　　月　　日

关于学位论文的独创性声明

本人郑重声明：所呈交的论文是本人在指导教师指导下独立进行研究工作所取得的成果，论文中有关资料和数据是实事求是的。尽我所知，除文中已经加以标注和致谢外，本论文不包含其他人已经发表或撰写的研究成果，也不包含本人或他人为获得北京航空航天大学或其它教育机构的学位或学历证书而使用过的材料。与我一同工作的同志对研究所做的任何贡献均已在论文中作出了明确的说明。

若有不实之处，本人愿意承担相关法律责任。

学位论文作者签名：_____ 日期：　　年　　月　　日

学位论文使用授权书

本人完全同意北京航空航天大学有权使用本学位论文（包括但不限于其印刷版和电子版），使用方式包括但不限于：保留学位论文，按规定向国家有关部门（机构）送交学位论文，以学术交流为目的赠送和交换学位论文，允许学位论文被查阅、借阅和复印，将学位论文的全部或部分内容编入有关数据库进行检索，采用影印、缩印或其他复制手段保存学位论文。

保密学位论文在解密后的使用授权同上。

学位论文作者签名：_____ 日期：　　年　　月　　日

指导教师签名：_____ 日期：　　年　　月　　日

摘　要

Cache 攻击是通过利用 Cache 泄露的内存访问模式来获取用户私密信息的一种攻击方式。自从揭露了这一漏洞之后，很多安全研究人员基于此漏洞，设计了很多强有力的攻击模式，成功的在 Intel x86 平台上实现了针对 DES、AES 加密算法的攻击，破解了部分甚至全部的密钥。除此之外，也有工作者通过 Cache 攻击获取了用户的输入，甚至通过 Cache 在程序间进行通信。然而，由于 Android 等移动设备的指令集、Cache 结构及 Cache 替换等与 Cache 攻击密切相关的属性与 Intel x86 平台不同，因此虽然很早以前就在 x86 平台上证明了 Cache 攻击是有效的攻击方式，但直到最近才有人提出基于 ARM 处理器的 Cache 攻击方式，并且很少有基于移动平台的 Cache 攻击成功的案例。本文致力于移动设备上的 Cache 攻击研究，设计并实验针对 AES 加密程序的 Cache 攻击程序，总结攻击成功的关键因素，分析并提出预防攻击的措施。

本文以 Lenovo k51c78 作为目标机，将 AES 加密算法作为攻击对象，设计并实现了在 Android 平台上的 Cache 攻击。首先，介绍了 Cache 攻击的背景以及研究的意义、Cache 的结构、Cache 攻击的基本模式，并详细介绍本文的攻击对象 AES 加密算法，并强调其加密过程中泄露出来的漏洞。其次，分析了在目标机系统上能够实现精确计时的方式，并设计了能够满足实验要求且不需要额外权限的线程模拟计时器，该计时器能够满足系统未提供可用计时方式的极端情况下的计时需求。接着，给出了获取目标机快速高效驱逐策略的方案，该方案通过脚本对各种不同的驱逐策略进行比较，最终选择一个既快速又高效的驱逐策略。随后，本文根据移动设备的特点设计了针对 AES 加密程序的 Cache 攻击方案，在该方案中，攻击程序和 AES 加密程序运行与同一个 CPU 上，攻击程序不断的监测 AES 加密过程中对 Cache 的使用情况，并将 Cache 泄露的信息写到指定的文件中，当攻击结束时，通过分析程序对该文件进行分析，获取到每一个假设密钥$\hat{k}$的度量分数$\hat{m}$，则度量分数最高的假设密钥$\hat{k}$就是通过Cache攻击获取的用户的密钥。之后本文介绍了攻击过程中的关键技术以及本文给出的解决方案。

通过本文的实验，证明了不仅仅在 Intel x86 平台上，在 Android 等移动设备上，Cache 攻击也是一种强力有效的窃取用户隐私的方式，为了保护用户的隐私，本文最后提出了几点防护措施。

关键词： Cache 攻击，旁路攻击，AES 破解，AES 攻击

Abstract

The Cache attack is an attack mode that USES the memory access mode to obtain the user's private information. Since revealed this loophole, many security researchers based on the vulnerability, design a lot of strong attack mode, successful on Intel x86 platform realizes the attacks on DES, AES encryption algorithm, cracked some or all of the key. In addition, some workers get the user's input through the Cache attack and even communicate through the Cache. However, due to the Android mobile devices such as instruction set, the structure of the Cache and Cache replacement is closely related to the Cache attacks such as the properties of the different from Intel x86 platform, although so long ago on x86 platform to justifying the Cache attack is an effective way of attack, but until recently been proposed based on ARM processor Cache attack way, and there are few Cache attack successful examples based on mobile platform. This paper is dedicated to the Cache on the mobile device attack research, design and experiment for AES encryption program Cache attacks, summarize against the key success factors, analysis and put forward measures to prevent attacks.

In this paper, Lenovo k51c78 is used as the target machine, and AES encryption algorithm is used as an attack object, and the Cache attack on the Android platform is designed and implemented. First, this paper introduces the Cache against the background and research significance, the basic model of the structure of the Cache, the Cache attacks, and introduced in detail in this paper, the target of AES encryption algorithm, and stressed that leak out of holes in the encryption process. Secondly, analyzes on the target system can realize precise timing, and designed can satisfy the experimental requirement and does not need additional permissions thread simulated timer, the timer can satisfy the system did not provide a time of extreme cases timing requirements are available. Then, gives access to the target of fast and efficient drive strategy plan, the plan through the script on the comparison to various drive strategy, finally select a fast and efficient drive strategy. Then, in this paper, based on the characteristics of mobile devices designed for AES encryption program Cache attack plan, in the scenario, the attack programs and AES encryption program is run with the same CPU to attack programs in the process of continuously monitoring the AES encryption to Cache usage, and the Cache

leaked information written to the specified file, when at the end of the attack, by analyzing the application analysis of the file, access to each hypothesis key k^ metric fraction m^, then measure scores the highest assumes the key k^ attack is access to user keys. Then this article introduces the key techniques in the attack and the solution presented in this paper.

Through the experiment, this paper proves that not only on the Intel x86 platform, the Android mobile devices, such as the Cache attack is also a powerful and effective way to steal user privacy, in order to protect the privacy of users, this paper finally puts forward some protective measures.

Key words: Cache attack，Cache side-channel attack，AES attack

目 录

图 目

表 目

第一章 绪论

1.1 研究背景及意义

从第一台手机诞生到现在，伴随着移动互联网突飞猛进式的增长，手机等移动设备已经成为我们日常生活中不可或缺的一部分。它在给我们的生活带来便利的同时也对我们的隐私带来了潜在的威胁，尤其是手机存储着包含账号、密码等用户私密信息的现在。因此，随着技术的发展，隐私安全问题也越来越得到人们的重视。虽然移动设备开发厂商以及 Android、Windows 等操作系统采取了很多安全措施来保护用户的隐私，包括虚拟内存管理、权限管理、可执行环境等技术。然而，由于系统硬件的一些固有缺陷，一些针对计算机以及移动设备的攻击方式逐渐显现出来，其中就包括本文所研究的 cache 攻击。其攻击方式为基于 cache hit 与 cache miss 的时间差异来探测被攻击程序执行过程中的内存及 cache 的访问情况，并依次获取被攻击程序私密信息。由于攻击程序和被攻击程序之间没有直接的交互，并可以运行在不同的核上，访问各自的内存区域，因此不需要额外的权限就能进行攻击，因此是一种强大的攻击方式。

期初，在 Intel x86 平台上的 Cache 攻击得到了科研工作者的重视，并有很多学者从事这一方向的研究。Kocher[22]和 Kelsey[20]提出了通过分析高速缓冲存储器 cache 在运行时泄露出的信息来破解计算机加密算法的方法。基于这些思想，近些年不断的有人提出在 Intel x86 上进行 cache 攻击的方法，并成功的实现了监听用户输入、破解加密算法等攻击，也反向证明了 Cache 攻击是有效的。比如，Page et al.[32]提出了一种破解 DES 加密技术的Cache攻击模型，并将DES密钥的搜索空间从56位降低到了32位。Tsunoo[38]随后提出通过分析 DES 加密过程的查表索引以及其 Cache 访问特性的方式，首次实现了针对 DES 密钥的 Cache 攻击，并在一台 600-MHZ 的 Pentium III 个人电脑上通过 223 个明文成功的获取了 DES 全部密钥。Dan[7]研究使用内存设计单通道攻击。Gallais[9]研究了 Cache 攻击中误差容忍相关的技术，旨在增加攻击的效率。Bernstein[3]在消除网络传输延迟的条件下实现了针对 OpenSSL[29]中 AES 加密的时间驱动攻击，其中，服务端使用 AES 加密算法加密数据。Percival[33]提出了多线程间共享 Cache 缓存可能产生的漏洞，该漏洞其他线程提供一个简单的，高带宽的隐秘铜套，通过使用该漏洞，恶意程序能够监视其他线程，窃取被攻击程序加密算法的密钥等信息。依此 Percival 设计并实

现了一种针对 RSA 加密算法的攻击方式。鉴于 Percival 提出的方法，Osvik[31]设计了一种针对 AES 加密技术的 Cache 计时攻击方式。Gruss[11]研究了基于 Flush+Reload 方法的快速攻击策略。Irazoqui [16]主要针对多处理器条件下的 Cache 攻击技术进行研究。Irazoqui[18]对云端加密库漏洞进行监测，确认攻击的可能性。Liu [24]实现了基于最后一级 Cache 单通道攻击实践。Spreitzer [36]则针对移动设备上计时驱动的 Cache 攻击进行研究。Gruss[10]和 Oren [30]则针对 JavaScrip 代码实现的程序进行 Cache 攻击。Yarom[40]通过 L3 Cache 实现了基于 Flush+Reload 的攻击。Neve[26][27][28]则将 Osvik 的切入点转移到 AES 加密过程中的最后一轮，提出一种新的最后一轮驱动的 Cache 攻击方法。通过这个漏洞，攻击者能够轻易的获取到包含用户输入、加密算法密钥等在内的私密信息，进而威胁到用户的人生财产安全。

然而，由于 Android 等移动设备的结构与 Intel x86 设备的结构有很大的区别，在指令集合、cache 组织方式以及 cache 替换策略等与 cache 攻击密切相关的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 的手机上执行有效的 cache 攻击方法。Moritz Lipp et al.[23] 提出了通过在移动设备上进行 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。其中最典型的攻击模型为 cache 模板攻击，该模型在探测阶段不断的探测待攻击时间执行时共享库地址空间的加载情况，形成一个 cache 模板矩阵，该矩阵对应着某一事件执行时共享库各个地址的访问情况。在通过该模板矩阵进行攻击时，探测用户执行事件时共享库地址的加载情况，并与 cache 模板矩阵进行对比，进而分析出用户执行的操作。通过近些年的研究，cache 旁路攻击已经被认可为一种强大的攻击方法，在研究攻击方法的同时也提出了一些修复漏洞的方法，包括在 Android 6.0.1 中修复了对 /proc/self/pagemap 文件的访问权限。通过控制在非特权模式下用户线程对自己存储空间页表映射表的权限，能够有效的抑制攻击程序通过共享库执行 cache 模板攻击。

在国内，由于起步较晚，针对 Cache 攻击的研究停留在重复国外论文实现的层次上，2009 年起，赵新杰[43][44][45]等人，研究了在 Intel x86 平台上针对 AES 的 Cache 计时攻击技术，2015 年，邓柳于勤[42]等人研究了基于 ARM 处理器的 Cache 计时分析的特

性研究，唐烨[41]等人则在 2016 年对针对 AES 加密算法的缓存攻击进行研究。

基于这些背景，本题旨在发掘更多在移动设备端的 cache 攻击方式，以及有效的攻击模式，主要针对用户输入、高级加密技术（AES）以及可信执行环境，探索并实现有效的 cache 攻击。挖掘与用户输入和 AES 加密有关的可能发生信息泄露的漏洞，进一步针对攻击提出一些应对措施，从而促进移动设备端安全化进程，使用户能够更加安全，更加放心的使用手机。

1.2 研究内容及目标

本文的研究目标为研究在移动设备端 Cache 攻击关键技术，实现具体的端到端的基于 AES T-table 的 cache 攻击，并依据对攻击过程的总结提出能有效预防移动设备端 cache 攻击的建议。

为了最终实现本文的研究目标，本文主要的研究内容可以分为 5 部分：

1. 研究 Cache 的结构以及 Cache 泄露出的漏洞，讨论 Cache 攻击的几种策略，研究 AES 加密过程中的漏洞。
2. 研究 Cache 攻击依赖的计时方式和针对移动设备的驱逐策略。
3. 研究在移动设备上 Cache 攻击的设计。
4. 针对移动设备与 Intel x86 的区别，研究减少误差的手段。
5. 总结系统和软件的漏洞，提出移动设备预防 Cache 攻击的措施。

1.3 本文研究限制

Cache 攻击是与硬件结构密切相关的攻击方式，虽然在进行攻击时不需要获取额外的权限，但由于不同的移动设备的硬件结构不一样，而诸如 cache 大小、cache set 数、cache 的替换策略、实虚存转换权限等参数是与 cache 攻击是否成功密切相关的，因此在一台设备上成功的攻击模式可能在其他的型号设备上并不奏效。此外，同一设备，不同版本的操作系统提供的支持也不相同，比如 Android 6.0.1 后限制了非特权用户对 /proc/< pid >/pagemap文件的访问，因此使用较新系统的设备无法通过访问该表获取虚拟内存和物理内存的映射关系，也无法使用本文设计的方法进行 Cache 攻击。不同版本的操作系统，提供的诸如计时接口也会有差异，有可能在 Android 5.0.1 系统上能够使

用POSIX提供的接口实现精确的计时方式，但是到 Android 6.0.1 时却不能。因此，本文的方法仅仅局限于选定的 Lenovo k51c78 目标及，但思路是通用的，在不同的设备不同的系统版本上，在获取了精确的计时方式、快速高效的驱逐策略之后，本文设计的攻击方式都是有效的。

1.4 本文的组织结构

本文一共由六章组成，包含的主要内容如下内容：

第一章 绪论。包含 Cache 攻击的背景以及在移动设备端进行 Cache 攻击的现状，随后指出移动设备研究 Cache 攻击的意义，随后，描述了本文的研究内容和研究目标，以及本文的研究限制。

第二章 Cache 攻击的相关技术以及原理。这章主要包含与 Cache 攻击密切相关的 Cache、内存以及被用作攻击目标的 AES 加密技术相关的知识，具体包含 CPU caches 结构，Cache 与内存映射方式，Cache 替换策略，缓存 Tag 以及 Index，Cache 包含性，最后介绍了两种有效的 Cache 攻击模式。

第四章 攻击方案的设计。本章介绍实现 Cache 攻击过程中比较重要的几个功能模块的设计，首先研究移动平台上如何实现精确的计时方式，接下来介绍驱逐策略以及攻击方式的设计，最后描述了针对 AES 攻击的一步攻击设计，最后给出了 cache 攻击相关的关键技术及其解决方案。

第四章 攻击方案的设计。本真主要介绍了针对 AES 机密算法的 Cache 攻击过程设计，并引入 KS 检验减少由于伪随机替换算法和计时方式引入的误差，并在最后介绍了攻击过程中的关键技术和解决方案。

第五章 AES 攻击验证。本章主要分为 4 个部分，第一部分介绍了通过在目标机上对多种计时方式进行测试，最终选择计时最精确的计时方式为后续实验提供支持第二部分介绍了通过脚本的方式获取针对目标机有效的 cache 驱逐策略。第三部分实现了针对 AES 加密算法的同步攻击。第四部分对异步攻击方式进行探索。

预防攻击措施。详细介绍了移动设备暴露出来的攻击漏洞，以及预防措施。

第二章 相关技术以及原理

2.1 Cache 结构

cache 攻击就是攻击者针对 cache 结构，设计程序通过 cache 获取被攻击程序的内存访问情况，据此分析出用户的私密信息，例如用户的按键输入、秘钥等。熟悉 cache 结构就成为设计攻击程序或对其进行防卫的必要条件，本章主要包含 CPU cache 的详细介绍、cache 组织、cache 与内存的映射关系以及 cache 攻击技术等信息。

2.1.1 CPU Caches

当代计算机以及各种移动设备的 CPU 性能不仅仅依赖与其时钟周期，还受到其指令集以及和其他设备的交互的影响。因为内存应该尽可能快的向 CPU 提供其中存储的数据，因此如下图所示的层次图被设计出来缩小内存与 CPU 的速度差异。通常情况下来说，从 cache 中读取数据要比从内存中读取快很多倍，因此称之为缓存。

由于访问速度越贵的内存价格越昂贵，内存架构组织成金字塔型的层次结构在访问速度和价格之间做折中，越靠近 CPU 的存储访问速度越快，价格也越昂贵，越靠近内存的存储价格越便宜，相应的访问速度也越慢。因此访问存储在缓存中的数据要比访问存储在内存中的速度要快很多，需要的时间短很多。

为了性能方面的考虑，通常情况下 L1 缓存直接与 CPU 中获取指令以及加载和存储数据的核心逻辑相连。对于冯诺依曼体系结构的计算机，通常情况下只使用一个 cache 用于存储指令和数据，然而哈佛体系结构的计算机则分别有一个指令缓存 I-Cache 和一个数据缓存 D-Cache,顾名思义 I-Cache 用于缓存指令而 D-Cache 用户缓存数据，并且指令缓存和数据缓存通过两条总线可以同时传输数据，因此数据读取速度更快一些。下图为 Cortex 的 Cache 结构。

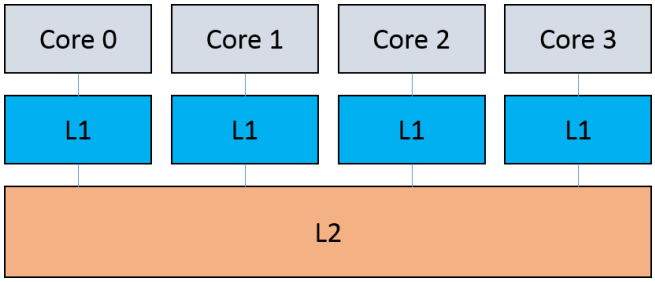

图 1 ARM Cache 结构

程序更倾向于访问已经访问过的地址以及附近的地址，比如在一个循环中，相同的代码被一遍又一遍的执行，这也被称之为程序运行的局部性原理。因此，为了提高程序的运行速度，需要尽可能的将之后需要访问的指令及数据提早缓存到 cache 中，间接提高访存速度。对一些追求实时性的硬件及程序 cache 使得读取指令或数据的时间存在不确定性，会导致问题的发生。

Cache 仅仅缓存了主存中的一部分数据或指令，因此 cache 必须能够记录下这部分数据的地址以及其相关的内容。当 CPU 想要加载或修改某一地址的数据时，它首先去 L1 cache 中查找，看所需要的数据在不在缓存中。如果不在缓存中，则 CPU 必须到更低一级的缓存或主存中去查找数据，以确保指令流水能够顺利的执行。出自性能方面的考虑，每次会从内存中将一个内存块缓存到 cache 中，也就是一个 cache line，其中每个内存块包含 B 个 Bytes，这也是 cache 缓存的最小单位。因此当一个地址被缓存到 cache 中时，其附近地址的数据也被一同缓存到 cache 中，根据程序的局部性原理，这些数据很有可能是程序不久将要访问到的，因此能够减少从内存中获取数据频率，增加程序的运行速度。其中每个 cache line 都有一个 tag，用于判断某地址的数据是否被被缓存到该 cache line 中。

2.1.2 全相联映射

有很多种 cache 实现的方式，其中最简单的就是直接相联映射。在一个基于直接映射方式实现的 cache 中，主存中的每个内存块只能唯一对应 cache 中指定位置的 set 中，这种对应方式是不能改变的，也就是说主存中的数据要么只能被缓存到 cache 中相应的 set 中，要么就是仅存储在内存中。由于主存大小远远大于 cache 容量，因此会有多个内存块对应到同一个 cache set 的情况。下图展示了一个拥有 128 个 set，每个 set 64 字节，总容量为 4KB 的 cache 与内存的映射情况。

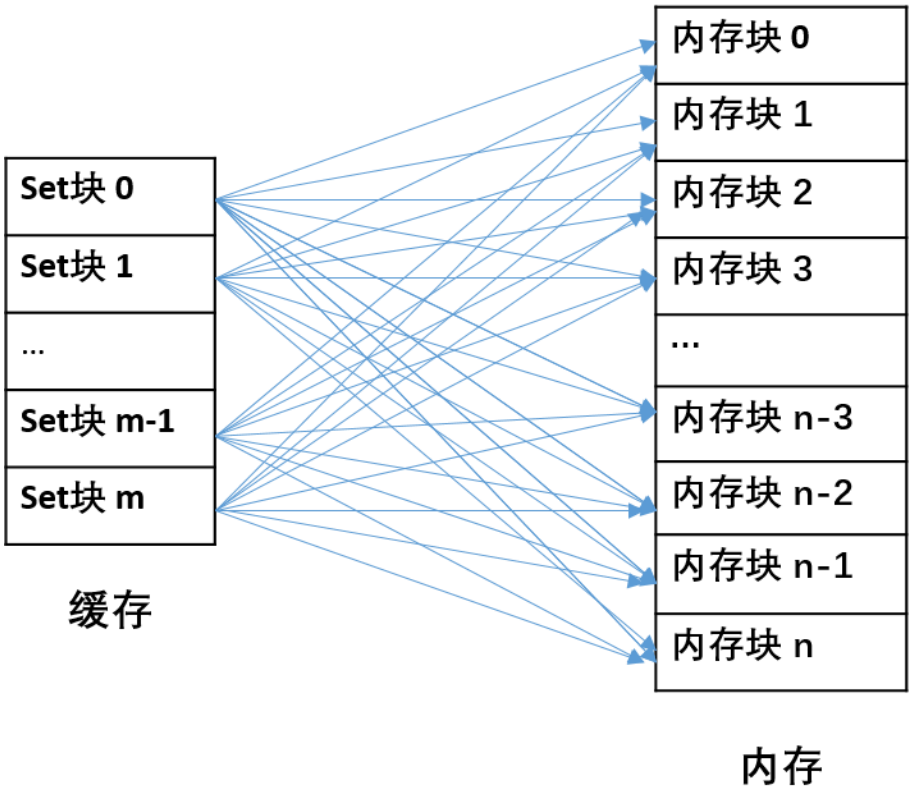

图 2 全相联映射

为了在 cache 中获取指定地址的数据，首先获取地址的 index 位，将地址中的 tag 位与 cache 相应 index 位置数据的 tag 位进行比较，如果 tag 位相等，表示 cache 命中，则根据 offset 位将响应位置的数据传输到 cache 中或修改改位置的数据。如果 tag 位不相等，则表示 cache 缺失，及所查找的数据不在这一级 cache 中，进一步将从下一级 cache 或从内存中获取该地址对应的数据到 cache 中，并替换之前缓存在该 cache line 的数据并更新 tag。

2.1.3 组相联映射

如果替换策略能够选择 cache 中的任何块存储数据，则这样的 cache 结构称为全相联映射。现在的 cache 被组织为若干个 set，其中每个 set 包含固定数目的 cache line，每个 cache line 对应内存中的一个数据块，这种组织方式被称为组相联映射。如果组相联映射结构中一个 cache set 包含 N 个 cache line，则将其称之为 N-way 组相联。因此，每个内存地址及其周围的一个块的数据映射到其中一个 cache set 中，且这些数据缓存在这个 set 中的任意一个 cache line 中。任意两个能偶映射到同一个 cache set 的 address 成为是相互关联的。关联的地址总会竞争同一个 cache set 中的 line，并由驱逐策略决定被驱

逐的 line。为了方便理解，如下描述本文用于实验的 Lenovo K51c78 测试机的 L2 cache，其 cache 总容量为 512KB，总共包含 512 个 set，cache 与内存的映射关系为 16 路组相联，因此 cache index 位为 9 位。

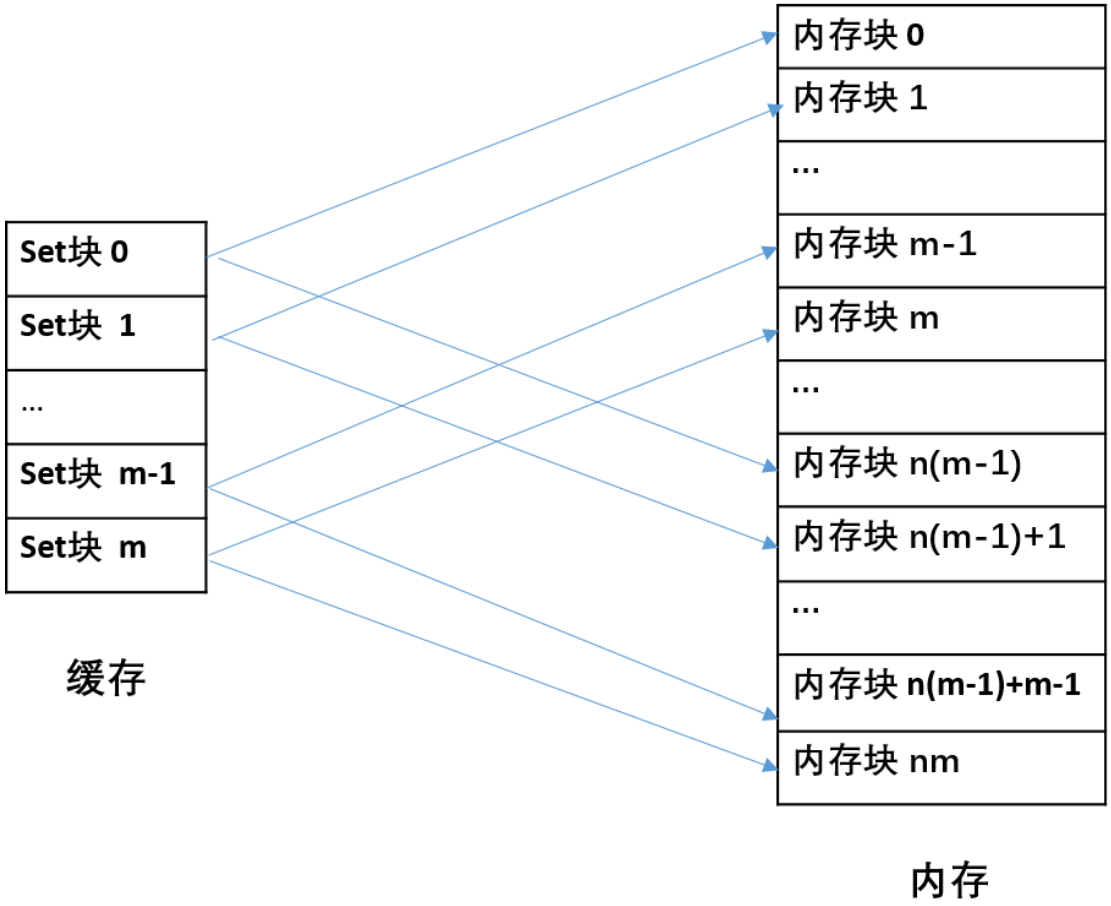

图 3 组相联映射

2.1.4 Cache 替换策略

在组相联结构的 cache 中，当 cache set 中每个 cache line 都缓存数据或当 cache 确实发生时，缓存的代码或数据必须从 cache 中驱逐到内从中以缓存新的数据或指令。决定驱逐哪个 cache line 的算法就被称之为替换策略。替换策略必须决定 cache set 中的哪些数据在将来不会被使用到。最近最少替换算法 LRU 总会替换掉最近没有被使用过的 cache line 中的数据。然而 ARM 结构的处理器的 cache 通常使用称之为 pseudo-LRU 伪随机替换策略，虽然也有 ARM 处理器使用 LRU 替换算法作为 cache 的替换策略，然而由于性能方面的考虑，在实际中通常使用伪随机替换策略。伪随机替换策略 Pseudo-random 将根据伪随机数生成器决定驱逐 cache set 中的哪个 cache line。

2.1.5 Tag 以及 Index

CPU 能够分别使用基于虚拟地址或物理地址的虚拟 index 或物理 index。通常虚拟索引 cache 比物理索引 cache 要块，因为虚拟索引在 cache 查找时不要将虚拟地址转换为物理地址。但使用虚拟索引会导致相同的物理地址被缓存到不同的 cache line 中，这会导致性能的下降。为了唯一的标记缓存到指定 line 中的地址，通常使用 tag 标签来进行标记。同样，tag 也能时虚拟的或者物理的。可能的搭配当时的优缺点如下所示：

VIVT-虚拟索引，虚拟标签

索引和标签都是使用的虚拟地址，由于在查找 cache 时不需要地址转换，因此查询速度较快。然而，由于虚拟标签不是唯一的，共享内存可能在 cache 中缓存多次

- PIPT-物理索引，物理标签

索引和标签都是使用的物理地址，由于查找 cache 时需要将虚拟地址转换为物理地址，因此速度较慢，不过共享内存仅会缓存一份到 cache 中。

- PIVT-物理索引，虚拟标签

物理地址用于索引，标签使用的虚拟地址，这种组合内有任何好处，因为查找时既需要将虚拟地址转换为物理地址，并且共享内存也可能在 cache 中缓存多份

- VIPT-虚拟索引，物理标签

虚拟地址用于索引，物理地址用于标签，这种组合相较 PIPT 的优势在于通过并行查找 TLB 转换物理地址，然而只有当物理地址转换完毕后才能比较 tag 标签。

2.1.6 Cache 包含性

为了提高 cache 命中的几率，CPUs 通常使用多级缓存 L1～Ln，其中越靠近 CPU 的缓存速度越快，价格也越贵，容量也越小。就像先前描述的那样，本文实现基于的目标机 Lenovo k51c78 的 cache 分为两层，L1 cache 结构为 4 路组相联，大小为 32KB，共有 128 个 sets，而 L2 cache 结构为 16 路组相联，大小为 512KB，共有 512 个 sets。此外目标机利用了哈佛体系 cache 结构的变种，及 L1 cache 分为指令 cache 和数据 cache，但它们使用同样的地址空间。每个核均有一个 L1 cache，由于 k51c78 共有 4 个核心，因此共有 4 个 L1 cache，以及 4 核公用的 1 个 L2 cache。

如果从 cache 中读取一个字，则存在与 cache 中的该字应该与该字对应在内存中的内容相等。然而，当有核执行存储指令企图修改数据时，在写操作执行之前先要查找对

应的数据是否缓存在 cache 中，如果 cache 命中，则有如下两种策略：

- 写回策略：

写会策略将数据写到 cache 中，因此会存在 cache 与内存数据不一致的问题。当没有将被写过的数据写会内存时这种方式没有什么问题，但当 cache set 中 cache line 都被占用，并且驱逐策略决定将当前 cache line 中的数据驱逐到内存中时，就需要判断该 line 中是否存有被修改过但还没有来得及写到内存中的数据。其中判断依据是以 cache line 的额外字段描述的，及脏位，若脏位被置为 1，则表示该 line 存有还未写到内存中的数据，这时需要先将 line 中的数据写会到内存中，再写入要缓存的数据。

- 写穿策略：

写穿策略在 CPU 执行存储命令时将要更新的数据同时写到 cache 和内存中，保持主存和缓存数据的一致性，由于每次写操作都需要执行写存操作，而内存访问速度远远慢于访问 cache 的速度，因此执行速度较慢。

对于包含多层缓存的处理器，cache 需要决定将数据缓存到那一层中，因此有几种不同的存储策略：

- 包含式缓存：

对于低级别的缓存来说，一个高级别的缓存是包含式的，当缓存在低级别缓存中的数据同时也缓存在高级别的缓存中。因此，对于包含式缓存，低级别缓存中的数据时高级别缓存中数据的一个子集。

- 排除式缓存：

当一个 cache line 仅仅能保存在两级缓存的其中一级中时，该 cache 被成为时排除式的。

- 非包含式缓存：

如果一个缓存既不是包含式的也不是排除式的，则这种缓存结构称为是非包含式缓存。

现代的因特尔 CPUs 的最后一级 cache 是包含式的，AMD CPUs 的最后一级 cache 是排除式的，而大多数 ARM CPUs 的最后一级 cache 是非包含式的。然而 ARM Cortex-A53/Cortex-A57 CPUs 的最后一级 cache 是包含式的。

2.2 Cache 攻击策略

cache 边信道攻击主要利用从内存加载数据与从缓存加载数据时间上的差距泄露的信息来进行攻击的。由于读取缓存在 cache 中的数据要比从内存中获取数据要快很多，通常从 cache 中读取数据只需要不到 1ns，而从内从中获取数据要几十到几百纳秒的时间，大了 2 个数量级，因此能够通过判断指定的数据是否缓存在 cache 中来推断其最近有没有被访问过。其中泄露出来的访问信息是潜在的漏洞，特别对于加密算法，因为其可能导致秘钥被破解。

早在 1996 年，Kocher[22]提出了通过度量加密操作所需的总时间来破解加密系统的想法，Kocher 也成为第一个提出通过 CPU cache 泄露的信息在不读取加密相关数据情况下破解加密系统的人。4 年后 Kelsey et al.[20]提出了旁信道攻击的概念，并且断言基于 cache 命中率对 Blowfish[6]以及 CAST[34]使用 S-boxes 的加密算法进行攻击时可行的。Page[32]和 Tsunoo et al.[38]基于 Page 提出的理论对数据加密标准(DES)进行攻击。同样也有针对高级加密技术 AES 的 cache 攻击的研究，比如 Bernstein[3]提出了针对 AES 加密技术的著名的 cache-timing 攻击，Neve[28]及 Neve et al.[26]对 cache-timing 进行了深入的分析。

本节将描述几种有效的 cache 攻击模式，这些攻击模式在 Intel x86 平台上能够有效的获取 cache 泄露的信息，包括 Evict+Time，Prime+Prime，Evict+Reload 等。

2.2.1 Evict+Time

2005 年，Percival[33]和 Osvik et al.[31]提出了能够更有效的从 CPU cache 获取私密信息的方法，Osvik et al.还规范了这两中概念，及 Evict+Time 以及 Prime+Probe，将在随后讨论。它们的基本思想都是判断 cache 中的哪些 set 被被攻击程序访问过。

Evict+Time 算法：

1.测量被攻击程序的执行时间

2.驱逐 cache 中指定的 set

3.再次测量被攻击程序的执行时间

Evict+Time 算法可以用来决定指定的 cache set 在被攻击程序执行期间有没有访问到。首先，测量得到被攻击程序的执行时间 t1，在下次测量被攻击程序执行时间之前，指定 cache set i 中的数据被驱逐到内存中，并测量得到被攻击程序的执行时间 t2。最后通过比较 t1 与 t2 的区别来判断被攻击程序在执行过程中有没有访问到指定的 set。若 t1

小于 t2，则表示驱逐 set i 中的数据增加了被攻击程序的执行时间，也就是驱逐操作将被攻击程序之前缓存在该 cache set 中的数据驱逐到内存中，也就是被攻击程序执行过程中需要访问到 set i。

Osvik et al.[31]和 Tromer et al.[37]提出 Evict+Time 能够对基于 OpenSSl 实现的 AES 实现强有力的攻击，并且在攻击过程中不需要加密过程中的明文以及密文的信息。

2.2.2 Prime+Probe

第二种攻击模式是由 Osvik et al.提出来的，并将其称之为 Prime+Probe。与 Evict+Time 相同，通过 Prime+Probe 攻击者能够判断 cache 中的某些 set 有没有被被攻击程序访问到。

Prime+Probe 算法：

1.占用指定的 cache sets

2.执行被攻击程序

3.检测 cache 中的哪些 sets 仍然被占用

Prime+Probe 算法主要由 3 个步骤组成，首先，攻击程序通过其内存空间的数据占用一个或多个指定的 cache set，对于使用 LRU 替换策略的 cache，通过连续读取能够映射到指定 set 的与该 set 所包含的 cache line 数量相等的数据，则可将该 set 之前缓存的数据全部驱逐到内存中，且需要确保读取的数据在攻击程序的内存空间中。此后，执行被攻击程序，被攻击程序执行过程中的访存操作会占用 cache 中的某些 set 的某些 line，并根据替换算法替换之前存在的部分数据。最后，攻击判断在第一阶段读取到 cache 中的数据是否还缓存在 cache 中。

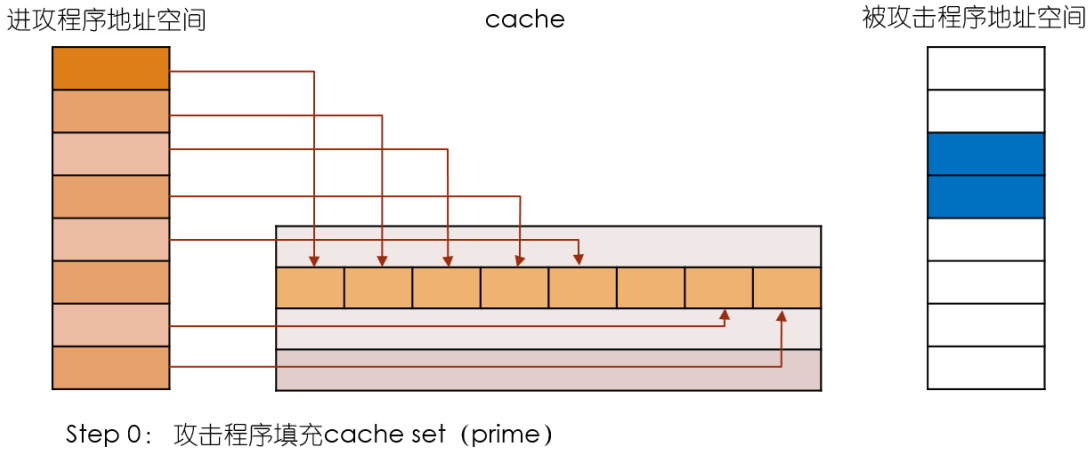

图 4 Prime 占用 Cache line

图 3 展示了攻击的详细过程，网格代表一个 8 路组相联（列）的 cache，共有 4 个 sets（行），且假设攻击者旨在检测被攻击程序在执行过程中对 cache set 1 的占用情况。在步骤 a 中，攻击程序通过连续访问能够映射到 set 1 中的数据来占用 set 1 中的 8 个 cache line，由于该 cache 使用的组相联的映射方式，连续两个能够映射到该 set 中的地址之间的间隙为 4×B 字节，B 为一个 cache line 所包含的字节数，为了将一个 cache set 中的所有数据清除干净，至少需要读与 cache set 所包含的 line 数相等的相关地址，本例中则至少需要读 8 个相关地址，记为m1,…,m8。在执行完步骤 a 后，cache set 1 中的所有 line 均缓存着攻击程序的数据，由于该 cache 为 8 路组相联，因此可以将缓存在该 cache set 中的数据表示为 b1、b2、b3、b4、b5、b6、b7、b8，每个字母表示一组 cache line 大小的数据，并由橙色标识。

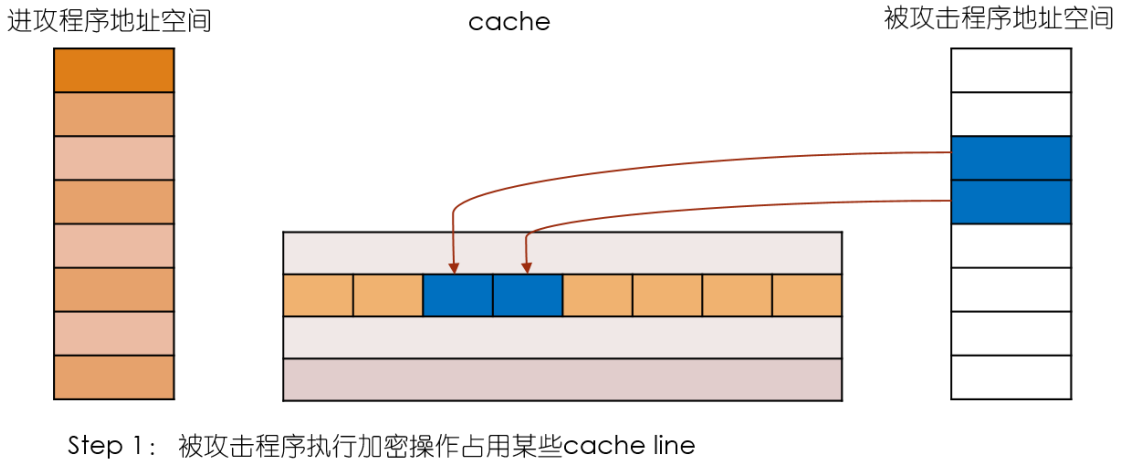

图 5 被攻击程序占用 Cache set

在步骤 b 中，被攻击程序执行，在其执行过程中，由于不断的执行指令并访问数据，为了提高访存速度，操作系统会将这些数据所在的数据块缓存到 cache 中，具体缓存到那个 cache set 中是由数据的起始地址决定的，而具体存储到哪个 cache line 中是由相应的替换算法决定的。如图所示，被攻击程序执行过程中访问的部分数据对应的 cache set 也为 set 4，且 set 4 中的所有 line 均被占用，因此 cache 根据替换策略替换了该 set 中的部分 line。在这一过程中缓存的数据使用蓝色标识，如图 5 所示。

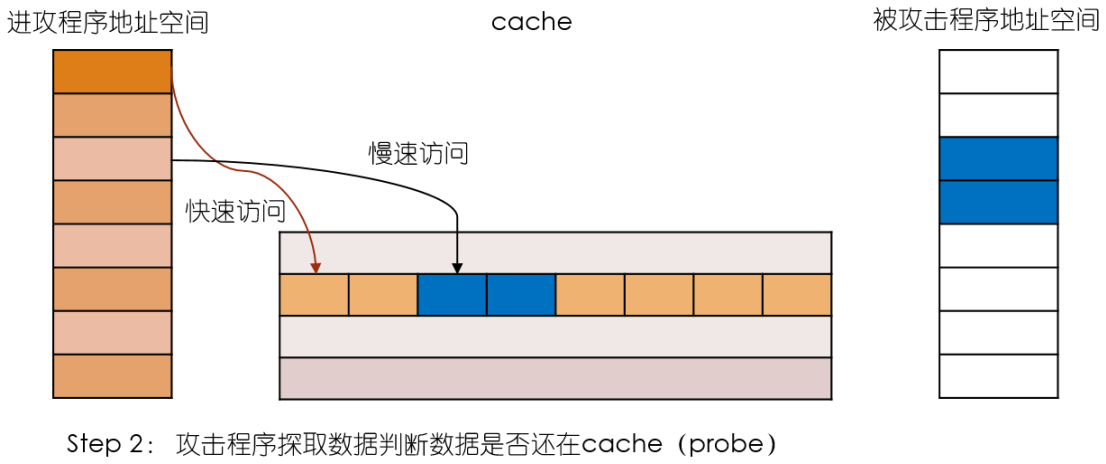

图 6 Probe 探测数据是否还在 Cache 中

如图 5 所示，在步骤 c 中，攻击程序再次访问步骤 a 中访问过的地址 m1,…,m8，并判断这些在步骤 a 中缓存到 cache set 中的数据是否还保留在 cache 中。判断的依据就是当再次访问这些数据时，如果访问时间快，则表示数据依然在 cache 中，如果访问时间较慢，则表示数据在被攻击程序执行期间被驱逐到了内存中，需要再次从内存中获取数据，因此导致较长的访问时间。

2.3 AES 加密

AES 加密技术是由美国国家标准研究所提出的，为了代替 DES 数据加密技术的高级加密技术。它是当前使用得最广的对称加密算法之一，其加密过程和解密过程使用的密钥是相同的。其加密过程如下图所示：

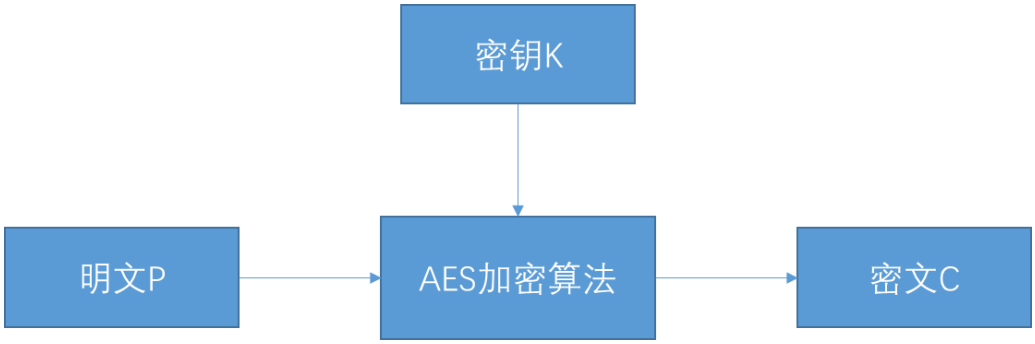

图 7 AES 加密过程

其中明文 P 代表需要加密的数据，密钥 K 则表示加密过程中使用的密钥。对于对称加密算法来说，加密和解密使用的密钥是相同的，其值由加密方和接收方共同协定，因此，当密钥被破解时，攻击者就能通过密钥解密发送方发送的数据，窃取用户的隐私。

密文 C 表示明文 P 经过加密后输出的值，也就是在网络中传输的数据。注意明文必须为 128 位，密钥的长度则可为 128 位、192 位或 256 位，在加密过程中，当数据长度不足时，将会进行补齐操作。

AES 加密操作由很多轮变换组成，其中包括字节替换（通过 S 盒做分组字节的转换）、行移位（转置操作）、列混淆（有限域加法和乘法）和轮密钥加（中间结果与分组做按位异或操作），需要注意的是最后一轮没有列混淆操作。

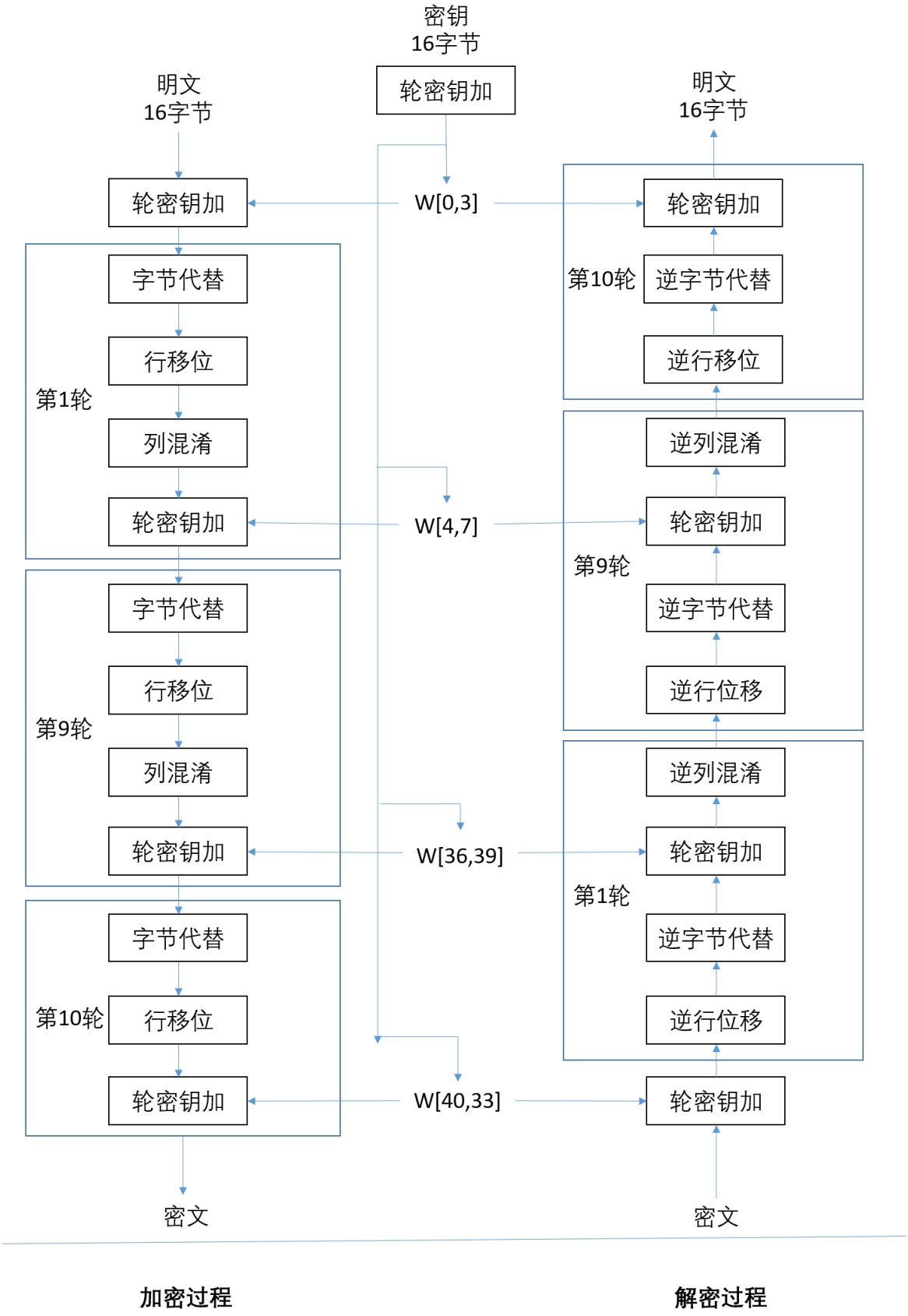

图 8 AES 加密解密过程

图 8 为 AES 加密解密过程的流程图，待加密的数据组合为 16 字节的明文后，与扩展密钥进行轮密钥加之后进过总共 10 轮的变换，每轮变换包含字节替换、行移位、列混淆以及轮密钥加操作，但最后一轮变换中不包括列混淆变换，最后生成 16 字节的密文，也就是加密之后的数据。解密过程就是相应加密过程的逆操作，通过密钥和密文进行轮密钥加操作后同样执行 10 轮混淆、移位等操作，最后恢复出明文。

在 Intel x86 平台上，以及有很多针对 AES 加密算法的 Cache 攻击实验，Spreitzer [35]实现了针对 AES 的 Cache 攻击，Acıiçmez[1][2]研究针对 AES 的远程攻击方式，并在在微处理器下的实现针对 AES 的 Cache 攻击，Bogdanov[4]研究了在嵌入式 CPU 中几种不同的的针对 AES 的 Cache 冲突时间攻击方式，Bonneau[5]研究了针对 AES 的 Cache 冲突时间攻击，并成功的恢复了 AES 密钥，Gallais[8]提升了针对 AES 的 Cache 攻击的性能，Gullasch [13]则研究了针对 AES 的 Cache 攻击实践，并成功的获取 AES 全部密钥。此后，Gülmezoğlu [14]实现了针对 AES 的快速的基于 Flush+Reload 方法的 Cache 攻击，Irazoqui [17]则在多核的情况下实验了跨多核的攻击针对 AES 的 Cache 攻击，Weiß [39]则在虚拟环境下实现了针对 AES 的攻击。可见在 Intel x86 平台上，Cache 攻击针对 AES 加密技术的威胁性挺大的。

2.4 Cache 攻击相关工作

1992 年，Hu[15]提出了在内部传输的上下文中 Cache 状态能够导致跨处理器的信息泄露的观点。Hu 介绍了格调度器，其是一个进程调度器，通过使用访问类属性调度进程来降低某些隐式通道的性能损失。它是作为 VAX 安全内核的隐蔽通道分析的一部分而开发的。VAX 安全内核是 VAX 体系结构的虚拟机监视器安全内核，旨在满足美国国家计算机安全中心 A1 级的要求。在描述缓存通道之后，以 VAX 安全内核为例，说明如何利用这个通道。并讨论了如何关闭此频道以及关闭频道的效果，演示了它在关闭缓存通道中的用法。最后，该工作通过一个扩展的例子来说明格子调度器的操作，最后讨论基本调度算法的一些变化。该文章也指出了 Cache 潜在的漏洞，为 Cache 攻击理论发展提供了基础。

Cache 攻击是一种针对缓存主存架构的攻击方式，其最初是由 Kelsey et al.[20]在 1998 年提出来的，其主要利用系统在读取数据时的缓存命中和缓存缺失之间的差异，并

据此获取缓存泄露出的信息，提出了将其用于密码分析的思想。在该论文中的概念是在Kocher的基础上提出的，文章中首先提出了单通道加密的概念，及使用以及实现的数据来进行加密的方式。之后提出了单通道攻击的概念，以及其导致的攻击漏洞，并阐述了真对三种加密方法的思想：针对IDEA的计时攻击、针对RC5的处理器标识攻击、正对DES的平均权重攻击。这些想法是Cache攻击具有里程碑意义的思想，通过利用Cache命中与未命中的时间差异，以及内存与缓存之间的映射关系，来获取用户的私密信息。随后，在Kelsey提出思想的基础上，有众多学者提出改进方案，并成功将该思想应用到现实的攻击工程中。

1999年，Koeune[21]描述了针对一个不健全实现的AES加密算法的Cache攻击，该AES实现中在使用代数运算时使用了粗心的方式，及在AES加密过程中使用了条件分支。除了经典的加密算法之外，也有的研究集中在针对密码系统物理实现的攻击上。内存信息的泄露在其他上下文环境下也被讨论过，加密算法的设计者们详细阐述了基础数学和加密策略的基本设计思想，并对微分和线性密码分析的基础进行了改进，并讨论了实现和优化问题。Koeune et al.研究了针对特定的密码系统的实现的攻击，其比他们的“古典”相对应的实现更有效率,在某种意义上,实施所需的资源通常是小得多。他们还描述了对 AES 候选 Rijndael 的定时攻击。展示了一个不小心的实现是如何通过每个关键字节的数千个度量来打破的，这提供了一个非常有限的实现知识。

在Kelsey[20]提出的通过Cache泄露的时间信息获取加密算法密钥的基础上，2002年，Page et al.[32]针对DES加密算法进行攻击，利用了DES加密算法在加密过程中查表索引与明文和密钥之间关系，并结合此时Cache中各个Set命中与缺失的情况，首次在仿真的环境下实现了针对 DES 加密算法的攻击，成功的缩小了在破解密钥时猜测的空间范围，将其从2^{56}降低到了2^{32}。然而，由于降低后的密钥空间仍然很大，通过暴力法依然无法破解，因此在现实破解过程中并没有太大的意义。但即使如此，Page也是第一个将Cache攻击思想应用到实际加密算法的研究者。

2004年，Bernstein[3]首次通过Cache攻击从本地计算机对在服务器端运行OpenSSL开源 AES 加密算法进行攻击，并成功的获取了服务器的密钥。其演示了对另外一台电脑上的网络服务器的AES密钥的恢复工作，给出漏洞归咎于AES加密算法的设计，而不是网络服务器使用的具体 AES 库。由于为普通计算机编写常量时间加密算法是极为

困难的，因此 AES 加密算法的漏洞广泛存在。文章中详细介绍了几种缺陷。文章中成功实现了一个非常简单的定时攻击。相同的技术也许可以从更复杂的服务器中提取完整的 AES 键，这些服务器实际上是用来处理 Internet 数据的，尽管攻击通常需要额外的时间来计算变量网络延迟的影响。且 Bernstein 传输这种类型的攻击不仅限于奔腾 III。其对不同的芯片进行测试，包括 AMD Athlon、英特尔奔腾 III、英特尔奔腾 M、IBM PowerPC RS64 四世和太阳 UltraSPARC III，都显示类似水平的 OpenSSL AES 计时可变性。因此可以对运行在所有这些 cpu 上的软件进行类似的攻击。文章选择了 Pentium III 是因为 Pentium III 是当时 Internet 服务器中最常见的 cpu 之一。由于在此之前在 Intel x86 平台上进行的针对 AES 等加密算法的攻击均是在本地端获取的，Bernstein 也开创了通过 Cache 计时攻击针对服务器端加密算法进行攻击的先河。

2005 年，Percival[33]描述了在多线程并发执行情况下针对 RSA 加密算法的 Cache 攻击。多线程并发执行及在多个执行线程之间共享一个超级标量处理器的执行资源，被广泛应用到 Intel Pentium 4 处理器中。在此实现中，由于处理器区域的效率和经济性，线程之间的处理器资源共享超出了执行单元。然而也因此引入了一些问题，其中令人担心的是线程共享对内存缓存的访问。Percival 演示了对内存缓存的共享访问不仅可以在线程之间轻松地使用高带宽的隐蔽通道，而且还允许恶意线程(理论上使用有限的权限)监视另一个线程的执行，因此会导致在许多情况下密钥被盗取。最后，Percival 对处理器设计者、操作系统供应商以及加密软件的作者提出了一些建议，阐述如何减轻或消除这种攻击漏洞。

2013 年，Yarom et al.[40]通过使用 Flash+Reload 的攻击模式对 RSA 加密算法进行攻击，成功的获取了 RSA 密钥中的大部分字节信息。在非信任进程之间共享内存页是减少多用户系统内存占用的常用方法。其演示了由于 Intel X86 处理器的漏洞，内存页面共享暴露了信息泄漏的过程。Yarom 使用了 Flash+Reload 方式，这是一种缓存侧向通道攻击技术，它利用这种漏洞来监视共享页面中的内存行。与之前的缓存侧向通道攻击不同，FLUSH+RELOAD 目标是最后一级缓存(即具有三个缓存级别的处理器上的 L3)。因此，攻击程序和受害者不需要共享执行核心。通过使用它从一个运行 gnupg1.4.13 的被攻击程序中提取私有加密密钥来演示 FLUSH+RELOAD 攻击的效果。并测试了在单个操作系统中两个不相关的进程之间的攻击，以及在单独的虚拟机中运行的进程之间的攻

击。平均而言，攻击可以通过观察单个签名或解密来恢复96.7%的密钥字节。该方法相对之前的攻击方式精度更高，且能够通过较少次数的分析就能获取攻击结果，Flush+Reload作为新的攻击思路，为之后的攻击方式提供了便利。

2016年，Moritz et al.[23]提出在移动设备上进行跨核Cache攻击的方式，该方式在攻击的过程中不需要获取系统权限。Moritz针对ARM设备处理器与Intel x86处理器在Cache结构以及Cache替换策略等的特点，提出了针对常用攻击方式Prime+Probe、Flush+Reload、Evict+Reload以及Flush+Flush在使用ARM处理器的移动设备端进行攻击的改进建议，并在移动设备上实现了多程序的跨信道通信。通过创建模板矩阵的方式，对用于输入的共享库进行监测，实现了监听用户键盘输入，监测用户点击或长按屏幕的功能。

2.5 本章小结

本章主要介绍Cache攻击的相关的技术，包括CPU Cache存在的漏洞，以及本文攻击的主要目标AES高级加密算法。CPU Cache漏洞方面详细介绍了Cache的设计，Cache与主存的映射关系，这是实现Cache攻击的基础，由于操作系统在访问内存时会将内存块映射到容量相对小很多的cache的不同区域，才使得通过cache获取用户私密信息成为可能。之后介绍了Cache的替换策略，其是实现Cache攻击的关键，也是实现有效攻击的基础，以及Cache中用于保证数据一致性的Tag和Index技术。随后讨论了Cache的包含性，能够帮助更好的理解Cache攻击的过程。最后本章介绍了两种高效的Cache攻击模式，Evict+Time和Prime+Probe，他们在Intel x86平台被证明是有效的攻击方式，通过一些改进，本文也成功的基于Prime+Probe实现了在移动设备上的Cache攻击，并成功的恢复了AES全部密钥。

第三章 计时方式及驱逐策略

在过去的几十年里，得益于摩尔定律CPU的运算速度以每年大概60%的速度提升，然而内存的读写速度每年的提升幅度在 7-9%之间，使得本来就慢得多的内存越来越跟不上 CPU 的速度，到现在两者之间已经有了很大的差距，对现在的计算机处理器，访问在一级缓存中的数据只需要0.3ns，而访问在内存中的数据需要50到150ns的时间，大约降低了 2 到 3 个数量级。为了解决CPU与内存的速度不匹配问题，现代计算机及移动设备在 CPU 与内存间放置多级缓存，在缓存命中的情况下大幅降低访存时间。但也引入了一些问题，及缓存缺失时会导致更长的访问时间。因此，Cache 被设计为特定的访问模式以降低cache的miss几率，如全相联映射、组相联映射等，针对特定的cache模式，根据内存与cache的映射关系能够轻易的对其进行攻击，获取被攻击程序执行时的缓存信息。

现代处理器使用一级或多级缓存的组相联缓存，从上到下内存容量逐渐增加，访问速度逐渐下降，其中每一级由S个cache set，每个cache set中包含W个cache line，每个cache line能够容纳B字节的数据，因此cache的总容量为B×W×S个字节。其中cache line是cache缓存的基本单位，当cpu读取的数据不在缓存中时，系统产生一个中断，并从内存中获取一个内存块的数据缓存到cache中，再从cache中读取数据到cpu，其中内存块 block 的大小与 cache line 的大小相同。对采用组相联映射的 cache，一个cache set包含W个cache line，当cache miss发生时，从内存中获取B字节的内存块缓存到cache中，set索引号为内存块起始地址a mod [W × B]，并根据cache使用的替换策略替换掉（驱逐）set中指定的cache line。常见的替换策略由最近最少使用算法 LRU、随机替换算法等，若使用LRU算法，则cache set中最近没有被访问过得cache line被驱逐到内存中去。

现代处理器有多到3级的缓存及L1到L3，其中L1的访问速度最快，容量最小，当L1查找失败时到L2查找，相应的L2的访问速度较L1要慢，容量比L1要高，L3同理，但L1到L3的访问速度比访问内存速度要快很多。为了简化攻击过程，本文不区分访问L1和访问L2、L3的区别，只区分cache命中和cache缺失。

3.1 获取精确计时方式

精确的计时方式是 Cache 攻击的前提，它需要将访存和访问缓存有效的区分开来，供攻击者获取待攻击程序运行导致的 cache set 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的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.[23]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，但这些接口并非在非特权模式下对所有 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。

虽然通过读取相应的计时寄存器能够获得最精确的计时时间，但其必须要在特权模式下才能访问，因此不具有普适性。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。包括：

3.1.1 clock_gettime 系统调用

"clock_gettime"是 Linux 提供的计时时间函数，计时粒度能够精确到纳秒。使用时需要啊包含 time.h 头文件。该函数声明为“int clock_(clockid_t id， struct timespec *tp)”，形式参数 id 代表获取的系统时钟时间类型。其有多种时间类型，如果将 id 设置为 CLOCK_REALTIME，则表示统计的时间为系统的实际时间，即从 UTC1970-1-1 0:0:0 开始到现在经历的时间，如果将 id 设置为 CLOCK_MONOTONIC，则代表从系统启动开始，到当前经历的时间，改时间不受用户修改对时间造成的影响，而如果将始终类型修改为 CLOCK_PROCESS_CPUTIME_ID，则表示当前进程运行到计时函数所花费的 CPU 时间，如果将计时类型设置为 CLOCK_THREAD_CPUTIME_ID，则表示统计当前线程运行到当前代码系统所花费的 CPU 时间。最终统计得到的时间记录到 timespec 结构中，其中有两个变量，其中 tv_sec 表示计时的秒数，而 tv_nsec 表示计时结果的纳秒数。

表 1 POSIX 接口获取系统时间

```
1.    #include <time.h>
2.    int64 get _time(void)
3.    {
4.       struct timespec t;
5.       clock_gettime(CLOCK_MONOTONIC, &t );
6.       return t.tv_sec * 1000*1000*1000ULL + t1.tv_nsec;
7.    }
```

表 1 POSIX 接口获取系统时间展示了如何通过 POSIX 接口获取系统时间，在引入头文件之后，调用clock_gettime函数，即可将当前系统时间记录到制定内存地址的数据结构中。

3.1.2 Perf 性能分析工具

Perf工具是 Linux 提供的一种性能分析工具。其主要基于对事件的采样，提供对于处理器的性能以与操作系统相关性能进行性能剖析的功能。因此可以用于对系统性能进行分析。

从 Linux Kernel 2.6.31 版本开始，Linux 内核提供了__NR_perf_event_open 系统调用。该系统调用向程序员提供了一种方式，通过该方式像打开文件一样打开一个性能计数器，通过设置不同的参数就可以让该性能技术器统计不同的系统事件，其中保罗获取系统的时间，并且可以简单的通过读取文件来获取统计结果。

表 2 Perf 获取系统时间

```
1.    void perf_init(int* fd)
2.    {
3.        static struct perf_event_attr attr;
4.        attr.type = PERF_TYPE_HARDWARE;
5.        attr.config = PERF_COUNT_HW_CPU_CYCLES;
6.        attr.size = sizeof(attr);
7.        attr.exclude_kernel = 1;
8.        attr.exclude_hv = 1;
9.        attr.exclude_callchain_kernel = 1;
11.       fd = syscall(__NR_perf_event_open, &attr, 0, -1, -1, 0);
13.       return true;
14.   }
15.   int64 perf_timing( int* fd)
16.   {
17.       long long result = 0;
18.       if (read(fd, &result, sizeof(result)) < (ssize_t) sizeof(result)) {
19.           return 0;
20.       }
21.       return result;
22.   }
```

在使用**表 2 Perf 获取系统时间**获取系统时间之前，需要引入linux/perf_event.h、sys/syscall.h以及unistd.h头文件。并在获取系统时间之前首先格式化获取的时间属性，比如上表将记录的时间设为PERF_COUNT_HW_CPU_CYCLES，及获取系统的 cpu 周期。

3.1.3 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并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。因为一次计数器的加一操作可看做是

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。

表 3 Thread 模拟获取系统时间

```
1.  long long counter;
2.  Thread count_thread:
3.      bind_to_cpu(1);
4.      while (true) {
5.          counter++;
6.      }
7.  int64 thread_timing()
8.  {
9.      memory_barrier();
11.     int64 time = counter;
13.     memory_barrier();
14.     return time;
15. }
```

表 3 Thread 模拟获取系统时间展示了通过线程模拟获取系统时间的伪代码，首先定义一个全局变量counter记录系统当前时间，之后通过启动一个线程对该计数器进行累加操作，需要注意的事计时线程需要绑定到一个固定的 CPU 上，以防止核间漂移对计时结果造成影响，且该 CPU 最好只进行计时操作，这样可以假设每次累加操作花费固定的时钟周期 m。因此counter计数器的数值反应该时刻距线程启动时刻的时间$t = counter * m$。对于只需统计时间区间的 Probe 函数来说，Thread 模拟计数方式返回的结果已经足够了。

3.2 获取高效驱逐策略

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.[12]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$a_1 a_2 a_3$，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。
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$a_1 a_2 \ldots a_{17}$到$a_1 a_1 a_2 a_2 \ldots a_{17} a_{17}$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

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:      for i = 0; i < N – D; i++ do
2:          for j = 0; j < A; j++ do
3:              for k = 0; k < D; k++ do
4:                  Access(i + k)访问第 i + k 个相关地址
5:              end for
6:          end for
7:      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。

3.3 本章小结

本章介绍实现 Cache 攻击的预备工作，首先呈现 3 种不需要系统额外权限就能获取系统时间的方式，及 POSIX 提供的 clock_gettime 接口、linux 内核提供的 perf 性能分析工具以及可以模拟系统计时的线程计数器，并介绍了 3 中时间源的调用方式，最后介绍如何获取高效的 Cache 驱逐策略。

第四章 攻击方案的设计

虽然最近几年有人设计并实现了针对 Intel x86 的 AES 攻击实验，并能够获取出部分甚至全部的密钥。然而由于移动平台的 Cache 结构、指令集、Cache 替换策略等与 Intel x86 平台上的设备有较大的不同，不能直接将 Intel x86 平台上的 Cache 攻击方式直接使用到移动平台上。当前基于 Android 系统的 ARM 指令集设备的 L2 Cache 通常使用伪随机替换算法，和 LRU 替换算法不同，伪随机替换算法会替换最近使用过的数据，因此，即使即使方式十分精确，也会在 Probe 度量阶段引入误差，且这种误差是不可避免的。并且通常使用时间源计时也会由于偶然因素引入误差，更增加了 Probe 获取时间的不可靠性。本文针对移动设备硬件结构可能引入的误差，引入柯尔莫哥洛夫-斯米尔诺夫检验(Kolmogorov-Smirnov test，下文称 KS 检验)获取更准确可靠性更高的 Probe 度量分，并将其引入到攻击方案中。

4.1 AES 攻击设计

在能够获取精确的系统时间来对驱逐操作和探测操作计时以及快速高效可靠的驱逐方式的基础上，还需考虑攻击程序对待攻击程序的攻击交互方式。本文主要针对 AES 加密算法，通过同步攻击的方式验证 Cache 攻击的有效性。

同步攻击可理解为已知数据的攻击，攻击程序和被攻击程序之间有交互操作发生，攻击程序能够触发被攻击程序的运行，在被攻击程序执行之前和之后可以执行指定的代码，且能够获取到被攻击程序的输入和输出。对于 AES 加密程序而言，对其进行同步攻击则表示攻击者能够获取到加密的输入及明文，也能够控制 AES 加密程序的执行。

4.1.1 AES 查表索引

在对 AES 加密算法进行攻击之前，需要首先了解 AES 加密数据的过程，特别是加密过程中内存的访问情况。在第二章中介绍过 AES 加密的流程，首先将待加密的数据划分为若干块，每块的大小均为 128 位（16 字节），它们是加密数据的基本单位，通常将其称之为明文，记为$\mathrm{p} = (p_0, p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8, p_9, p_{10}, p_{11}, p_{12}, p_{13}, p_{14}, p_{15})$。相应的密钥记为$\mathrm{k} = (k_0, k_1, k_2, k_3, k_4, k_5, k_6, k_7, k_8, k_9, k_{10}, k_{11}, k_{12}, k_{13}, k_{14}, k_{15})$，密钥是加密算法最重要的部分，所有数据均由密钥 k 加密得来，解密过程也需要密钥 k，当密钥

被攻击者获取时，用户的数据就毫无秘密可言，因此保护用户密钥是尤为重要的。第二章已经描述过，AES 加密过程总共分为 10 轮，每轮需要将密钥 k 和明文 p 做轮密钥加、字节替换、行移位、列混淆操作。可以基于硬件或软件这些操作，基于硬件方式实现的 AES 加密算法加密速度快，但设计复杂，成本较高。而基于软件方式实现的算法加密速度稍慢，但成本低，且在不断的优化后加密时间与硬件方式实现相差无几。

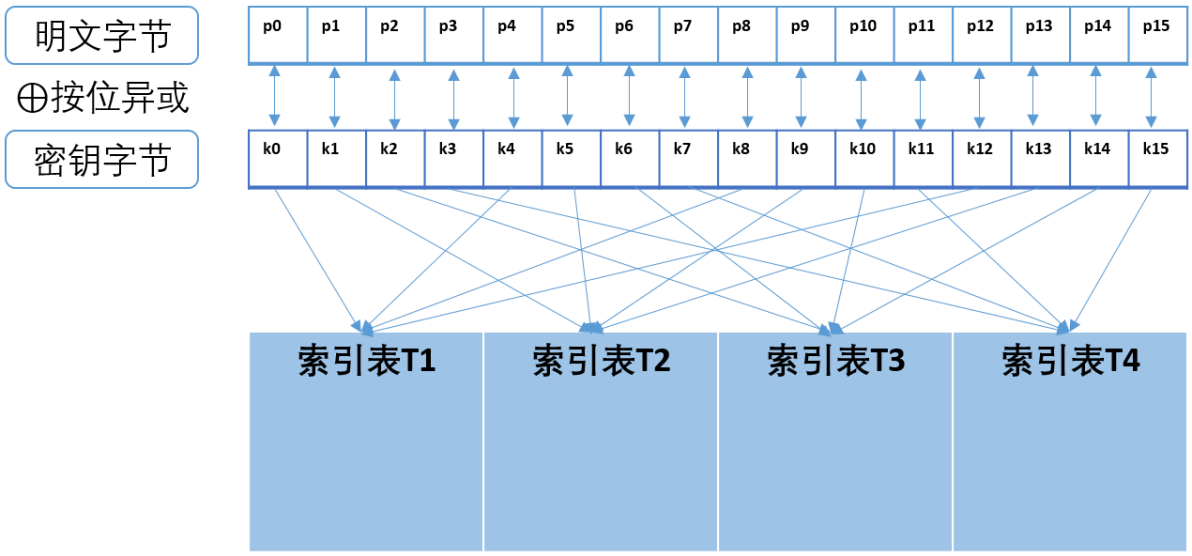

图 9 第一轮查表索引

本文主要针对软件方式实现的 AES 加密算法，其是通过事先计算好的矩阵索引操作来实现轮密钥加、字节替换、行移位以及列混淆操作的，主要有 8 个这样的查表矩阵包括 1 到 9 轮的查表矩阵 T1、T2、T3、T4，以及最后一轮的查表矩阵$T_0^{(10)}$、$T_1^{(10)}$、$T_2^{(10)}$、$T_3^{(10)}$，其中每个表包含 256 个数据项，每个数据项为 4 字节，关于矩阵值如何计算得到的不在本文的讨论范围之内。其中需要了解的是 10 轮加密过程中需要不断的查表，及访问 T1~T4 或$T_0^{(10)}$、$T_1^{(10)}$、$T_2^{(10)}$、$T_3^{(10)}$表中的部分数据，查表索引可由明文 p 和密钥 k 计算得来，其中第一轮查表索引比较特殊，为i = p ⊕ k，记为$i_n = p_n \oplus k_n$，其中 n∈0~15，查表索引如**图 9 第一轮查表索引**所示。为了表示方便，将 AES 加密过程中生成的 10 轮中间密钥记为$K^{(r)}$其中 r=1,…,10，每个中间密钥$K^{(r)}$又可拆分为 4 个字，每字为 4 个字节，记为$K^{(r)} = (K_0^{(r)}, K_1^{(r)}, K_2^{(r)}, K_3^{(r)})$，则第 0 轮的密钥记为原始密钥 k，记$K_j^{(0)} = (k_{4j}, k_{4j+1}, k_{4j+2}, k_{4j+3})$，其中 j=0,…,3。

当向 AES 加密程序提供 16 字节的明文$p = (p_0, \ldots, p_{15})$时，加密过程中的每一轮均

需要计算一个 16 字节的中间状态$x^{(r)} = (x_0^{(r)}, \ldots., x_{15}^{(r)})$。其中 x 的初试值$x^{(0)} = p \oplus \mathrm{k}$，对中间状态的每一个字节$x_i^{(0)} = p_i \oplus k_i$，其中 i=0,…,15。而加密过程中前 9 轮变化的中间状态可由如下公式计算得到，r 表示轮数，取值从 0 到 8，如**表 4 AES 加密过程中中间状态**所示。

表 4 AES 加密过程中中间状态

$$(x_0^{(r+1)}, x_1^{(r+1)}, x_2^{(r+1)}, x_3^{(r+1)}) \leftarrow T_0[x_0^{(r)}] \oplus T_1[x_5^{(r)}] \oplus T_2[x_{10}^{(r)}] \oplus T_3[x_{15}^{(r)}] \oplus K_0^{(r+1)}$$

$$(x_4^{(r+1)}, x_5^{(r+1)}, x_6^{(r+1)}, x_7^{(r+1)}) \leftarrow T_0[x_4^{(r)}] \oplus T_1[x_9^{(r)}] \oplus T_2[x_{14}^{(r)}] \oplus T_3[x_3^{(r)}] \oplus K_1^{(r+1)}$$

$$(x_8^{(r+1)}, x_9^{(r+1)}, x_{10}^{(r+1)}, x_{11}^{(r+1)}) \leftarrow T_0[x_8^{(r)}] \oplus T_1[x_{13}^{(r)}] \oplus T_2[x_2^{(r)}] \oplus T_3[x_7^{(r)}] \oplus K_2^{(r+1)}$$

$$(x_8^{(r+1)}, x_9^{(r+1)}, x_{10}^{(r+1)}, x_{11}^{(r+1)}) \leftarrow T_0[x_8^{(r)}] \oplus T_1[x_{13}^{(r)}] \oplus T_2[x_2^{(r)}] \oplus T_3[x_7^{(r)}] \oplus K_2^{(r+1)}$$

至于最后一轮计算中间状态，只需要将 T1、T2、T3、T4 替换为$T_0^{(10)}$、$T_1^{(10)}$、$T_2^{(10)}$、$T_3^{(10)}$即可，所得的$x^{(10)}$就是明文通过 AES 加密得到的密文。如上公式中的$T_i[index]$表示 AES Table i 中的第 index 项数据，数据占 4 字节，‘⊕’表示对操作数的每一位执行异或操作。

4.1.2 AES 攻击思路

AES 执行过程中对各个 Table 的查表操作是对其进行 Cache 攻击的切入点，对于相同的密钥 k，对于不同的明文 p 进行加密时，根据如上的计算，会访问到各个 Table 中不同索引位置，如果 Cache 中没有缓存相应位置的数据时，会将该数据以及其周围的数据加载到其内存所对应的 cache set 中的某一个 line 中，加载的数据块大小与 cache line 大小相等，在本文讨论的 Lenovo k51c78 被测机中，Cache line 为 64bytes，Table 中的每项数据为 4bytes，每次访问 Table 中的数据时，会将其周围共 16 个数据项加载到同一个 cache line 中。假设 T0 中起始数据所在内存对应 Cache 中的 set a 的索引为 0 的数据项，则 T0 中索引前 16 项的数据都将映射到 set a 中，而索引第 16 至第 32 项数据将映射到 set a+1 中，Table 中索引索引为 i 的数据将映射到的 set 索引为 a+(i/16)。攻击程序通过对自身内存的访问，能够获取到其他程序使用 Cache 的情况，通过第二章介绍的 Prime+Probe 攻击方式，攻击程序首先通过 Prime 方法读取其内存空间的数据占用 cache

中指定数个 set 中的所有 line，然后调用 AES 加密程序，待加密程序执行完毕之后，执行 Probe 方法，根据访问时间判断 Prime 阶段读取的数据是否还保存在 Cache 中，若均在 Cache 中，则表示 AES 加密过程中的多轮变化的查表操作没有访问到能够映射到该 set 的数据。反之，若 Probe 阶段探测到之前读取的数据不在 cache 中，则表明 AES 加密过程中的某一轮查表操作或其他访存操作访问到了能够映射到该 set 的数据，为了简化攻击操作，假设 AES 加密过程中除了查表操作外不存在其他访存操作。因此，假设已知了 Table 中各个索引表的起始地址对应的 cache set 索引，当攻击程序同时对 cache 中的所有 set 进行监控时，就能够监测到 AES 加密过程中访问了T_0、T_1、T_2、T_3、$T_0^{(10)}$、$T_1^{(10)}$、$T_2^{(10)}$、$T_3^{(10)}$中哪些区域的数据。

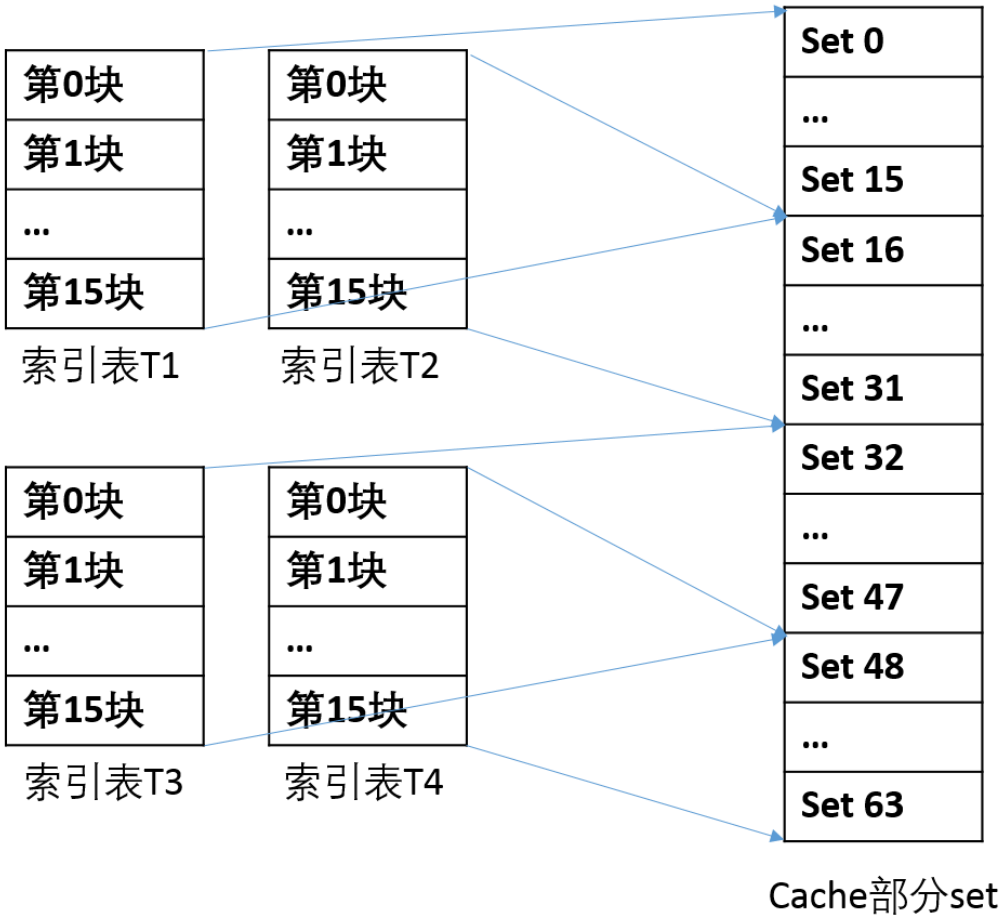

图 10 索引 Table 与 Cache 映射关系

为了方便讨论，本文将以目标机 Lenovo k51c78 的配置为例，讨论以其作为被攻击设备，设计针对 AES 的攻击的方案。该目标机的 L2 cache 大小为 512KB，共有 512 个 set，组织为 16 路组相联，因此每个 set 共有 16 个 cache line，每个 cache line 能缓存 64

字节的数据。被攻击的 AES 程序网上的开源版，其索引表为T_0、T_1、T_2、T_3、$T_0^{(10)}$、$T_1^{(10)}$、$T_2^{(10)}$、$T_3^{(10)}$，每个索引表包含 256 个数据项，可将其视为有 256 个元素的数据，其中每个数据项为 4bytes，并且假设已知索引表的首地址 a 对应的 cache set 索引 i，为了方便讨论，假设 a 的取值满足$a\%64 == 0$，及索引 table 的首地址映射到 cache line 的起始位置。则可以求得索引表中任意索引位置 n 对应的 cache set 位置，其中对应表示当该位置数据被访问时其将被保存的 cache set 号。由于攻击程序通过 Prime+Probe 只能探测到被攻击程序访问 cache set 的情况，并不能区分访问了 cache set 中哪些 line，更不能获取 cache line 中偏移的情况，因此通过 Prime+Probe 攻击程序并不能区分对索引表T0中索引 0 到索引 15 数据项的访问（对 0 到 15 项的任意一项的访问都将一整块的数据缓存到了 cache 中）。

本文设计通过假设验证的方式来实现获取 AES 密钥，其破解过程分为两轮，第一轮获取密钥中每个字节的前 4 位，称为第一轮攻击，第二轮获取密钥中每个字节的后四位，称为第二轮攻击。

4.1.3 第一轮攻击

第一轮攻击利用 AES 加密执行中第一轮对索引表的访问情况来进行攻击的。对于已知的明文 p 和密钥 k，第一轮的查表索引为$n = p \oplus k$，对应每个字节为$n_i = p_i \oplus k_i$，其中$i = 0, \dots, 15$。此处需要注意的是，并非 16 个查表索引均需要到T_0、T_1、T_2、T_3、$T_0^{(10)}$、$T_1^{(10)}$、$T_2^{(10)}$、$T_3^{(10)}$中查找相应位置的数据，其对应关系为：n_i对应的是T_l或$T_l^{(10)}$中的查表索引，其中$l = i \bmod 4$，及n_0是T_0的索引，而n_1是T_1的查表索引。

本文使用假设验证的方式对 AES 加密算法进行攻击，攻击步骤主要分为两个部分，第一部分及假设部分，及假设密钥 k 的取值$\hat{k}$，由于明文 p 对攻击程序来说是已知的，通过$\hat{n} = p \oplus \hat{k}$则可计算出$\hat{k}$对应的第一轮加密过程的查表索引，其中$\hat{n}_0, \hat{n}_4, \hat{n}_8, \hat{n}_{12}$表示$T_0$表的索引，$\hat{n}_1, \hat{n}_5, \hat{n}_9, \hat{n}_{13}$表示$T_1$表的索引，$\hat{n}_2, \hat{n}_6, \hat{n}_{10}, \hat{n}_{14}$表示$T_2$表的索引，$\hat{n}_3, \hat{n}_7, \hat{n}_{11}, \hat{n}_{15}$表示$T_3$表的索引。对于 AES 同步攻击来说$T_0$、$T_1$、$T_2$、$T_3$表的起始地址对攻击程序来说是可获取的，并且就像假设的那样，各个 Table 索引表的起始位置数据均能映射到某一 Cache set 中某一 line 的起始位置上，在本文中假设T_0、T_1、T_2、T_3的起始位置映射的 Cache set 索引分别为s_0、s_1、s_2、s_3，则可计算出在假设密钥为$\hat{k}$的情况下 AES 加密算

法在第一轮查表过程中访问到了T_0中索引位置为$\hat{n}_0, \hat{n}_4, \hat{n}_8, \hat{n}_{12}$的数据，之后这些数据分别被缓存在 cache 索引为$s_0 + \hat{n}_0/16$、$s_0 + \hat{n}_4/16$、$s_0 + \hat{n}_8/16$、$s_0 + \hat{n}_{12}/16$，访问到了T_1中索引位置为$\hat{n}_1, \hat{n}_5, \hat{n}_9, \hat{n}_{13}$的数据，之后这些数据分别被缓存在 cache 索引为$s_1 + \hat{n}_1/16$、$s_1 + \hat{n}_5/16$、$s_1 + \hat{n}_9/16$、$s_1 + \hat{n}_{13}/16$，访问到了T_2中索引位置为$\hat{n}_2, \hat{n}_6, \hat{n}_{10}, \hat{n}_{14}$的数据，之后这些数据分别被缓存在 cache 索引为$s_2 + \hat{n}_2/16$、$s_2 + \hat{n}_6/16$、$s_2 + \hat{n}_{10}/16$、$s_2 + \hat{n}_{14}/16$，访问到了T_3中索引位置为$\hat{n}_3, \hat{n}_7, \hat{n}_{11}, \hat{n}_{15}$的数据，之后这些数据分别被缓存在 cache 索引为$s_3 + \hat{n}_3/16$、$s_3 + \hat{n}_7/16$、$s_3 + \hat{n}_{11}/16$、$s_3 + \hat{n}_{15}/16$，对于已知的明文p和假设的密钥$\hat{k}$，能够计算出在第一轮变换中访问到了能够映射到 cache set 索引位置为$s_0 + (p_0 \oplus \hat{k}_0)/16$、$s_0 + (p_4 \oplus \hat{k}_4)/16$、$s_0 + (p_8 \oplus \hat{k}_8)/16$、$s_0 + (p_{12} \oplus \hat{k}_{12})/16$、$s_1 + (p_1 \oplus \hat{k}_1)/16$、$s_1 + (p_5 \oplus \hat{k}_5)/16$、$s_1 + (p_9 \oplus \hat{k}_9)/16$、$s_1 + (p_{13} \oplus \hat{k}_{13})/16$、$s_2 + (p_2 \oplus \hat{k}_2)/16$、$s_2 + (p_6 \oplus \hat{k}_6)/16$、$s_2 + (p_{10} \oplus \hat{k}_{10})/16$、$s_2 + (p_{14} \oplus \hat{k}_{14})/16$、$s_3 + (p_3 \oplus \hat{k}_3)/16$、$s_3 + (p_7 \oplus \hat{k}_7)/16$、$s_3 + (p_{11} \oplus \hat{k}_{11})/16$、$s_3 + (p_{15} \oplus \hat{k}_{15})/16$。

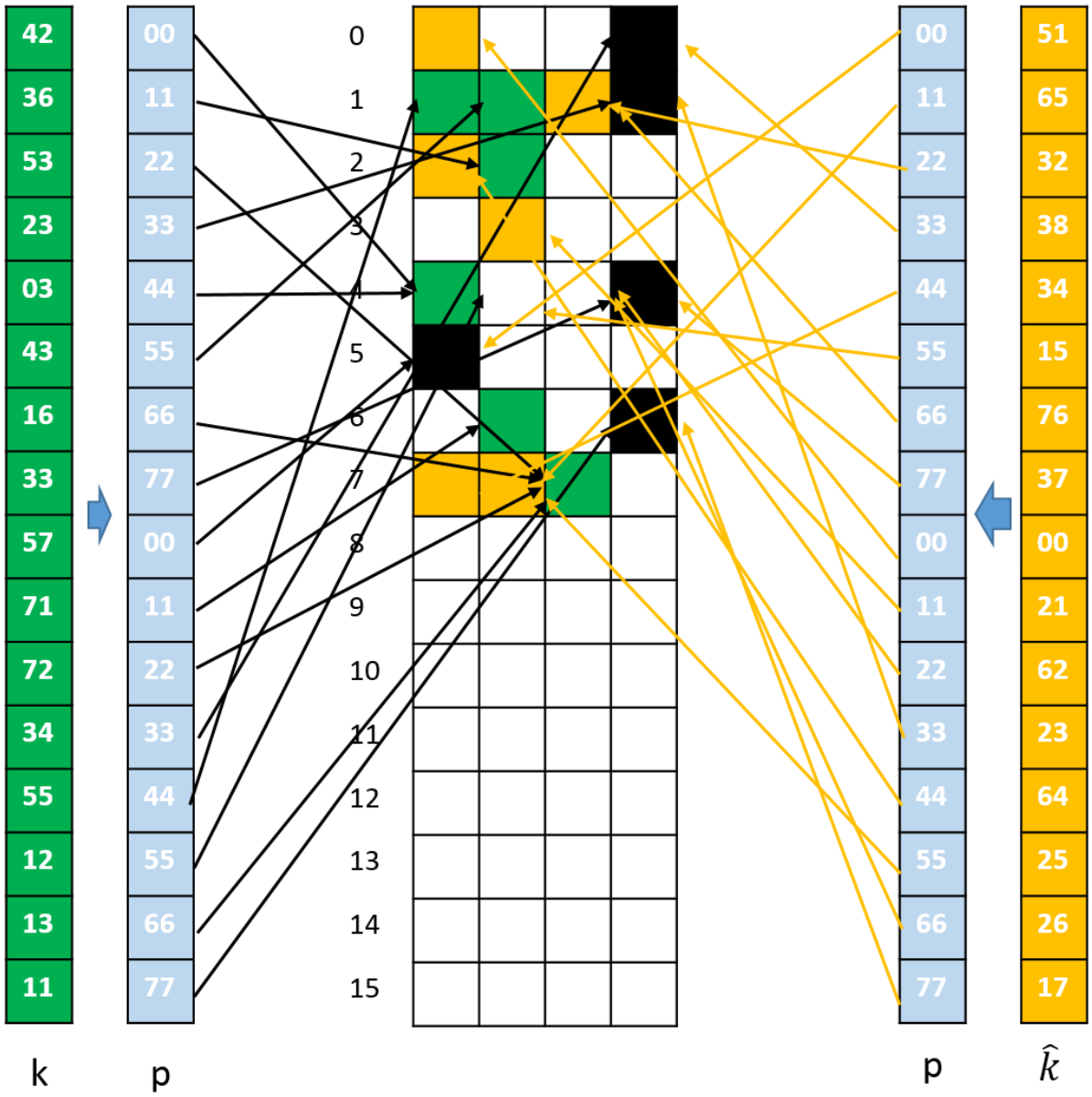

图 11 第一轮攻击访问 Cache 情况

图 11 第一轮攻击访问 Cache 情况描述了假设检验的原理，密钥 k 被设置为(0x42,0x36,0x53,0x23,0x03,0x43,0x16,0x33,0x57,0x71,0x72,0x34,0x55,0x12,0x13,0x11)，为了描述方便，假设 Table T1 首地址映射到 cache set 0 中某一 cache line 的首地址，且T_0、T_1、T_2、T_3地址空间相连，反应到图中为第二列的首地址紧接着第一列最后一行的末地址。每一个小方块代表一个内存块，映射到 cache 中对应一个 line。当对明文p = (0x00,0x11,0x22,0x33,0x44,0x55,0x66,0x77,0x00,0x11,0x22,0x33,0x44,0x55,0x66,0x77)进行验证时，访问到的 cache 块用橙色标注，绿色的 cache set 表示真实密钥对应到的 cache set，黑色为假设的密钥

k^=(0x51,0x65,0x32,0x38,0x34,0x15,0x76,0x37,0x00,0x21,0x62,0x23,0x64,0x25,0x26,0x17)和真实密钥 k 共同访问到的 cache set。因此，黑色的 cache set 越多表示假设的密钥$\hat{k}$与真实密钥 k 越接近。

因此，如果实验中假设的密钥$\hat{k}$值与真正的密钥 k 值相同，当攻击程序对这些 set 进行监控，执行 Probe 操作时将会探测到这些 cache set 在中的某些 line 被驱逐到了内存中，也就是被 AES 攻击程序访问到了，这会使得 Probe 阶段测得的时间相对较高，并将对这些 set 在 Probe 阶段所测得的时间相加并记为 t，作为该假设$\hat{k}$可疑度的度量分数 m，因此，度量分数 m 越高表示假设$\hat{k}$可疑度越高。由于 Table 中的 16 个数据项映射到同一个 set 中，因此实际决定查表数据映射到哪一个 set 中是由索引的前 4 位决定，后 4 位仅仅能决定数据缓存在 cache line 中的偏移，因此仅靠第一轮攻击仅仅能获取到每个密钥字节的前 4 位。为了获得密钥所有位的数据，需要进行进一步的分析。

4.1.4 第二轮攻击

仅仅利用 AES 加密过程中第一轮的查表索引以明文 p 和密钥 k 的代数关系只能获取 AES 密钥 k 中每个字节的前 4 位，仍有 16*4=64 位数据未知，如果使用暴力破解则仍有2^{64}种可能性，对现在的处理器性能来说量实在是太大了，因此需要考虑第二轮查表索引与明文 p 和密钥 k 的关系。

回顾上一节求解 AES 加密过程中 10 轮中间密钥的计算公式，不难推算出第二轮查表索引 n 与明文 p 和密钥 k 的关系如表**表 5 第二轮查表索引**所示。

表 5 第二轮查表索引

$$n_2 = s(p_0 \oplus k_0) \oplus s(p_5 \oplus k_5) \oplus 2 \bullet s(p_{10} \oplus k_{10}) \oplus 3 \bullet s(p_{15} \oplus k_{15}) \oplus s(k_{15}) \oplus k_2$$

$$n_5 = s(p_4 \oplus k_4) \oplus 2 \bullet s(p_9 \oplus k_9) \oplus 3 \bullet s(p_{14} \oplus k_{14}) \oplus s(p_3 \oplus k_3) \oplus s(k_{14}) \oplus k_1 \oplus k_{15}$$

$$n_8 = 2 \bullet s(p_8 \oplus k_8) \oplus 3 \bullet s(p_{13} \oplus k_{13}) \oplus s(p_2 \oplus k_2) \oplus s(p_7 \oplus k_7) \oplus s(k_{13}) \oplus k_0 \oplus k_4 \oplus k_8 \oplus 1$$

$$n_{15} = 3 \bullet s(p_{12} \oplus k_{12}) \oplus s(p_1 \oplus k_1) \oplus s(p_6 \oplus k_6) \oplus 2 \bullet s(p_{11} \oplus k_{11}) \oplus s(k_{12}) \oplus k_{15} \oplus k_3 \oplus k_7 \oplus k_{11}$$

其中⊕表示异或操作，$s(p_0 \oplus k_0)$表示对$p_0 \oplus k_0$的结果通过 sbox 函数处理后的结果，其中n_i表示索引 n 中的第 i 个字节，和上一节中的描述的一样，n_2是索引表T_2中的索引，

n_5是索引表T_1中的索引，n_8是索引表T_0中的索引，n_{15}是索引表T_3中的索引。

第二轮攻击与第一轮攻击相同，采用假设验证的方式，因此在已知明文为 p，假设密钥为$\hat{k}$的情况下，如果假设密钥为$\hat{k}$为真实的密钥 k，可以计算得到 AES 加密程序在第二轮查表过程中，至少访问了能够映射到 set 索引为$s_0 + \hat{n}_8$、$s_1 + \hat{n}_5$、$s_2 + \hat{n}_2$、$s_3 + \hat{n}_{15}$中的数据。在验证阶段，获取这 4 个 set 在 Probe 阶段的时间 t，将其相加得到假设密钥$\hat{k}$的可疑度，数值越高，表示在 AES 加密阶段中访问了能够映射到这几个 set 中的数据，当假设的密钥$\hat{k}$不是真正的密钥 k 时，通过计算可能会得到 AES 加密过程中并未访问过的 set，则对这些 set 进行 Probe 操作获取的时间 t 相对要少，因此会得到相对较小的度量分。

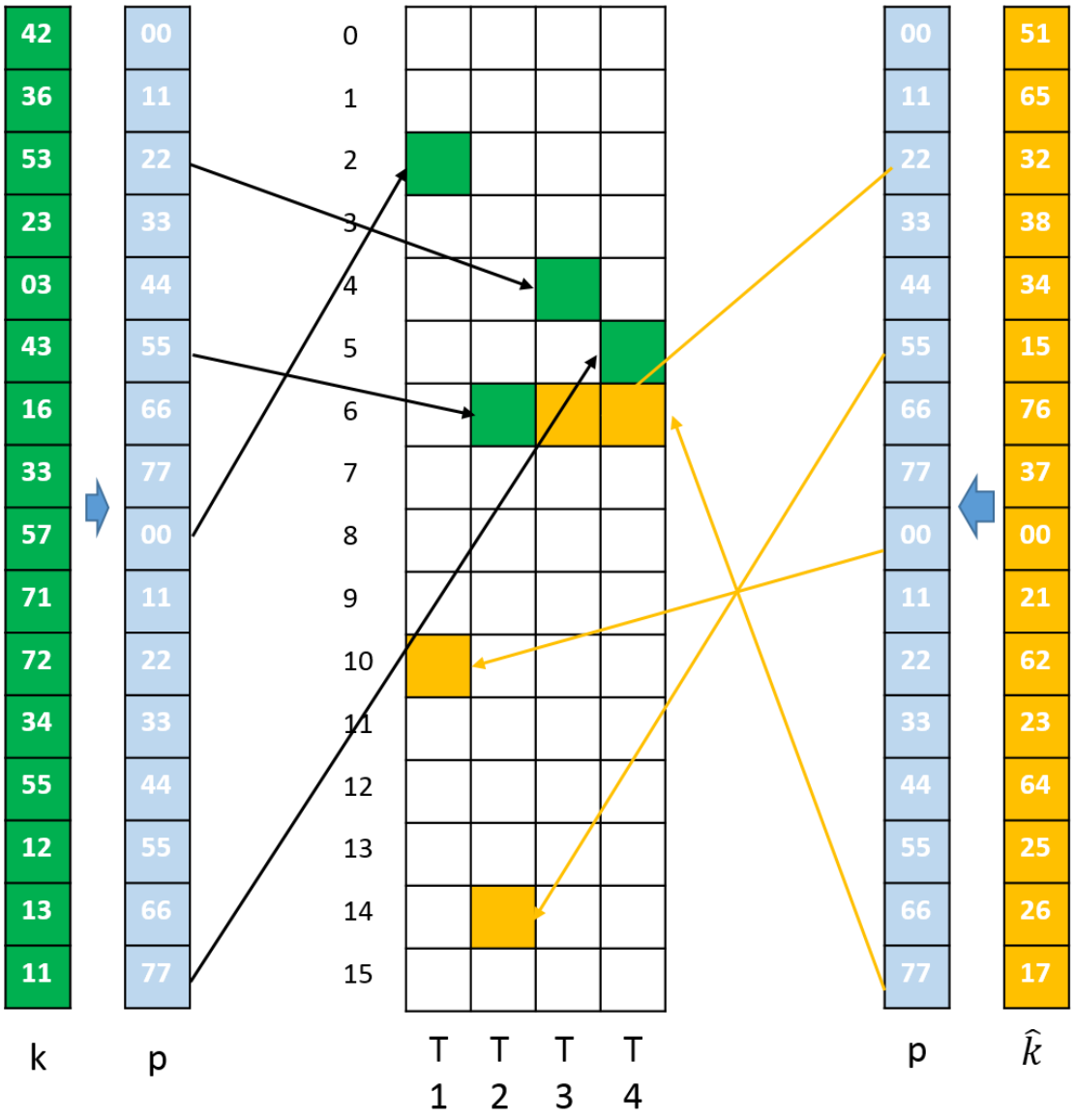

图 12 第二轮攻击访问 Cache 情况

图 12 第二轮攻击访问 Cache 情况表示假设过程中对 4 个 Table 中内存块的访问情况，与第一轮攻击类似，图中索引表中的每个小方块代表一个内存块，能够缓存到一个 cache line 中，T1 中的第一个内存块对应 cache 第一个 set。与**图 11 第一轮攻击访问 Cache 情况**相同，第二轮攻击分析过程中使用的密钥 k 被设置为(0x42,0x36,0x53,0x23,0x03,0x43,0x16,0x33,0x57,0x71,0x72,0x34,0x55,0x12,0x13,0x11)，在攻击过程中需要对密钥的所有取值空间进行假设，当明文的取值 p＝(0x00,0x11,0x22,0x33,0x44,0x55,0x66,0x77,0x00,0x11,0x22,0x33,0x44,0x55,0x66,0x77)时，计算得到对于真实的密钥 k，第二轮在表T_0、T_1、T_2、T_3上的查表索引分别为 0x49、0x65、0x2b、0x58，对应到图中橙色的方块表示，而对于假设的密钥取值 k≒(0x51,0x65,0x32,0x38,0x34,0x15,0x76,0x37,0x00,0x21,0x62,0x23,0x64,0x25,0x26,0x17)，计算得到在表T_0、T_1、T_2、T_3上的查表索引分别为 0x63、0xee、0xad、0x6d，对应到图中为绿色方块，可见当假设的密钥和真实密钥不同时，在 AES 加密第二轮查表过程中访问的内存块也不相同，通过攻击程序不断获取 Cache 中的各个 set 的访问情况，可探测到 AES 加密算法第二轮访问的情况，依此进行攻击，获取部分或全部的密钥值。

4.2 KS 检验

4.2.1 KS 检验概述

KS 检验是假设检验中的一种，它可以被用来判断随机抽样中样本与样本或总体与总体之间的关系是由于误差引起的还是由本质差别引起的。假设检验的原理是首先对总体做出某种假设，然后通过抽样的方法来判断是否接受该假设。

总体中个体的差异是普遍存在的，样本之间的差异也是普遍存在的，因此我们不能仅仅根据样本做出结论。在获取数据的过程中，如果多个抽样的平均值与假设总体的平均值不相同时，应当考虑两种不同的可能：一是这几个样本来自于总体的样本，均值不相同是由于抽样中产生的误差；二是这些样本来自于总体不同的分布，及其差别不仅仅是由于抽样中产生的误差产生的，而是抽样总体本身的属性与假设的总体分布不同产生的。

在本文的研究背景下，可以假设通过时间源获取的时间服从某一个正太分布，其平均值为测量的精确时间t，方差为σ，因此单次获取的时间会在一定的范围内波动，根据

总体的特性，有时候波动会很大，这时通过单次 Probe 获取的时间作为 set 有没有被访问到的依据会引入大量的误差，致使监测 set 访问情况结果不明显，甚至做出错误的判断。

在使用假设检验方法时，通常情况下，需要首先对总体的某项或某些项参数做出假设，之后根据获取的样本做出接收或拒绝假设的判断，它利用了类似于反证法的思想。首先假设针对总体的某项假设成立，之后根据样本推测假设是否合理，若产生不合理现象则拒绝原假设，反之没有足够的理由拒绝原假设，及接受原假设。与反证法不同的是，假设检验中不合理现象指的是小概率事件发生了。小概率事件指的是在一次实验中几乎不可能发生的时间，如果在单次实验中小概率事件发生了，则表示产生了矛盾，通常将发生概率小于 0.05 的时间称为小概率事件，视情况也能修改该值。在 KS 检验中将小概率事件发生的概率α称为显著性水平。

KS 检验主要用于比较假设总体与实际样本或两个总体是否属于同一分布，原假设H_0：样本与假设总体同分布或两个总体属于同一分布，备择假设H_1：样本与假设总体同分布或两个总体属于不同分布。判断标准为比较样本与总体或不同总体之间的差异，及$D = |f(x) - g(x)|$，其中$f(x)$表示抽样总体，而$g(x)$表示假设总体。

4.2.2 KS 检验验证

为了验证 KS 检验区分两个不同分布的能力，本文借助 python 提供的工具包，分几种情况验证 KS 检验的结果。

如下表 1 验证某分布是否属于正太分布。通过调用 python numpy 包提供的函数生成 1000 个服从均值为 0，方差为 1 的样本，然后使用 KS 检验判断其分布是否为正太分布。其中，假设H_0：样本服从正太分布。

表 6 验证某分布是否属于正太分布

源码	from scipy import stats
	import numpy as np
	x = np.random.normal(0,1,1000)
	y = stats.kstest(x, 'norm')
结果	KstestResult(statistic=0.020635092089306473, pvalue=0.78807018015955721)

如表 1 所示，有 numpy 生成的 1000 个样本通过 KS 检验得到 pvalue=0.78807018015955721，而制定的显著性水平α为 5%，明显 pvalue 大于α，故接受原假设，及 x 服从正太分布。

下表 2 验证两个总体是否属于同一个分布，分布为总体 x 和总体 y，它们均由 python 的 numpy 包生成，但均值和方差均不相同，其中 x 服从均值为 5，方差为 6 的正太分布，而 y 服从均值为 0，方差为 1 的正太分布。

表 7 KS 验证两指定分布是否属于统一分布

源码	from scipy import stats
	import numpy as np
	x = np.random.normal(5,6,1000)
	y = np.random.normal(0,1,1000)
	ks_2samp(x,y)
结果	Ks_2sampResult(statistic=0.68799999999999994, pvalue=2.6733739150999497e-208)

很明显，总体 x 和总体 y 服从不同的分布，通过 KS 检验分别对其 1000 个样本进行检测，检测结果的 pvalue=2.6733739150999497e-208，远远小于显著性水平α，因此拒绝原假设，接受备择假设，及 x 和 y 服从不同的分布。

4.3 关键技术解决方案

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 L2 cache 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。本文主要通过两种方式来解决伪随机替换策略可能带来的问题，其一是通过脚本基于目标机进行大量的实验，直到获取到有效的针对目标机的驱逐策略，具体的实现方式见下章验证部分，其二是通过 ks 检验获取 Probe 时间分布

与 cache 命中下的分布进行比较，替代仅仅通过 Probe 时间来度量假设密钥$\hat{k}$的可疑度。

至于异步攻击来说，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。本文在设计异步攻击实验时，将攻击程序和被攻击程序包装被可执行的二进制文件，而不是创建两个 app，这样是为了减少除核心代码外之外的程序访问内存导致 Probe 阶段对被攻击程序访存情况的误判。

此外，由于 AES 加密过程中的查表操作涉及到多种操作：字节替代、行移位、列混淆以及轮密钥加操作。加解密中每轮的密钥分别由初始密钥扩展得到。且加密算法中 16 字节的明文、密文和轮密钥都通过一个 4*4 的矩阵表示。在对 AES 的 T-table 实现进行密钥攻击时，需要熟悉攻击过程以及每个过程中对那个 table 进行查找，并挖掘出可利用的内存位置，提取出 AES 可供利用的数学特性，这些均为 AES 攻击中的难点。本文对 AES 攻击中前两轮的访存情况与明文和密钥之间的关系进行详细的分析，算出了在明文 p 和密钥 k 已知的情况下访问的内存对应的 cache set。

4.4 本章小结

本章介绍了 Cache 攻击方案的设计，其也是本文的核心。首先介绍了如何获取精确的计时方式，其是攻击是否能够成功的关键，然后介绍了如何实现快速高效的驱逐策略，并介绍如何通过脚本的方式获取针对特定机器的驱逐策略。之后详细介绍了针对 AES 加密算法的同步攻击方式，以及一部攻击方式的设计，并介绍了攻击过程可能会出现的问题。最后讨论了设计过程中的关键技术和难点，以及本文给出的解决方案。

第五章 AES 攻击验证

为了验证本文设计的针对移动设备的 Cache 攻击工具的有效性，本章将 Lenovo k51c78 当作被攻击设备，通过 Prime+Probe 对运行于目标机上的 AES 加密程序进行攻击并获取其密钥，以证明 Cache 攻击的有效性以及本攻击方式设计的合理性。目标机的具体配置如下：

表 8 Lenovo k51c78 配置

目标机	**Lenovo k51c78**
CPU	ARM Cortex-53
指令集	Arm64-v8a
L2 cache	512KB
内存映射	16 路组相联
Cache set 数	512
Cache line 大小	64bytes
CPU 核心数	8 核

5.1 时间度量

在 Intel x86 平台上，为了向 cache 攻击提供精确的时间戳，通常使用非特权的 rdtsc 指令。然而在 ARMv7-A 或者 ARMv8-A 指令集架构的移动设备上，没有与之类似的非特权指令。相对的提供了一个性能监控单元 PMU 来监控 CPU 的活动，通过 PMU 虽然可以获得精确的系统时间，但是由于其只能在特权模式下访问，顾在本文中不考虑此种计时方式，本文主要考虑 3 种非特权下也能够提供精确计时的方式。并在 Lenovo k51c78 目标机上进行测试，选择最稳定的计时方式向随后进行的 AES 攻击提供服务。

5.1.1 Perf 计时方式验证

从 Linux 2.6.31 开始，perf 工具引入到了 Linux 内核中。它提供了独立于硬件的用户监测 CPU 性能计数器的接口。通过使用perf_event_open系统调用，在用户态就能访问到 CPU 性能计数器的相关信息，通过将监测属性设置为 PERF_COUNT_HW_CPU_CYCLES，则可返回精确的时钟周期数，就像特权指令获取的那

样。由于该指令依赖于系统调用获取 cpu 的周期，返回的结果会有一定的延迟。下表描述了通过 perf 实现的计时方式，首先定义一个结构体，用于存储从性能计数器获取的值，然后将其传递给perf_event_open函数，性能计数器的值将被读取出来并保存在该结构体重。

为在 Lenovo k51c78 上统计的 Perf 计时度量结果，做了 50000 次实验，但没有获取有效数据，可得 Perf 攻击在目标机上不可用。

5.1.2 POSIX 计时接口验证

然而，某些移动设备可能也不支持 perf 工具，或者 perf 获取的时间精度不够高，此时需要找到其他能够获取精确时间的方式。POSIX标准提供的clock_gettime()函数就是可选方案之一，依赖于向它传递的时钟，它能够返回不同精度的系统时间，比如微妙或纳秒。本文在目标机 k51c78 上测试了 POSIX 计时的准确性，如下所示。

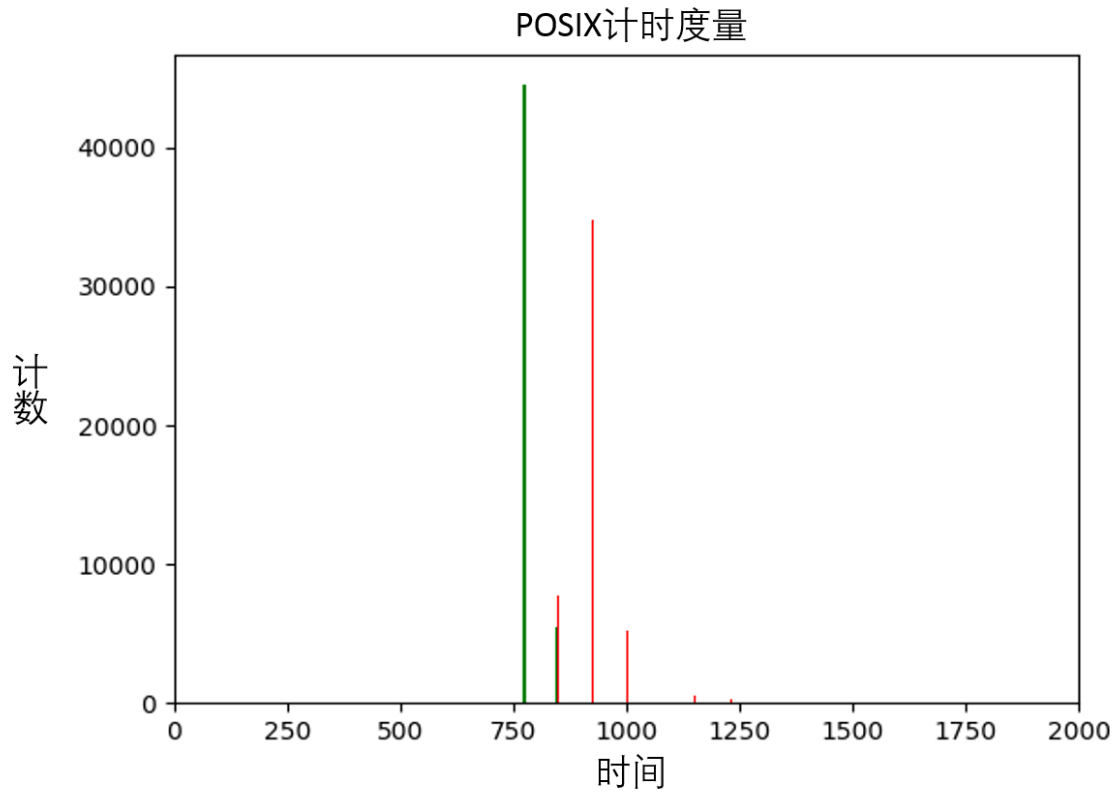

图 13 POSIX 计时度量

图 13 POSIX 计时度量描述了将 POSIX 函数作为时间源得到的 cache hit 与 cache miss 时间分布的直方图，横坐标表示时间，纵坐标表示在时间点上统计的次数。其中总共统计了 50000 个样本的时间，并将其画到一个直方图中，其中蓝色柱表示统计 cache

命中的时间分布，红色柱表示统计的 cache 未命中的时间分布，观察得到红色柱形相对于蓝色柱形有个向右的时间偏移，可将 POSIX 计时方式能够将 cache hit 和 cache miss 区分开来。

5.1.3 线程模拟计时方式验证

在最坏的情况下，如果 perf 工具和POSIX提供的计时方式都不可用，则需要设计一种能够将 cache hit 和 cache miss 区分开的工具，考虑到通常情况下 cpu 执行一条指令的时间是几个时钟周期，可以通过线程不断累加计数器的方式来模拟获取系统时间，不过这种方式只能获取时间区间的值，不过对获取 cache hit 和 cache miss 时间来说已经足够了。

和 Perf 计时方式相似，线程模拟计时也没有得到有效数据，因此线程计时方式在目标机上也不适用。

通过对 3 中计时方式进行度量，发现仅有 POSIX 计时方式适合目标机，所幸 POSIX 提供的接口能够有效的将 Cache hit 和 Cache miss 区分开来。如下的实验中将把计时的时间源设置为 POSIX 提供的方法。

5.2 Cache 驱逐

为了找到一个既快速又高效的驱逐策略，设计者需要详细的了解 cache 的机构以及 cache 的替换策略。对于类似于最近最少替换策略 LRU 等简单的替换策略来说，找一个满足这两个条件的驱逐策略是相对容易的。然而，对与使用伪随机替换策略的 cache 来说就难很多。

为了找到针对目标机 Lenovo k51c78 的最优的驱逐策略，本文采用第三章描述的驱逐策略的设计，将驱逐策略定义为所示：

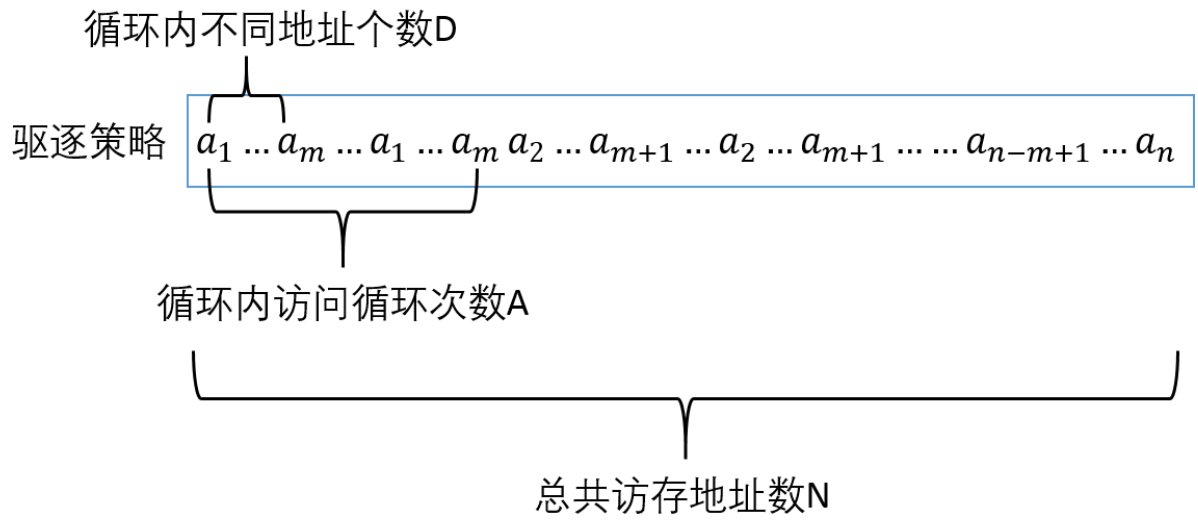

图 14 Cache 驱逐策略

为了能够在使用伪随机算法的移动设备端对指定的 Cache Set 执行高效的驱逐，本文使用**图 14 Cache 驱逐策略**所示的访存模式，当对指定的 Sets 进行驱逐操作是，参数 D 表示循环内的不同地址个数，A 表示驱逐过程中循环内的循环次数，N 表示驱逐过程中总的访存个数，其中被访问的所有地址均能映射到指定的 Set s。

为了获取驱逐效率高，且耗时短的驱逐策略，本文通过脚本的方式对多种可能的参数 D、A、N 组合进行测试，所得结果如**表 9 驱逐策略评估**所示：

表 9 驱逐策略评估

地址个数 N	循环次数 A	环内地址数 D	驱逐率	驱逐时间
24	2	5	100	9121.185348
27	1	8	99.989997	10653.13603
24	3	5	99.919984	12706.28952
21	8	10	99.91996799	36992.2931
24	2	9	99.8582996	13455.61102
22	7	4	99.74994999	12150.68623
20	3	10	99.72994599	13360.58188
22	6	5	99.72994599	20646.16884
23	8	9	99.70519467	41633.32493
17	6	5	99.689969	13905.53812
21	4	10	99.65404965	19774.70509
22	4	8	99.649965	20161.85157
20	1	10	99.64989497	6192.992599

部分驱逐策略的评估结果如上图所示，获取既高效耗时又短的驱逐策略，本文选择参数组合为 N 地址总数为 20，循环内访问地址个数 D 为 10，循环次数 A 为 1 的驱逐策略，通过测试可见其驱逐率为 99.6%，所消耗的驱逐时间为 6192ns。本文如下的实验将使用该驱逐策略组合进行实验。

5.3 AES 攻击

为了验证能够通过 Prime+Probe 以及 K-S 检验对 Android 移动设备进行有效攻击。本文针对 Lenovo k51c78 目标机进行同步 AES 攻击，并成功过去了所有密钥，具体步骤如下。

5.3.1 获取监测数据

攻击过程中的第一步为获取被攻击程序运行期间对 Cache 中各个 Set 的监测数据，在本实验中，首先执行攻击程序，不断探测 AES 加密算法对 Cache 的访问情况，攻击程序运行界面如**图 15 攻击程序的执行过程**所示：

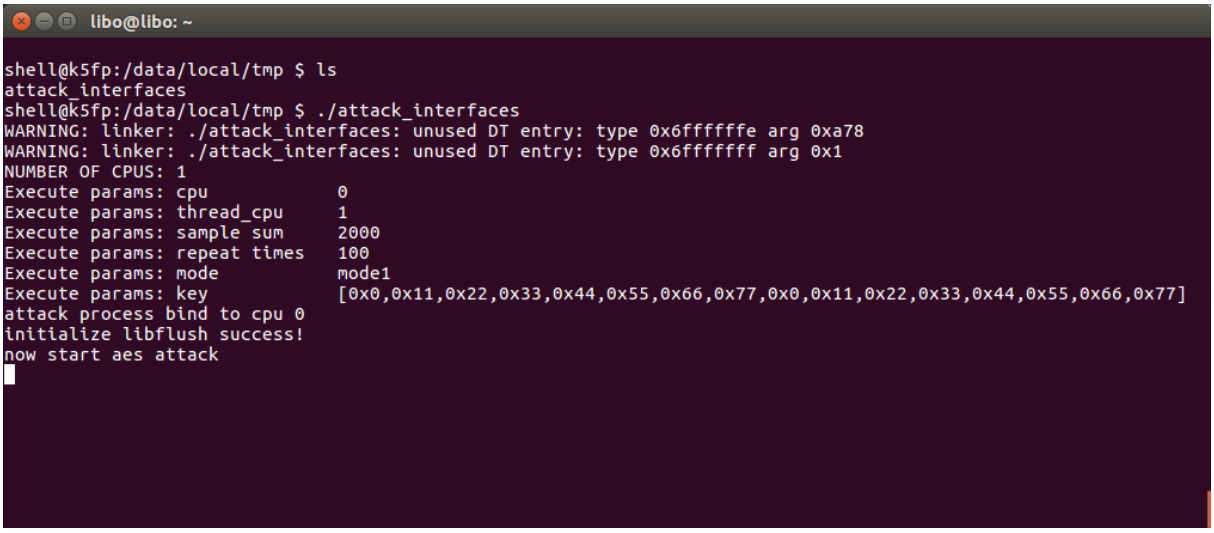

图 15 攻击程序的执行过程

可见，攻击程序绑定到核 0 上，在程序运行过程中总共对 2000 个明文进行加密操作，每个明文重复加密 100 次，获取的数据如**图 16 监听结果**所示：

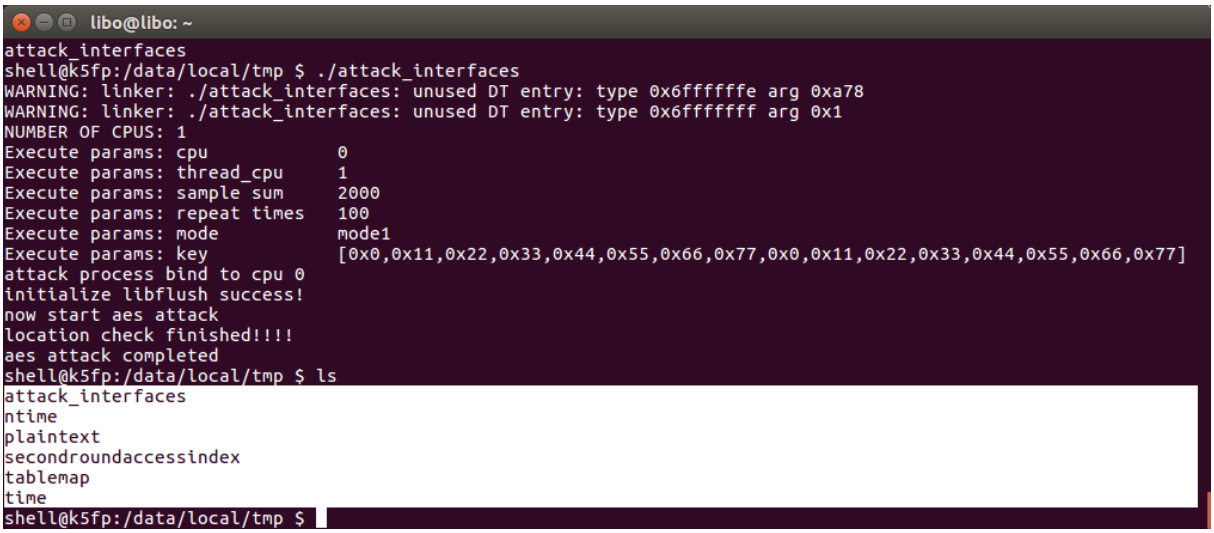

图 16 监听结果

监听结束后共生成 4 个文件，plaintext 文件保存加密的明文信息，tablemap 保存 AES 加密查找表 T-table 中的数据与 Cache Set 的映射关系，time 文件记录攻击过程中监测到的各个 Cache Set 的 Probe 阶段时间，ntime 文件记录各个 Set Probe 阶段全命中的时间，用于于 time 进行对比。在获取监听到的数据之后，将其导入电脑中，通过 python 脚本进行分析获取攻击结果。

5.3.2 对数据进行分析

在分析过程中，对 AES 加密程序进行攻击主要分为 3 个步骤，包括准备阶段、第

一轮攻击和第二轮攻击。准备阶段针对目标机的系统及其 cache 结构，找到精确的计时方式，以及能将 cache hit 和 cache miss 快速有效区分开的驱逐策略。如 4.1 节所示，本文通过对几种计时方式进行分析，最终选取在 k51c78 目标机上表现最好的POSIX作为 cache 攻击的时间源。4.2 节具体讨论了如何获取快速高效的 cache 驱逐策略，通过脚本对成千上万种不同的驱逐策略进行分析，最终选择驱逐率较高且耗时较短的驱逐策略。

攻击步骤中的第二步为第一轮攻击，主要为获取 AES 密钥中每个字节的前 4 位。该轮利用 AES 加密过程中第一轮查表索引与明文 p 和密钥 k 之间的关系，及索引 $i = p \oplus k$，在已知明文 p 的情况下，由攻击程序探测 AES 加密算法在执行对明文进行加密过程中 cache 中 set 的访问情况，最终反推出被攻击程序的密钥 k。

通过多次试验，发现为了提高POSIX提供的函数获取系统时间的精度，需要对其进行预热操作。使用POSIX初次计时会引入较大的误差，因此在 Cache 攻击过程要要尽量减少前几次计时的权重，本文在攻击过程中通过改进度量方式来降低预热带来的误差。由于单次计时会引入较大误差，最终导致无法获取正确的 AES 密钥，本文通过横向对比的方式来获取假设密钥$\hat{k}$的度量分$\hat{m}$。首先，对指定的 set s 进行实验，测量 Prime 操作后紧接着 Probe 操作所得时间 t，则 t 表示 set s 中所有 line 都 cache 命中的时间总和，测量得到多组数据，将其记为集合$\tilde{t}$。之后，对 AES 进行攻击，针对 set s，首先通过 Prime 函数占用 set s 中的所有 line，然后调用 AES 加密算法，对明文进行加密，紧接着调用 Probe 函数探测 set s 的占用情况，得到时间 t，通过对该明文重复进行以上步骤，得到时间集合$\check{t}$。如果在 AES 加密过程中访问到了能够映射到 set s 中的数据，则会将 set s 中的某些 line 驱逐到内存中，在随后的 Probe 阶段将导致 cache miss，由于 cache miss 的时间比 cache hit 的时间要多，此时如果将$\tilde{t}$于$\check{t}$进行对比，则会看到$\check{t}$相较$\tilde{t}$有一个向右的偏移。因此只需要比较$\tilde{t}$和$\check{t}$是否属于同一个分布，就能判断 AES 在加密过程中有没有访问到 set s。

为了量化度量分，引入了 ks 检验，其接收两个分布s_1、s_2作为参数，判断其是否属于同一个分布，并返回一个代表相似度的值 r，r 取值在 0 到 1 之间，其越接近 0 表示s_1和s_2越相近，越接近 1 表示s_1和s_2越不同。将 ks 检验作用于$\tilde{t}$和$\check{t}$就能方便的判断 AES 在加密过程中是否访问到 set s，返回的结果也可作为被用于计算假设密钥$\hat{k}$的可疑度。

因此，首先通过 AES 加密算法对 n 个明文进行加密，每个明文重复重复进行 m 次

加密，以获取单个样本多个数据，每项数据包含对每个 cache set 的监测信息，通过假设密钥$\hat{k}$，并让其遍历密钥所有可能的取值，并获取每项假设的可疑度，则可疑度最高的假设值就是破解得到的密钥。

本实验设置 AES 密钥值为：k＝(0x00,0x11,0x22,0x33,0x44,0x55,0x66,0x77,0x00,0x11,0x22,0x33,0x44,0x55,0x66,0x77)，第一轮攻击利用 AES 加密第一轮索引$i = p \oplus k$，通过对 1000 个不同明文进行加密，每个明文重复加密 2000 次，前 1000 次探测 Prime 后紧接着 Probe 操作获取的时间，而后 1000 次探测 Prime 后执行 AES 加密，随后执行 Probe 操作获取得到的时间，对这两部分数据进行 ks 检验并将其作为该明文样本的数据。第一轮攻击结果如下图所示：

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
第0字节	9340150	9282150	9280950	9262150	9294150	9264250	9283750	9279400	9288850	9289300	9269250	9301550	9272250	9280250	9285700	9266850
第1字节	9291550	9370150	9267350	9302700	9294700	9306350	9319550	9304050	9311250	9271200	9293950	9263850	9291900	9277650	9288700	9316300
第2字节	8790050	8805950	8901850	8786950	8790850	8792850	8797550	8794850	8808550	8799600	8806250	8774550	8811700	8810500	8768000	8786500
第3字节	9216300	9249600	9243050	9291500	9232700	9227350	9226100	9225900	9234500	9234450	9256200	9208450	9240200	9244400	9211400	9235150
第4字节	9285950	9268200	9281050	9279450	9350650	9272800	9279500	9267000	9284500	9288700	9280000	9268400	9283600	9293550	9276050	9281550
第5字节	9288600	9295150	9309350	9321450	9301000	9348500	9308450	9310250	9290050	9269450	9258600	9303400	9307500	9277150	9292350	9289950
第6字节	8781250	8823800	8801150	8781650	8788000	8817250	8880650	8784700	8784000	8787950	8823200	8780950	8786700	8799450	8804100	8801750
第7字节	9237350	9224100	9239900	9248300	9234200	9246350	9217900	9296650	9240000	9236900	9226100	9225350	9211700	9230950	9232850	9228650
第8字节	9343350	9281800	9274100	9292450	9282150	9286000	9281000	9268600	9276550	9270700	9287700	9287050	9292500	9262000	9283550	9271450
第9字节	9312400	9351200	9328600	9320800	9286500	9280200	9276850	9294200	9281050	9301150	9288500	9274650	9291850	9301700	9305300	9276250
第10字节	8783350	8778500	8910950	8816300	8768550	8826950	8791150	8810550	8811050	8799200	8803600	8781650	8828450	8792700	8776050	8747550
第11字节	9223650	9233700	9231000	9296000	9255900	9235950	9226500	9234650	9224100	9228200	9210250	9239050	9237000	9224800	9239750	9236750
第12字节	9265150	9282250	9273100	9258300	9329450	9259350	9253800	9279200	9311950	9303700	9289900	9298050	9288800	9295200	9291500	9261250
第13字节	9309750	9289050	9293400	9287550	9274200	9330500	9319100	9303850	9289300	9327050	9316100	9295800	9287550	9285650	9277800	9284550
第14字节	8794150	8792300	8800900	8790500	8792150	8801350	8895900	8786600	8782150	8769150	8787000	8818600	8826800	8802350	8796300	8790350
第15字节	9239200	9215800	9245650	9238850	9216600	9231750	9231450	9292850	9234350	9238200	9219950	9237600	9228500	9228800	9237600	9240100

图 17 第一轮攻击结果

上图表示第一轮攻击结果，从上到下，第 0 字节到第 15 字节的攻击结果，从左到右为字节前四位的所有可能取值，从 0 到 F，每个小图的横坐标表示密钥每个字节前 4 位假设值，从 0 到 15 遍历了密钥每个字节前 4 位的所有取值，表格中的值表示可疑度，其值越高表示对应的假设值可疑度越高，其中每个图可疑度最高的值即为攻击得到的结果。因此，本轮攻击得到的密钥的每个字节的前四位为：(0x0,0x1,0x2,0x3,0x4,0x5,0x6,0x7,0x0,0x1,0x2,0x3,0x4,0x5,0x6,0x7)。与真实的密钥每个字节前四位相同，因此第一轮攻击成功获取了其所需信息。

第一轮攻击每个密钥字节的前4位是假设的一个基本单位，密钥字节之间互不影响，因此可以 16 个字节分别获取，密钥假设空间共有2^4种可能，对应图一中子图横坐标。由于第二轮查表索引比第一轮复杂，第二轮攻击也较第一轮攻击要复杂一些。AES 第二轮查表索引 n 为：

$$n_2 = s(p_0 \oplus k_0) \oplus s(p_5 \oplus k_5) \oplus 2 \bullet s(p_{10} \oplus k_{10}) \oplus 3 \bullet s(p_{15} \oplus k_{15}) \oplus s(k_{15}) \oplus k_2$$

$$n_5 = s(p_4 \oplus k_4) \oplus 2 \bullet s(p_9 \oplus k_9) \oplus 3 \bullet s(p_{14} \oplus k_{14}) \oplus s(p_3 \oplus k_3) \oplus s(k_{14}) \oplus k_1 \oplus k_{15}$$

$$n_8 = 2 \bullet s(p_8 \oplus k_8) \oplus 3 \bullet s(p_{13} \oplus k_{13}) \oplus s(p_2 \oplus k_2) \oplus s(p_7 \oplus k_7) \oplus s(k_{13}) \oplus k_0 \oplus k_4 \oplus k_8 \oplus 1$$

$$n_{15} = 3 \bullet s(p_{12} \oplus k_{12}) \oplus s(p_1 \oplus k_1) \oplus s(p_6 \oplus k_6) \oplus 2 \bullet s(p_{11} \oplus k_{11}) \oplus s(k_{12}) \oplus k_{15} \oplus k_3 \oplus k_7 \oplus k_{11}$$

可以看出在 Table T_2的上的查表索引n_2与k_0、k_2、k_5、k_{10}、k_{15}相关，然而由于n_2的后 4 位仅仅决定访问到的数据在 cache line 中的偏移，因此在第二轮攻击获取密钥字节的后四位过程中k_2的后 4 位对决定n_2索引能够映射到的 cache set 没有影响，因此只需同时设定k_0、k_5、k_{10}、k_{15}的值就能决定在 Table T_2的上的查表索引。同理，k_3、k_4、k_9、k_{14}的值就能决定在 Table T_1的上的查表索引，k_2、k_7、k_8、k_{13}的值就能决定在 Table T_0的上的查表索引，k_1、k_6、k_{11}、k_{12}的值就能决定在 Table T_3的上的查表索引。

通过第一轮攻击，已经成功获取到每个密钥字节的前 4 位。在进行第二轮攻击的过程中，可以分为 4 个步骤，第一步假设k_0、k_5、k_{10}、k_{15}值$\hat{k}_0$、$\hat{k}_5$、$\hat{k}_{10}$、$\hat{k}_{15}$后四位的值，并使其遍历取值空间，共有$2^{4*4} = 65536$种组合，每种组合和明文 p 相运算就能获取到访问索引，通过第一轮攻击中相同的方式即可获得$(\hat{k}_0, \hat{k}_5, \hat{k}_{10}, \hat{k}_{15})$组合的可疑度，其中可疑度最高的组合方式的取值就是攻击获取的密钥值。通过同样的方式能够获取到密钥的其他字节的后四位。本文分析的实验结果如下所示：

表 10 第二轮攻击结果(k_0, k_5, k_{10}, k_{15})

k_0后四位取值	k_5后四位取值	k_{10}后四位取值	k_{15}后四位取值	度量分
13	3	2	5	0.952152
1	4	10	8	0.952174
8	15	10	15	0.952371
8	5	8	9	0.952374
1	15	0	3	0.9524125
7	4	4	5	0.952766
0	5	2	7	0.9533185

表 11 第二轮攻击结果(k_4, k_9, k_{14}, k_3)

k_4后四位取值	k_9后四位取值	k_{14}后四位取值	k_3后四位取值	度量分
10	4	8	10	0.9384495
4	7	6	11	0.9384925
1	13	0	1	0.938506
9	6	12	15	0.9385945
13	0	8	12	0.938629
2	14	10	6	0.938634
4	1	6	3	0.938652

表 12 第二轮攻击结果(k_8, k_{13}, k_2, k_7)

k_8后四位取值	k_{13}后四位取值	k_2后四位取值	k_7后四位取值	度量分
4	8	7	11	0.9570985
11	7	12	15	0.95713
14	14	5	11	0.9571695
8	1	12	2	0.9572065
14	3	8	2	0.9572265
4	7	14	8	0.9574545
0	5	2	7	0.9578595

表 13 第二轮攻击结果(k_{12}, k_1, k_6, k_{11})

k_{12}后四位取值	k_1后四位取值	k_6后四位取值	k_{11}后四位取值	度量分
4	15	3	14	0.830743
1	9	6	10	0.8309515
5	15	0	12	0.8312265
7	14	5	3	0.831524
4	1	12	4	0.8315825
14	4	9	12	0.831855
4	1	6	3	0.8327175

表 10 第二轮攻击结果$(\boldsymbol{k_0},\boldsymbol{k_5},\boldsymbol{k_{10}},\boldsymbol{k_{15}})$、**表 11 第二轮攻击结果**$(\boldsymbol{k_4},\boldsymbol{k_9},\boldsymbol{k_{14}},\boldsymbol{k_3})$、**表 12 第二轮攻击结果**$(\boldsymbol{k_8},\boldsymbol{k_{13}},\boldsymbol{k_2},\boldsymbol{k_7})$、**表 13 第二轮攻击结果**$(\boldsymbol{k_{12}},\boldsymbol{k_1},\boldsymbol{k_6},\boldsymbol{k_{11}})$表示第二轮攻击的结构，由于每一个图中均有$2^{4*4}=65536$种可能的密钥组合，因此仅仅把可疑度较高的密钥组合列出来。在**表 10 第二轮攻击结果**$(\boldsymbol{k_0},\boldsymbol{k_5},\boldsymbol{k_{10}},\boldsymbol{k_{15}})$中，获取到的可疑度最高的$(k_0,k_5,k_{10},k_{15})$密钥组合为(0,5,2,7)，从**表 11 第二轮攻击结果**$(\boldsymbol{k_4},\boldsymbol{k_9},\boldsymbol{k_{14}},\boldsymbol{k_3})$获取到的可疑度最高的$(k_4,k_9,k_{14},k_3)$密钥组合为(4,1,6,3)，从**表 12 第二轮攻击结果**$(\boldsymbol{k_8},\boldsymbol{k_{13}},\boldsymbol{k_2},\boldsymbol{k_7})$获取到的可疑度最高的$(k_8,k_{13},k_2,k_7)$密钥组合为(0,5,2,7)，从**表 13 第二轮攻击结果**$(\boldsymbol{k_{12}},\boldsymbol{k_1},\boldsymbol{k_6},\boldsymbol{k_{11}})$获取到的可疑度最高的$(k_{12},k_1,k_6,k_{11})$密钥组合为(4,1,6,3)，整理得到第二轮获取到的 AES 密钥字节后 4 位为 $(0x0,0x1,0x2,0x3,0x4,0x5,0x6,0x7,0x0,0x1,0x2,0x3,0x4,0x5,0x6,0x7)$，与第一轮攻击结果组合得到通过 Cache 攻击得到的 AES 密钥为$\hat{k}=$ (0x00,0x11,0x22,0x33,0x44,0x55,0x66,0x77,0x00,0x11,0x22,0x33,0x44,0x55,0x66,0x77)，与实验设置的密钥值相同。

5.4 本章小结

本章在目标机 Lenovo k51c78 上做了针对 AES 加密算法的 Cache 攻击，并通过同步攻击成功破解了 AES 全部密钥。证明了 Cache 攻击不仅仅在 Intel x86 平台，在 ARMv7-a 以及 ARMv8-a 等移动平台上也能通过内存访问模式获取到用户的私密信息。本章首先通过实验选择在目标机上比较精确的POSIX计时接口，然后通过脚本对多种驱逐方式进行度量，最终选择驱逐效率最高的一种驱逐策略。在确定好计时方式和驱逐策略之后进行 AES 同步攻击，成功获取 AES 全部密钥，最后对 AES 异步攻击方式进行探索。

第六章 预防攻击措施

通过上一章的实验，可以确信 Cache 攻击不仅仅在 Intel x86 平台上获取用户的私密信息，在做了相应改进的情况下同样也可以应用在 ARM-v7a、ARM-v8a 等移动设备上，比如本文通过对数千个样本进行分析就成功获取到了 AES 的全部密钥值。为了避免不法分子利用该漏洞获取用户的私密信息，本章讨论 cache 的漏洞以及相应的对策。

6.1 攻击漏洞

Cache 攻击通过利用攻击程序探测 Cache 上泄露的被攻击程序的内存访问模式来进行攻击的。被攻击程序与攻击程序之间互不干扰，各自访问各自内存空间的数据，但 Cache 使得他们之间的数据有了间接的交互，比如当被攻击程序运行时，当其执行访存操作时，将到 cache 上寻找数据是够缓存在 cache 中，若存储在 cache 中则直接将数据返回，若没有缓存在 cache 中，则到内存中寻找数据，并将其缓存在对应的 cache set 中，占据其中的一个 line。然后攻击程序执行，其访存操作同样也会将数据读取到 cache 中，当读取的数据正好也映射到被攻击程序映射到的 cache set 时，根据 cache 替换策略，就有可能将其驱逐回内存中。这样，当下次被攻击程序访问被攻击程序驱逐到内存中的数据时，就会导致一次 cache miss，必须从内存中重新获取数据，并使得读取时间变长，也就相应的影响了被攻击程序的行为。Cache 攻击本质上也就是通过利用这一漏洞进行攻击的。

除了 Cache 设计的漏洞之外，程序本身的漏洞也是实现 cache 攻击的必要条件。攻击程序需要通过 Cache 获取被攻击程序与私密信息相关联的内存访问模式，并依此反推出被攻击程序的私密信息。以 AES 加密过程为例，攻击程序通过监测 AES 加密程序对 Cache 中各个 set 的使用情况来反推密钥，而 AES 加密过程中对 Cache set 的使用情况是由明文和密钥来决定的。因此当获取到 cache 使用情况，以及被加密的明文数据之后，就能反推出用户密钥。

此外，对 cache 攻击来说，被攻击程序不同的输入能够影响到对内存中不同区域的访问是至关重要的，由此攻击程序才能通过 Cache 探测私密信息。本章首先讨论系统硬件、操作系统以及应用程序的漏洞，随后针对这些漏洞给出预防缓存攻击的措施。

6.1.1 Cache 设计缺陷

Cache 是为了解决 CPU 与内存速度的不一致问题，并在性能与价格妥协的产物。为了提高访存速度，系统在访问内存时会将访问的数据以及周围数据缓存到 cache 中，下次再访问该数据时将直接从 Cache 中返回数据给 CPU。Cache 的出现导致了访存速度不一致问题，及直接从 Cache 获取数据的访存操作比从内存中获取数据的访存操作速度要快。因此，如果能够获取其访存的时间，就能推测出数据是在内存中还是缓存在 Cache 中。

移动设备与 Intel x86 设备的 Cache 结构相似，都由多层 Cache 缓存，为了方便分析，本文仅讨论有两层结构的 Cache，下图为 ARM 平台的 Cache 结构示意图，图中的处理器共有核 0 和核 1 两个核，每个核心都有两个 L1 Cache，及 L1I 和 L1D，分别为指令缓存器和数据缓存器，它们共用一个 L2 缓存。

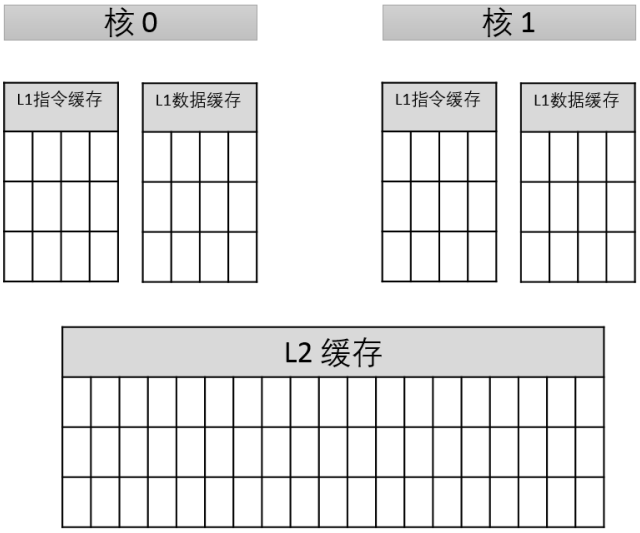

图 18 ARM 处理器 Cache 结构

由于 Cache 容量比内存容量小很多，不可能将内存中的所有数据缓存到 Cache 中，因此为了提高整体的访存速度，开发人员设计了多种不同的主存与缓存的映射关系，及全相联映射、组相联映射、直接相联映射等，这些映射关系的本质是规定什么样的地址映射到 Cache 中的某一区域。由于内存比缓存大很多，导致内存中多个地址映射到同一个缓存区域，这为不同程序间相互影响提供可能。当一个程序 a 有足够大的内存空间，对其空间内地址进行访问，就能占用整个 Cache，此时假设另一程序 b 也在运行，则 a 程序的访存操作就有可能使得 b 程序缓存在 Cache 中的数据被驱逐到内存中，使得程序 b 运行速度较慢。同理程序 b 的访存操作也会使得程序 a 的访存操作较慢，通过对访存

操作计时，就能判断程序 a 的访存和使用 Cache 的情况。

6.1.2 操作系统缺陷

操作系统在实现 Cache 攻击的过程中提供了不可或缺的支持，它像攻击者提供了精确的计时方式，提供了查询虚存与实寸映射关系的文件，提供了驱逐 Cache Set 中数据的指令，而这些支持是实现 Cache 攻击的关键。

虽然操作系统限制进程只能查询本地内存空间虚存与实寸的映射关系，但对于 Cache 攻击来说，已经足够了。Cache 攻击的目标是能够某个 Set 进行监控，拥有驱逐该 Set 中所有数据的能力，因此首先需要知道哪些地址与该 Set 相关，及哪些地址中的数据能够映射到该 Set 中。操作系统提供的查询本地内存空间虚存与实寸映射关系的 /proc/<pid>/pagemap 文件保存着各个进程虚存与实存的映射关系，因此能够获取指定虚存映射到的物理地址，根据 Cache 与内存的映射关系，通过物理地址又能获取到此地址映射到的具体 Set 号，就完成了虚拟地址到 Cache Set 号的映射。对于指定的 Set 号，获取多个与该 Set 相关的地址，且满足这些地址不在同一个内存块中，得到与该 Set 相关的地址集合 m。读取 m 中的地址就能达到将该 Set 中存储的数据驱逐到内存中的目的。

除了提供获取虚拟与实寸的映射关系外，有的系统提供了针对指定 Set 的 Cache 驱逐指令，通过该指令能够将指定的 Cache Set 中的数据全部驱逐，这会得到比地址驱逐更高的驱逐率，并为 Cache 攻击提供服务。

由于很多应用或服务有获取系统时间的需求，操纵系统通常提供多种不同的获取系统时间的方式，比如计时寄存器、POSIX 提供的计时函数、perf 性能监控工具等，这些计时方式除了向普通程序提供获取精确的系统时间之外，也向 Cache 攻击程序提供服务。通过这些计时方式，攻击者能够探测到被攻击程序对 L2 缓存的使用情况，从而实现攻击过程。

对操作系统来说，缺陷在于将攻击程序视为普通的程序，并为其提供精致准确的服务，因此某种程度上可以说操作系统是攻击者的帮凶。

6.1.3 应用程序漏洞

除了 Cache 本身存在的缺陷以及操作系统的漏洞之外，应用程序本身的缺陷也是 Cache 攻击能够成功的必要条件。Cache 攻击仅仅能够通过访存时间获取其他程序对 Cache 的使用情况，并对 Cache 的使用进行分析以获取与私密信息相关的有用信息。而

对于一些程序而言，其私密信息与内存之间有对应关系，内存又与 Cache 又有某种确定的映射关系，因此私密信息与 Cache 的使用情况之间也有某种映射关系，当攻击程序得到被攻击程序对 Cache 的使用情况之后，就能得到用户的私密信息。

以 T-Table 实现的 AES 加密算法为例，加密密钥 k 为用户的私密信息，使用该加密算法对不同的明文 p 进行加密时，在 1~10 轮的加密过程中会对不同的索引表中的区域进行加密，由于索引表较大，不同区域的索引数据映射到 Cache 中的不同 Set 中。针对同样的明文 p，当密钥 k 值不同时，在加密过程中也会导致对内存中不同区域数据的访问，最终映射到不同的缓存中。当攻击者能够获取 Cache 状态时，就能根据明文以及探测到的 Cache 访问情况反推出用户的密钥。

很多情况下，用户对键盘的输入也是很重要的私密信息，然而对于某些保密措施比较为简单的输入库，其不同输入会导致输入库不同区域的访问。

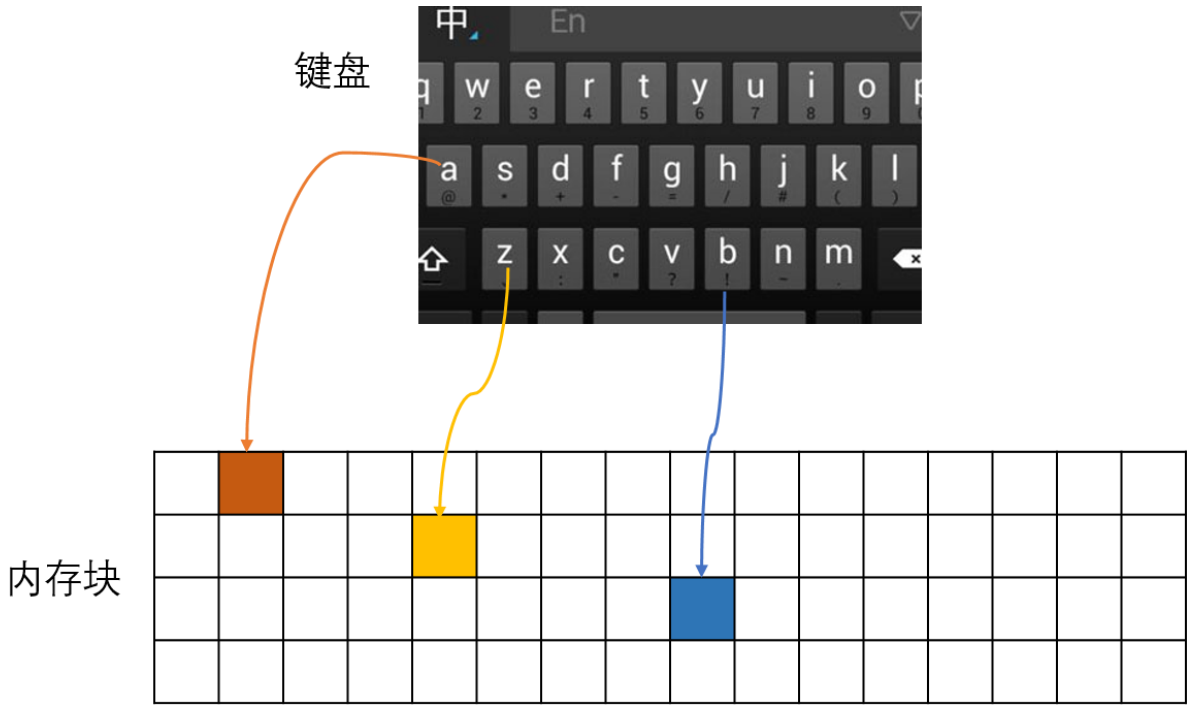

图 19 输入库漏洞

图 19 输入库漏洞显示了输入库可能出现的漏洞，当用户输入不同的字母时，可能导致对不同的内存区域的访问，且这些内存区域能够映射到不同的 Cache Set 中，极端情况下，假设对 26 个英文字母的点击访问的内存地址映射到 26 个不同的内存区域中，则攻击程序只需要不断的监测这 26 个不同的 Set，就能判断出用户输入的是键盘上的哪些字母。

由于用户在编写共享库或其他软件的时候很少会考虑到 Cache 攻击可能利用的漏洞，因此，程序泄露出来的漏洞是广泛存在的。

6.2 预防攻击措施

本文首先对 Cache 结构的进行详细描述，然后设计和实现了基于 Lenovo k51c78 目标机的 AES 加密算法的攻击，并成功的获取到到了 AES 的全部密钥。上一节中讨论了 Cache 设计中的缺陷、操作系统的漏洞以及应用程序的漏洞，本节将将这 3 个漏洞作为出发点，总结 Cache 攻击中的关键点，以预防 AES 加密算法信息泄露为例提出几个针对 Cache 攻击的几点预防措施。

6.2.1 避免内存访问

Cache 攻击通过探测被攻击程序对 Cache 的使用情况，并结合私密信息与内存使用情况的联系，分析出用户的私密信息。因此程序没有对内存的访问时，Cache 攻击就没有可分析的数据，依此为出发点，在设计应用程序时尽量避免对内存数据的访问，特别是跨内存块的数据访问。由于攻击者只能拿到 Cache Set 粒度的信息，通过 Cache 攻击无法判断应用程序对映射到该 Set 中的内存块的哪些区域地址的访问，也无法判断访问了多少数据，因此在无法避免内存访问时也可考虑尽量对一个内存块的数据进行访问。

本文执行的 AES 攻击利用了加密过程各轮的访存操作，每一轮查索引操作会读取不同索引位置的数据，这些访存操作最终影响到 Cache 中的各个 Set，因此通过改进 AES 加密程序，避免对 table 索引表的访问，能够有效的防御 cache 攻击。对 AES 来说，可以通过以下方式达到减少访存的目的。

首先，可以通过等效的逻辑运算代替查索引表操作。对 AES 来说，这种替换方式是相当容易的，因为查表操作都有相应的简洁的逻辑描述，但计算的性能会比查表的性能慢几个数量级。由于通过逻辑运算替换了查表操作，就不存在在对不同明文和不同密钥进行加密时访问到不同内存区域的数据，也不会导致对不同 cache set 的访问，因此能够有效的预防 cache 攻击。

此外，也可以将 AES 加密操作需要使用的查表表保存才寄存器中而不是缓存在 cache 中来预防 cache 攻击。在 Intel x86-64，PowerPC AltiVec 以及 Cell SPE 架构的设

备中，已经有大到能够容纳 256bytes 的 S-box 表，并且有相关的指令进行高效的查询操作。由于不涉及到对 cache 的访问，加密时直接从特定的缓存中获取数据，加密过程不会对各个程序共享的 L2 缓存进行访问，因此即使攻击者知道加密的明文，也无法通过 Cache 攻击获取用户的密钥信息，而且由于索引表常驻寄存器，加密所需的时间更短。

6.2.2 修改索引表

对 AES 加密算法来说，加密和解密过程中需要经历 10 轮变换，包括行移位、列混淆、轮密钥加等操作，每一轮操作都需要进行若干查表操作，其中加密和解密的过程需要对几个不同的查找表做索引操作，这些索引表的格式比较相似。如本文第二章和第四章，AES 大部分实现使用 4 个 1024-byte 查找表$T_0,\dots,T_3$作为 1-9 轮的查表索引。然而，针对索引表也有相关的改进措施，比如有的 AES 加密实现将$T_0,\dots,T_3$改进为为 1 个 256-byte 的查找表作为 S-box 函数，2 个 256-bytes 的查找表作为$2\bullet \mathrm{S}[\cdot]$，以及 1 个 1024-byte 查找表和 1 个 2048-byte 的查找表（$T_0,\dots,T_3$压缩为一个查找表），对第十轮的查表索引$T_0^{(10)},\dots,T_3^{(10)}$也做相应的转换。这样可以减少索引表所占的内存空间，缩小 AES 加密过程中访存位置对明文和密钥的敏感度，进而提高 Cache 攻击的难度。

对于本文第 4 章在 Lenovo k51c78 执行的针对 AES 的 Cache 攻击而言，缩小索引表会降低当假设密钥$\hat{k}$并非真实密钥 k 时通过计算得到的指定的内存在 AES 加密过程中没有被访问到的概率，及增加了误判的概率，因此能提高破解 AES 密钥的难度。也可以通过对极端情况进行分析研究降低索引表所占内存区域对 Cache 攻击的影响。假设通过某种方式，将 AES 索引表所占内存缩小到一个内存块大小，此时，不论明文和密钥取何值，均会访问到该内存块中的数据，因此该内存块将常驻在缓存中。当使用 Cache 攻击程序探测 Cache 的使用情况时，只能获取到该 Set 一直被加密算法访问，此外得不到任何信息。

对其他程序也是一样，应该尽量避免使用大的表结构数据，这样能够减少通过 Cache 能够获取的有关用户程序的信息。

6.2.3 动态表索引

Cache 攻击通过观察内存的访问类型来了解到 AES table 查找的信息。除了消除 table 可能泄露的信息这种方法之外，也可以扰乱 table 泄露出来的访问情况。比如，AES 加密程序在内存中有几份索引表的副本，其中每份副本映射到的 cache set 均不相同，在加密过程中需要查索引表时，通过随机数生成算法随机选择一个查找表进行查找。理想状态下，AES 加密程序每个索引表均有 S 份副本，其中 S 为 cache set 的个数，虽然这会导致几乎所有的查表操作均会导致 cache miss 的发生，但当决定访问副本的随机数完全随机时，AES 的索引表将任意的缓存在 cache 中。由于随机访问表索引的存在，使得通过明文和密钥无法获取准确的索引表查找位置对应的 Cache Set 号，而在攻击过程中获取的索引号只有 1/S 的可能性是正确的，则最终正确获取所有访问 Set 数据的可能性为（$1/S$）^{n}，n 为探测的 Cache Set 数，当 n 很大是，获取正确数据样本的可能性趋近于 0。因此通过 Cache 攻击工具获取正确有效的用户密钥的可能性也趋近于 0。

然而，在内存中同时保存索引表的多个副本在内存资源紧张的时候会加重系统负担，以 AES 加密为例，其 1 到 10 轮加密过程中共用到了T_0、T_1、T_2、T_3、$T_0^{(10)}$、$T_1^{(10)}$、$T_2^{(10)}$、$T_3^{(10)}$ 8 个索引表，其中每个表 1KB，总共占用 8KB 的内存空间，若副本 S 的取值为 5，则需要占据 40KB 的内存空间，这在内存资源吃紧时是不可承受的。为了在内存占用和安全性之间做出折中的话，可以动态变换 S 的取值，当内存空闲率较高时，增大 S 的取值，而在内存吃紧时缩小 S 的取值。另一种混淆内存访问方式的方法是在每次加密完成后就改变索引表的布局，并记录下改变的方式，方便下次查表时能够访问到正确的数据。这种方式不需要保存索引表的副本，但是由于每轮查表后会改变索引表的布局，因此会增加加密执行的时间，适合在对加密时间要求不高，加密次数较少的情况中。

6.2.4 操作系统支持

本文执行的针对 AES 的 Cache 攻击实验，依赖操作系统提供的多项支持，包括提供精确的计时方式、提供对/proc/< pid >/map文件的访问以获取虚拟内存到物理内存的映射关系、提供将 Cache 中某一 Set 数据驱逐到内存中的系统指令等功能，而这些支持对实现Cache攻击是至关重要的。因此，如果操作系统关闭或收紧对这些服务的支持，

会有效的提高用户数据的安全性。

当然如果操作系统禁止用户程序访问计时相关的接口，禁止用户态下对虚实地址映射表的访问，那么获取访存和访问 Cache 的时间就没有那么容易，也无法获取虚拟内存映射到的具体的 Cache Set 号，因此不能通过读取能够映射到指定 Cache Set 中的数据来对该 Set 进行驱逐操作，因此也就没法对其进行攻击。但这也会误伤其他正常调用这些方法的程序。为了预防 Cache 攻击，同时不妨碍其他正常程序对计时接口的访问，可以检测攻击模式下对接口的访问模式与正常使用模式的区别，并屏蔽不正常的访问。

此外针对操作系统提供的计时寄存器以及 POSIX 和 perf 接口，由于其向调用者返回精确的系统时间，POSIX 返回当前系统时间，精确到纳秒级别，perf 甚至能精确到具体的 CPU 周期，而系统提供的计时寄存器也能提供 CPU 周期级别的系统时间。通过这些工具能够精确的度量出一项访存操作访问的是主存还是访问缓存到 Cache 中的数据，因此能简单的通过其进行 Cache 攻击。然而，对于普通的应用程序来说，精确到纳秒级别甚至 CPU 级别的系统时间性能有些过剩了，通常情况普通应用程序只需要精确到秒或毫秒精度的时间，因此可以适当降低操作系统提供的计时方式的精度，使得通过计时方式无法判断一次访存操作访问的内存中的数据还是直接从 Cache 中获取数据，这也就无法通过计时方式实现 Cache 攻击。

针对系统提供的通过虚存获取其对应的物理内存的服务，在 Android 平台新发布的 Android 6.0.1 以后的版本中，已经阻止了非特权模式下对/proc/< pid >/map虚实地址转换表的访问，因此对于没有 root 的移动设备，无法通过对该表进行访问获取到虚拟内存地址到物理地址之间的转换关系，因此也无法获取到一指定虚存映射到的 Cache Set 的索引，能够有效的预防常规需要依赖虚实地址转换的 Cache 攻击。

6.3 本章小结

本章介绍了 Cache 结构设计引入的漏洞，由于其在设计之初没有考虑到通过内存访问不同的程序之间能够相互获取信息，才使得通过 Cache 获取用户私密信息成文可能。随后根据本文在设计并执行针对移动设备 Lenovo k51c78 的 AES 攻击的过程中总结的与攻击成功相关的知识，提出在移动设备上可行的保护用户隐私，预防 Cache 攻击的措施，具体包括限制对内存的访问、更改 AES 索引表、动态转移所以表以及限制程序对操

作系统提供支持的访问。此外本章还介绍了 Cache 设计存在的缺陷、操作系统存在的漏洞，以及应用程序可能出现的漏洞，并针对这些缺陷给出以 AES 加密算法为例的建议，以保护用户的隐私。

总结与展望

工作与研究总结

自从 Cache 攻击被提出以来，涌现了大批科研人员对其进行研究，提出了多种攻击方式以及预防措施，并在 Intel x86 平台破解了 DES、AES 加密算法的部分或全部密钥，也能够通过 Cache 攻击跟踪用户的输入，通过 Cache 进行通信传递数据。然而由于移动设备的指令集、Cache 组织及替换策略等与 Intel x86 平台有很大的差距，因此之前没有成功在 ARM 平台上实现 Cache 攻击的示例，直到最近几年才有基于移动设备获取用户私密信息的 Prime+Probe、Flush+Reload、Evict+Reload 等方法研究，本文基于 Prime+Probe 方法，并对 Intel x86 平台的攻击方式进行改进，通过引入 ks 假设检验获取度量分数的方式来降低计时误差，降低偶然因素以及 cache 伪随机提替换算法对实验结果的影响，并通过攻击成功的获取了 AES 的全部密钥。

本文首先介绍了 Cache 的结构，讨论了 Cache 与主存的各种不同映射方式，以及 Cache 的替换策略、Cache 的包含性等与缓存攻击相关的内容，之后介绍了 Cache 攻击两种典型的攻击方式，及 Evict+Time 和 Prime+Probe 两种攻击模式，本文在后面的实验中通过 Evict+Time 获取缓存命中和缓存缺失的时间阀值，通过 Prime+Probe 实现针对 AES 加密算法的攻击。之后本文详细介绍了 AES 加密算法的执行过程，重点描述了在执行加密过程中的各轮查表操作，这些查表操作也是对其进行攻击的切入点。

在随后的章节中，本文介绍了前人已经研究过的几种能够实现精确计时的方式，及 POSIX 提供的系统调用、Perf 性能分析攻击提供的调用接口以及通过线程模拟获取系统时间，它们作为时间源是基于计时方式的缓存攻击的基础。之后本文介绍了在没有系统指令支持的情况下驱逐指定 Cache 中数据的方式。接下来本文详细描述了 KS 检验的思想，并验证其分辨不同样本分布的能力。结合 KS 检验，随后描述了本文设计的 Cache 攻击的详细步骤，通过分两轮攻击的方式一步步的获取 AES 密钥的全部字节信息，之后给出了攻击过程中的关键技术和解决方案。

在给出攻击方案的设计之后，本文以 Lenovo k51c78 作为攻击对象，针对开源的 T-Table 实现的 AES 加密算法进行实验，在验证可用的时间源，高效的 Cache 驱逐策略之后，通过引入基于 KS 检验的 Prime+Probe 攻击模式，成功的获取了 AES 的全部密钥信

息。

最后本文总结了攻击过程中使用的 Cache 设计缺陷、操作系统的漏洞以及待攻击程序的漏洞，并针对这些漏洞给出相应的预防措施，用户保护用户的私密信息。

对未来工作的展望

本文通过对 AES 进行攻击并成功获取密钥来证明本文设计的合理性以及 Cache 攻击的威力，以引起开发人员和安全工作人员的重视。然而本文实现的攻击也有一些局限性，需要进一步的工作解决。

1) 本文的实验主要针对的是 Lenovo k51c78 目标机，计时方式、驱逐策略以及同步的 Cache 攻击都是在目标机平台上进行实验的。由于不同移动设备或不同的系统版本平台上实验结果会有不同，比如在 Lenovo k51c78 平台上POSIX提供的接口是几种可用的时间源中最准确的，perf提供的接口获取的时间误差较大，而在三星 S5 设备上perf工具是最精确的计时方式。由于不同设备的 Cache 结构通常也不相同，Cache set 数以及每个 set 中的 line 都有可能不相同，驱逐策略也不相同，因此本文中的实验过程不能直接迁移到其他设备上去。

2) 本文针对目标机找到了精确的时间度量方式、有效的 Cache 驱逐策略之后，实现了针对 AES 的同步攻击。并假设攻击程序与 AES 加密算法之间有交互，知道被加密的明文，能够触发 AES 加密程序，能够在 AES 加密前以及加密后执行代码，并且知道 AES 索引表在内存中的分布情况。然而在真实的使用环境中，攻击程序并不了解这么多关于被攻击程序的信息。本文在目标机上对 AES 加密程序进行第一轮异步攻击，但攻击结果不太理想，还需进一步工作。

3) 本文实验基于的目标机系统为 Android5.0.1，然而在 Android6.0.1 以后的版本中限制了非特权模式下对/proc/＜pid ＞/pagemap文件的访问，因此攻击程序不能通过该文件直接计算出内存中虚拟地址与物理地址的映射关系，也就不能计算出内存中的某一虚拟地址被 cache 缓存的 set 号，也就没法通过本文所描述的方式进行攻击。因此，在更高版本的设备上，需要考虑其他方式来获取虚存与 cache set 的映射方式。

参考文献

[1] O. Acıiçmez, W. Schindler, Çetin K. Koç, Cache Based Remote Timing Attack on the AES[M]// Topics in Cryptology – CT - RSA 2007. Springer Berlin Heidelberg, 2007:271-286.

[2] O. Aciicmez, Advances in side-channel cryptanalysis : microarchitectural attacks[J]. 2006.

[3] D. J. Bernstein. Cache-timing attacks on AES[J], Vlsi Design IEEE Computer Society, 2005, 51(2):218 - 221.

[4] A. Bogdanov, T. Eisenbarth, C. Paar, 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.

[5] J. Bonneau, I. Mironov, Cache-collision timing attacks against AES[J]. 2006, 4249:201-215.

[6] A. Carlisle, Constructing of Symmetric ciphers using the CAST design Procedure. In: Designs, Codes, and Cryptography 12,1997:283-316.

[7] P. Dan, Theoretical Use of Cache Memory as a Cryptanalytic Side-Channel.[J]. Journal of Arid Environments, 2002(10):393-446.

[8] J. F. Gallais, I. Kizhvatov, 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.

[9] J. F. Gallais, I. Kizhvatov, Error-Tolerance in Trace-Driven Cache Collision Attacks[J]. Cosade, 2011:222--232.

[10] D. Gruss, C. Maurice, S. Mangard, Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript[M]// Detection of Intrusions and Malware, and Vulnerability Assessment. 2016.

[11] D. Gruss, C. Maurice, K. Wagner, et al, Flush+Flush: A Fast and Stealthy Cache Attack[M]// Detection of Intrusions and Malware, and Vulnerability Assessment. Springer International Publishing, 2016.

[12] D. Gruss, R. Spreitzer, S. Mangard, Cache template attacks: automating attacks on inclusive last-level caches[C]// Usenix Conference on Security Symposium. USENIX Association, 2015:897-912.

[13] D. Gullasch, E. Bangerter, S. Krenn, Cache Games -- Bringing Access-Based Cache Attacks on AES to Practice[J]. 2010(1):490-505.

[14] B. Gülmezoğlu, M. S. İnci, G. Irazoqui, et al, A Faster and More Realistic Flush+Reload, Attack on AES[J]. 2015.

[15] W. M. Hu, Lattice scheduling and covert channels, in IEEE Symposium on Security and Privacy ,1992:52–61

[16] G. Irazoqui, T. Eisenbarth, B. Sunar, Cross Processor Cache Attacks[C]// The, ACM. 2016:353-364.

[17] G. Irazoqui, T. Eisenbarth, B. Sunar, S$A: A Shared Cache Attack That Works across Cores and Defies VM Sandboxing -- and Its Application to AES[J]. 2015:591-604.

[18] G. Irazoqui, M. S. Inci, Eisenbarth T, et al, Know Thy Neighbor: Crypto Library Detection in Cloud[J]. Proceedings on Privacy Enhancing Technologies, 2015, 1(1):25-40.

[19] G. Irazoqui, M. S. Inci, T. Eisenbarth, et al, Lucky 13 Strikes Back[C]// ACM Symposium on Information, Computer and Communications Security. ACM, 2015.

[20] J. Kelsey, B. Schneier, D. Wagner, 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.

[21] F. Koeune, J.-J. Quisquater, A timing attack against Rijndael. Technical Report CG-1999/1, Université catholique de Louvain, http://www.dice.ucl.ac.be/crypto/tech_reports/CG1999_1.ps.gz,1999

[22] P. C. Kocher, 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.

[23] M. Lipp, D. Gruss, R. Spreitzer, et al, ARMageddon: Cache Attacks on Mobile Devices[J]. Mundo Electrónico, 2016, 6(1): 60-65.

[24] F. Liu, Y. Yarom, Q. Ge, et al, Last-Level Cache Side-Channel Attacks are Practical[J]. IEEE Symposium on Security & Privacy, 2015:605-622.

[25] C. Maurice, C. Neumann, O. Heen, et al, C5: Cross-Cores Cache Covert Channel[M]// Detection of Intrusions and Malware, and Vulnerability Assessment. 2015:46-64.

[26] M. Neve, J. P. Seifert, Z. Wang, A refined look at Bernstein's AES side-channel analysis[C]// ACM Symposium on Information, Computer and Communications Security. ACM, 2006:369-369.

[27] M. Neve, J. P. Seifert, Advances on Access-Driven Cache Attacks on AES[C]// Selected Areas in Cryptography, International Workshop, Montreal, Canada, August 17-18, 2006:147-162.

[28] M. Neve, Cache-based Vulnerabilities and SPAM analysis[J]. Doctor Thesis Ucl, 2006.

[29] OpenSSL the open-source toolkit for SSL/TLS. http://www.openssl.org/, 2005

[30] Y. Oren, V. P. Kemerlis, S. Sethumadhavan, et al, The Spy in the Sandbox: Practical Cache Attacks in JavaScript and their Implications[J]. Computer Science, 2015.

[31] D. A. Osvik, A. Shamir, E. Tromer, Cache Attacks and Countermeasures: The Case of AES[J]. Lecture Notes in Computer Science, 2005:1-20.

[32] D. Page, Theoretical use of cache memory as a cryptanalytic side-channel, Department of Computer Science, University of Bristol. http://www.cs.bris.ac.uk/Publications/pub_info.jsp?id=1000625, Technical Report CSTR-02-003,2002

[33] C. Percival, Cache missing for fun and profit[J]. Proc of Bsdcan, 2005.

[34] B. Schneier. Description of a New Variable-Length Key, 64-Bit Block Cipher (Blowfish). In: Fast Software Encryption. Lecture Notes in Computer Science. Cambridge Security Workshop Proceedings.1994:191–204.

[35] R. Spreitzer, T. Plos, Cache-Access Pattern Attack on Disaligned AES T-Tables[C]// Constructive Side-Channel Analysis and Secure Design. 2013:200-214.

[36] R. Spreitzer, T. Plos, On the Applicability of Time-Driven Cache Attacks on Mobile Devices[C]// Network and System Security. 2013:656-662.

[37] E. Tromer, D. A. Osvik, A. Shamir, Efficient Cache Attacks on AES, and Countermeasures[J]. Journal of Cryptology, 2010, 23(1):37-71.

[38] Y. Tsunoo, T. Saito, T. Suzaki, et al, Cryptanalysis of DES Implemented on Computers with Cache.[J]. Proc of Ches Springer Lncs, 2003, 2779:62-76.

[39] M. Weiß, B. Heinz, F. Stumpf, A Cache Timing Attack on AES in Virtualization Environments[M]// Financial Cryptography and Data Security. Springer Berlin Heidelberg, 2012:314-328.

[40] Y. Yarom, K. Falkner, FLUSH+RELOAD: a high resolution, low noise, L3 cache side-channel attack[C]// USENIX Security Symposium. 2014:719-732.

[41] 唐烨. 针对 AES 加密算法的缓存攻击研究[D].中国科学院深圳先进技术研究院,2016.

[42] 邓柳于勤,陈财森,蔡红柳,薛廷梅,于茜.基于 ARM 处理器 Cache 特征的计时分析研究[J].四川兵工学报,2015,36(11):118-121+124.

[43] 赵新杰,王韬,郭世泽,刘会英.分组密码 Cache 攻击技术研究[J].计算机研究与发展,2012,49(03):453-468.

[44] 赵新杰,王韬,郭世泽,郑媛媛.AES访问驱动Cache计时攻击[J].软件学报,2011,22(03):572-591.

[45] 赵新杰,王韬,矫文成,郑媛媛,陈财森.一种新的针对 AES 的访问驱动 Cache 攻击[J].小型微型计算机系统,2009,30(04):797-800.

攻读硕士学位期间得到的学术成果

[1] Bo Li, Bo Jiang. Cache Attack on AES for Android Smartphone* [A].ICCSP,2018.
[2] 李勃. 基于 ARM 处理器的 AES 缓存攻击技术研究[J]. 软件工程与应用, 2018, 7(1): 1-12.

致 谢

转眼间，研究生 2.5 年生涯就快结束了，回首才发现这 2 年多过的是在是太块，仿佛昨天才刚到 G1046 接收姜老师的面试，然而看到实验室新来的师弟师妹们，才意识到自己最后的学生生涯就要过完了。在这两年间，自己学到了很多知识，从一开始的打字都得看着键盘找着一个又一个的英文字母到现在老司机似的指尖在键盘上穿梭，从一开始的简单的算法都得照着模板敲到现在已经能够独自设计一些复杂的程序；从一开始的什么都得老师指导到现在遇到问题知道自己想思路自己解决，也深刻的意识到在姜老师的指导下，自己是在一步一步前进。所以很庆幸自己大四考研的时候再辛苦再累都坚持下来，庆幸自己考上了计算机系研究生，也庆幸自己遇到姜老师，因此才能来到 LES 大家庭。

首先要衷心的感谢姜老师，感谢他在我硕士研究生期间对我无私的指导，在我对研究方向还不太了解的时候，耐心的给我讲解，并指导我学习的方向，感谢姜老师对我的督促，在研究生期间，很多次想要懈怠，是姜老师不断的督促我，才能使我不断的成长。而且不仅仅在学习中，在生活中姜老师和蔼可亲，平易近人，热爱生活的生活态度同样也感染着我，让我不断的成长。在研究生期间，姜老师不仅仅负责我们学习方面的指导，也负责管理 LES 实验室，他勤勤恳恳的态度深深的感染着我，也督促着我成为一个像他一样勤恳、脚踏实地的人。

其次也要感谢万寒老师，感谢她对实验室的管理工作，给我们营造了一个良好的学习生活环境。万老师不但人长得好看，也教会我们爱护工作环境从我做起，并为实验室的干净整洁付出了很多心血。

感谢和我一起工作学习的燕保跃、何一辉、柳俊杰、张耀月、丁军、于雷、刘文博，感谢他们这两年间在我学习和生活中给予的帮助、鼓励和关怀，正是在他们的鼓励下才能使我顺利完成这个课题。同时，也要感谢已经毕业的师兄师姐，他们在实验室的优秀表现为我提供了榜样，在学习生活中也给与了我很多帮助和指导，同样，也要感谢实验室的师弟师妹们，他们给实验室注入了新的血液，他们的努力、开朗一直都感染着我，给我的科研生活增添了很多色彩。

特别要感谢的是我的父母，感谢他们 25 年来对我无微不至的关怀，感谢他们的养

育之恩。在我求学的这些年间，一旦有需求他们都会无条件的支持我，给与我心灵上的慰藉。感谢他们对我的付出！

最后，感谢各位老师百忙之中抽出时间审查我的论文，谢谢。