班　级　1803012
学　号　18030100178

西安电子科技大学

本科毕业设计论文

题　　目　Arm Trustzone 侧信道脆弱性研究与攻击实现

学　　院　计算机科学与技术学院

专　　业　计算机科学与技术

学生姓名　何开颜

校外导师姓名　谷大武

校内导师姓名　杨利英

毕业设计（论文）诚信声明书

本人声明：本人所提交的毕业论文《Arm Trustzone 侧信道脆弱性研究与攻击实现》是本人在指导教师指导下独立研究、写作成果，论文中所引用他人的无论以何种方式发布的文字、研究成果，均在论文中加以说明；有关教师、同学和其他人员对本文本的写作、修订提出过并为我在论文中加以采纳的意见、建议，均已在我的致谢辞中加以说明并深致谢意。

本文和资料若有不实之处，本人承担一切相关责任。

论文作者：_____（签字） 时间：　　年　　月　　日

指导教师已阅：_____（签字） 时间：　　年　　月　　日

摘　要

随着智能设备越来越多地融入我们的日常生活，这些设备的安全性已经成为一个重要的问题。ARM 处理器系列推出了 TrustZone 技术，通过称为安全世界的隔离执行环境来为普通世界提供安全服务或操作。启用了 TrustZone 的处理器中的高速缓存被扩展了一个 NS 位来标识一个缓存行是由安全世界还是正常世界使用。这种高速缓存的设计提高了系统性能，因为它避免了在两个世界切换期间的缓存刷新操作。然而，这样的设计也带来了两个世界的缓存争夺及缓存驱逐现象，为本文的攻击提供基本条件。

本文实现了一种基于时间的 TrustZone 缓存侧信道信息泄漏攻击，该攻击方法利用了普通世界和安全世界之间的高速缓存争夺，以恢复一些安全世界中本无法获取的重要数据或信息。本文的攻击目标是运行在 TrustZone 的 AES 加密程序中的密钥，利用加密过程中的 Te 表的地址与 L1D 缓存之间的映射，使用 Prime+Probe 缓存攻击方法来判断加密程序在执行过程中是否使用到了某些 Te 表项，从而根据获得的密文推断出最后一轮密钥，以此恢复出初始的 16 字节密钥。

本文的攻击方法利用 Arm 处理器内部的性能监测单元作为探测数据加载的 CPU 周期的工具，在普通世界的程序中实现了基于 L1D cache 的安全世界程序的侧信道攻击，并且成功在平均 801.07 次 AES 加密内提取出 AES 的加密密钥，平均耗时 0.75 秒。同时，该攻击方法基于支持 TrustZone 的 ARM 处理器的缓存设计，因此它对相同架构的设备均有潜在的威胁，故本文同样提出了一些缓存攻击对抗方法以及类似领域在未来的潜在发展方向。

关键词：Arm TrustZone　Prime+Probe 攻击　OpenSSL AES　高速缓存

Abstract

With intelligent devices progressively embedded into our lives, their security has become a major concern. TrustZone technology was introduced with the ARM processor series, which provides secure services or operations to the normal world through an isolated execution environment called the secure world. In a TrustZone-enabled CPU, a NS bit is added to the cache to determine if a cache line is utilized by the secure or the normal world. This cache architecture enhances system performance by avoiding cache flush operations during the changeover between the two worlds. However, such a design also introduces the phenomenon of cache contention and cache eviction between the two worlds, providing the basic conditions for the attack in this thesis.

We implement a time-based TrustZone cache side channel attack, which takes advantage of cache contention between the two world, in order to retrieve essential data or information that would otherwise be unreachable in the secure world. The attack in this thesis targets the key in the AES encryption program running in TrustZone, and uses the mapping between the address of the Te table during encryption and the L1D cache. Based on this, the thesis uses the Prime+Probe cache attack to determine whether the encryption program uses some Te table entries during execution. Thus, we can infer the last round of key based on the obtained ciphertext, and then recover the initial key.

The attack method in this thesis uses the performance monitoring unit inside the Arm processor as a tool to detect the CPU cycles of data loading, and implements the L1D cache-based side-channel attack on a secure world program. The experiment can successfully extract the AES encryption key within an average of 801.07 rounds encryption and 0.75 seconds. At the same time, because the attack approach is based on the cache design of TrustZone enabled ARM processors, it poses a risk to devices with similar architecture. Lastly, this thesis also proposes some cache attack countermeasures and

potential future directions in similar areas.

Key words: Arm TrustZone Prime+Probe attack OpenSSL AES Cache

目 录

第一章 绪论 **1**
1.1 研究背景及意义 1
1.2 国内外研究现状 2
1.3 研究内容和组织框架 4
1.3.1 研究内容 4
1.3.2 组织框架 4
第二章 关键技术 **7**
2.1 ARM TrustZone 7
2.2 处理器内存层次架构 8
2.2.1 内存与 Cache 的映射关系 8
2.2.2 Cache 的驱逐与替换 10
2.3 缓存侧信道攻击 11
2.3.1 现有的 cache 攻击手段 11
2.3.2 针对 TrustZone 的攻击方法 13
第三章 攻击模型及假设 **15**
第四章 攻击对象 **17**
4.1 OpenSSL AES 17
4.2 最后一轮密钥的攻击 21
4.3 Neve&Seifert 排除法 23
第五章 攻击方案 **25**
5.1 分配内存和缓存驱逐 25
5.2 周期计数器探测缓存 26
5.3 普通世界与安全世界交互设计 27
5.4 攻击算法流程 30

第六章 实验与分析 31
6.1 攻击结果与性能分析 31
6.2 cache 攻击应对方法 35
第七章 总结与展望 37
7.1 方案总结 37
7.2 未来工作 37
致谢 39
参考文献 41

第一章　绪论

1.1　研究背景及意义

随着智能设备的普及，设备的安全问题也受到了更多的关注。可信执行环境(Trusted Execution Environment，TEE) 是由 Global Platform 所提供的定义, 通过在中央处理设备中建立一套安全性范围, 以确保其内的应用和数据信息在秘密性和完全性上得到保障。可信执行环境的原理是, 把操作系统的硬件和软件资源区分为两种运行环境——即可信执行环境 (TEE) 和普通执行环境 (REE)。这两种环境都是相互安全隔离并且共存的运算环境, 都具有各自单独的内存数据通路和计算所需要的空间。TEE 为 REE 提供了部分安全服务, 如指纹认证、支付、数字认证等，而且有它自己的执行环境, 安全等级也较普通环境中的操作系统高。同时，普通运行环境的应用程序也无法访问 TEE, 原因就是在 TEE 里面, 各个应用的程序也是彼此独立的, 所以不能因为无权限而互访。TEE 也提供了安全性与成本上的均衡, 可以满足大部分应用的安全性要求。与多方安全计算和联邦学习这些纯软件方面的解决方案比较,TEE 可以支撑更多的算子和复杂运算, 还可以进行关联运算、协同查询、联合建模和预测等各种运算, 业务表达性更强。同时,TEE 支持多层次、高度复杂的算法逻辑实现, 计算效率极高。

TrustZone 是 ARM 处理器系列提出的一种 TEE 技术，该安全扩展提供了在一个隔离的执行环境中保护安全敏感数据的功能。TrustZone 已被广泛地应用于各种商业产品和学术项目[1–4]，以实现安全处理。受保护的环境被称为安全世界，而普通的操作系统运行环境被称为普通世界或不安全的世界。原理上，一个处理器上可以运行两个独立的操作系统，一个是运行在普通执行环境上的 Linux，另外一个就是运行在可信执行环境之上的小内核 (kernel)，它结构简单，代码少，攻击面小，从而安全性得到了提高。由于 ARM 架构确保了许多硬件系统资源都是双份的, 因此每个世界中的虚拟中心可以独享自身的那份资源, 大大简化了设计。

与别的资源不同的是，cache 是安全世界与普通世界共用的。为了区分，启用

TrustZone 的处理器的高速缓存行另外扩展了一个标志 (NS) 位，以指示当前的高速缓存行的作用域。安全世界对于 cache 使用的权限要远远大于普通世界。对于普通世界，处理器会无视这个 NS 标志位，对于安全世界，处理器需要按照对应的 NS 位，以安全世界或者普通世界的身份进行地址转换。尽管安全世界的缓存行不能被普通世界访问，但在争夺缓存行的时候，两个世界是平等的。当处理器在一个世界运行时，如果目标缓存组已满，它可以驱逐另一个世界使用的缓存行。这种缓存设计的存在，使得两个环境切换期间，不需要重新刷新高速缓存，以此可以提高系统性能。同时，两个世界对这些高速缓存的访问模式是类似且不受保护的，这使 TrustZone 很容易受到高速缓存侧信道攻击，具体来说，攻击者虽然无法直接获得 TrustZone 内程序执行的过程，但是通过间接获取缓存的数据变化情况，获得 TrustZone 中的安全应用的数据信息，从而获取一些秘密值。

1.2 国内外研究现状

传统的密码分析侧重于算法层面，利用算法在执行上的漏洞进行密码破解。而侧信道攻击利用了加密算法在执行时，软件或硬件泄露的信息，进行侧面信息提取，从而进行重要信息的破解。

侧信道攻击在计算机安全领域有较广泛的应用[5,6]，同时也有许多防御手段[7]。随着对缓存的大量研究，缓存侧信道攻击在学术界也吸引了众多学者的视线。使用侧信道信息作为攻击加密方案的手段的概念首次出现在[6]的一篇开创性论文中。在[6]中，Kocher 利用计算时间的差异来破解 RSA 和基于离散对数的加密算法的实现。除了时间，其他物理属性，如电磁辐射[8]、功耗[5]或声学噪音[9]也被研究为可行的侧信道来源。Bernstein[10]是第一个表明由数据缓存，会存在时间依赖性的学者，这使得 AES[10]的 T table 实现能够被攻击并恢复出密钥。基于缓存的侧信道攻击有三个类别：时间驱动[10]，跟踪驱动[11]和访问驱动[12]。它们之间的区别在于攻击者的能力，其中时间驱动性攻击限制最少。

Osvik 等人[13]提出了两种技术供攻击者确定受害者访问的缓存集，即 evict+time 和 prime + probe。在 evict + time 中，攻击者修改一个已知的缓存集，并观察受害者的加密操作的执行时间的变化。在 prime + probe 中，攻击者在执行加密操作之前用已知的状态填充缓存，并在执行完加密操作后，观察这些缓存状态的变化。Gullasch 等[14]确定了另一个由系统内存删除而得来的缓存侧信道攻击。攻击者在启用 KSM

的情况下，冲刷恶意进程和受害者进程之间共享的内存，如加密库。在受害者执行加密算法后，攻击者测量将内存载入寄存器的时间，以确定该内存是否被受害者进程访问过。后来 Yarom 等人在[15] 中将这种缓存攻击方法命名为 flush+reload。国内学者也对该缓存侧信道攻击方法进行了扩展，研究了差分 flush+reload 的攻击方法[16]。随着人们对云计算的兴趣越来越大，另一个研究方向[17,18] 专注于从虚拟机而不是本地机器的进程中恢复秘密。

目前的研究大多集中在对英特尔 x86 架构处理器的侧信道调查上，但很少涉及 ARM 平台。Weiss 等人[19] 首次在 ARM 处理器中测试了 Bernstein 攻击。除了 Bernstein 的定时攻击，在[20] 中使用了 evict+time 来攻击 AES 的 T 表实现。在[21] 中还使用了内存复制来发起 flush+reload 攻击，以跟踪共享库的执行路径。Lipp 等人[20] 针对 ARM 处理器，利用 Java 来实现缓存攻击以恢复加密密钥，同时，他们能够在 TrustZone 中监视缓存活动。与[20,21] 相比，在此篇文章中，我们的研究目标是不同的。由于 TrustZone 的内存分离保护，普通世界和安全世界在执行时的内存是分离的，而 evict + reload[20] 或 flush + reload[21] 都需要基于内存共享的前提，在本课题中，无法使用上述两种方法。故在本文的研究中，我们可以使用 prime+probe 方法对缓存进行信息提取并攻击加密密钥，在[20] 中作者简要地讨论了 prime+probe 的可行性。在[22] 中，Zhang 等人针对 ARM TrustZone 环境，提出了第一个基于时间的侧信道攻击。除了缓存定时侧信道，最近的一项研究证明了一个新的缓存侧信道攻击，基于缓存不连贯性的意外缓存命中[23]，该攻击可以从 Cortex-A7 处理器的安全世界中恢复秘密值。

基于[22] 中的对缓存的时间侧信道攻击，我们同样针对 ARM TrustZone 环境，进行缓存的攻击实现，同时尽可能提升攻击效率和准确度。在我们的攻击方案中，利用了操作系统 sudo 权限，触发安全世界中的加密执行，并且探测共享的缓存在加密前后的数据变化。本文所利用的缓存时间侧信道攻击是基于 ARM TrustZone 的特性进行设计的。TrustZone 在缓存中添加一个额外的标志位来标识当前的缓存行是安全世界还是普通世界是使用的，而不采用缓存分离机制。任意一个世界在执行程序时，如果数据地址对应的缓存组已经被占满，就会遵循一定策略实行缓存驱逐，这时候缓存的数据状态就会发生改变，攻击者端使用一定的探测方法就能知道受害者使用的具体数据，为本文研究的缓存攻击提供了一定的先决条件。

1.3 研究内容和组织框架

1.3.1 研究内容

我们在 Cortex A-53 处理器的 Raspberry Pi 3B 开发板上实现了 TrustZone 缓存侧信道攻击，以一个基于 T-表的 AES 实现为例，展示了 TrustZone 的侧信道信息泄漏。来自正常世界的攻击可以观察多次加密过程中缓存的数据变化，并且恢复完整的 AES128 的 16 位密钥，本文工作可以归纳为以下三点：

(1) 基于 TrustZone 中普通世界和安全世界之间的缓存设计架构，成功识别并利用了缓存的侧信道信息，该架构允许两个世界共享相同的缓存来提高系统性能。

(2) Prime+Probe 是 Intel 架构上采用的较广泛的缓存攻击方法，本文详细介绍了该技术在 ARM 处理器上的采用，其中有许多难点，例如缓存替换策略的随机性、无法中断加密过程、缺少 rdtsc 指令直接获取 cache miss 次数以及 CA 和 TA 交互过程设计。

(3) 在得到缓存数据信息变化后，利用不同位置（如缓存、内存）的数据加载在时间上的差异性，结合缓存侧信道可行性分析，得到加密过程中的加密密钥。

1.3.2 组织框架

本文的结构安排如下：

第一章: 阐述该课题的科研背景、重要性以及国内状况, 并提出本文的科研内容与组织架构。

第二章：介绍本课题涉及的关键技术领域，ARM TrustZone 的运行机制、内存结构在程序运行时对地址的加载以及现有的缓存侧信道攻击方法。

第三章：介绍本课题利用的攻击模型及合理假设。

第四章：阐述本课题攻击的对象，即 OpenSSL 的 AES128 加密过程，以及如何利用缓存配置和加密密文攻击密钥的原理。

第五章：具体分析并阐述本课题的实验方法。

第六章：对上述实验方法进行具体实验，以验证方法的可行性和算法的效果，并介绍应对缓存攻击的可能的方法。

第七章: 对本课题所采取的攻击方法加以总结, 并同时给出了对今后工作方向的展望。

第二章　关键技术

2.1　ARM TrustZone

ARM TrustZone 是基于硬件的安全功能, 它通过对原有硬件结构进行了调整, 并在处理器层次引入了两种不同权限的工作域——安全世界和普通世界, 任何时刻处理器都只在其中的一种环境内工作。而且由于这二种世界之间全部是以硬件分隔的, 并且有不同的权限, 在普通世界中运作的软件访问安全世界中的信息或资源是被严格控制的, 反过来在安全世界中运行的应用能够正常访问普通世界中的数据。

一旦处理器的操作状态被划定，普通世界状态下的 Linux 即使以 root 身份进行操作，也不能访问安全世界状态下的任何资源，包括设备、内存数据、获取缓存数据等。无论外部环境是否安全，安全世界内的执行都是安全的。这是因为当 CPU 访问安全设备或安全存储器地址空间时，芯片级安全扩展组件检查 CPU 发送的访问请求的读/写安全状态信号（非安全位，NS 位）是否等于 0 或 1，以确定 CPU 发送的当前资源访问请求是安全请求还是普通请求。当普通世界的 CPU 发送系统总线访问指令时，访问请求的读/写安全状态信号位被强制为 1，表明当前 CPU 的访问请求是普通世界的操作请求。当普通请求试图访问一个安全世界的资源时，它将被安全扩展组件视为不安全的请求，对受保护资源的访问将被禁止。

两个世界间的硬件隔离以及不同的权限等功能,保障了正常应用程序代码与数据的有效机制：普通世界（Normal World）通常用于运行操作系统（如安卓、iOS 等），提供普通运行环境（Rich Execution Environment，REE）；安全世界（Secure World）总是使用安全环境或者安全内核（TEE-kernel），建立可信的执行环境（Trusted Execution Environment，TEE），可以存储和访问敏感数据。这样一来，即使正常情况下的操作系统被损坏（如 iOS 被黑客攻击或 Android 系统被获得 ROOT 权限），黑客仍然无法访问存储在 TEE 中的敏感数据与重要信息。

图2.1描述了 Cortex-A 上采用的 TrustZone 架构。普通世界（或非安全世界）与

安全世界之间，存在着一种监视模式（Monitor mode）的处理器模式，该模式负责在两个世界过渡时保留处理器状态。普通世界中可以通过调用安全监视器（SMC）的特权指令进入监视模式；此时执行在监视模式的代码首先保存在普通世界中的 CPU 上下文, 将 CPU 中的 NS 位置为零, 然后进入安全世界的特权模式；安全世界的操作系统唤醒安全世界中的应用处理完相应的资源访问请求后，也会发送 SMC 指令，再次回到监视模式；运行在监视模式的代码同样保存安全状态下的 CPU 上下文，再重新回到普通世界特权模式。

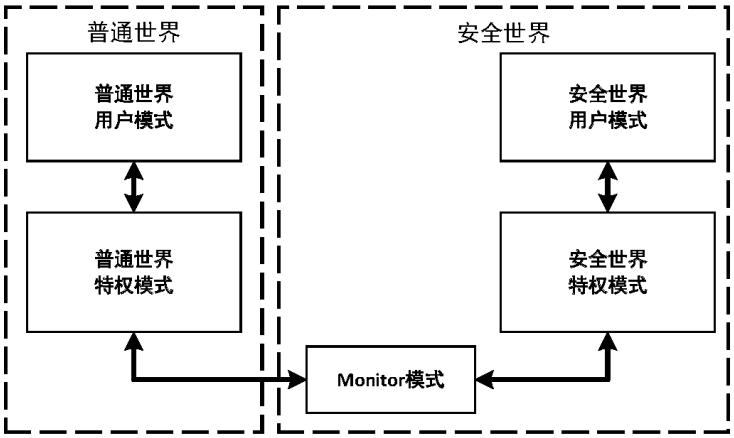

图 2.1　Cortex-A 上的 TrustZone 架构

2.2　处理器内存层次架构

本文的攻击环境是基于 ARMv7l 架构的 Cortex-A53 处理器搭建的，故在本节重点介绍 Cortex-A53 处理器的内存层次架构。

2.2.1　内存与 Cache 的映射关系

现代计算机系统中，程序在虚拟地址空间中运行，当处理器需要在内存中获取一个值时，它将虚拟地址传递给内存管理单元（MMU），MMU 首先从旁路快表缓冲（TLB）中检索虚拟地址到物理地址的转换。若没有找到对应项，MMU 就解析页表，并将解析得到的物理地址放入 TLB 中，以提高下一次同一地址的查询速度。一旦获得物理地址，CPU 就通过访问系统内存来获得该地址的值。

现代处理器通常有多级缓存结构，以实现更快的内存访问。缓存的层次越高，速度越快，并且越靠近核心处理器。在上述 CPU 通过访问内存来获取数据前，缓

存控制器会先检查要获得的数据是否在缓存中，若在缓存中，则 CPU 可以直接从缓存中获取数据，速度较快，这称为一次缓存命中（cache hit）。若不在缓存中，则需要从内存中把数据先加载到缓存中，再加载至 CPU 寄存器中，速度较慢，这个过程称为缓存未命中（cache miss）。与物理内存相比，缓存的尺寸通常较小，其中的数据只占内存的一小部分。缓存命中率越高，数据载入时间就越少，整体运行速度就越快。Cortex-A53 拥有 4 个核，可以查看核的 Online 或 Offline 状态，整体是两级缓存配置，每个核都有可配置的 16−64KB 的 L1 指令缓存（L1i Cache），16−64KB 的 L1 数据缓存（L1d Cache），并且这 1−4 个处理器之间有共享的 128KB−2MB 的 L2 缓存。

缓存中内存分配的基本单位为缓存项，一个缓存项由标签、缓存行和若干标志位组成，每个缓存行一般为 64 字节。多个路的缓存项组成一个缓存组，多个组再组成整个缓存。高速缓存通常采用 N 路组相联映射。在本文所研究的环境中，每个缓存行大小均为 64 字节，L1 数据缓存则由 128 组的 4 路缓存项构成，即大小为 32KB。第二级缓存由 16 路组相联构成，共有 512 组，每个缓存行大小也为 64KB。由于 TrustZone 的加入，每个缓存行都被扩展了 NS 位，它指定了该缓存行的状态，这个设计的目的是使用 NS 位来区分当前这个缓存行是安全世界的还是普通世界的数据，世界切换时则不需要刷新缓存，以此提高运行效率。

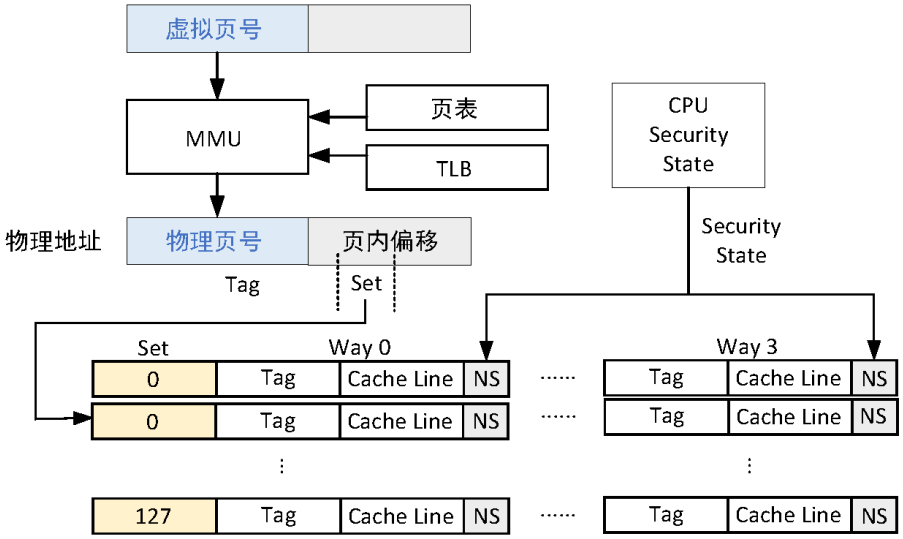

图 2.2　L1D Cache 抽象图以及内存层级结构

由于缓存容量相对内存来说非常小，仅能存储内存中一部分数据，故需要构建物理地址到缓存的映射，以确认目标缓存中的数据是否为目标物理地址的数据。

由于我们的主要攻击对象为数据缓存，图2.2为 L1D Cache 的抽象图以及整体的内存层级结构。对于一个 32 位的地址来说，第 0-5 比特位为缓存行内的偏移，第 6-12 这 7 个比特位为缓存组的映射，匹配相应的缓存组后，用 13-31 比特位匹配这个缓存组中的 4 个缓存项的标签，若有能匹配上的，这个缓存行的内容即目标内存的内容，可以直接加载进寄存器进行数据操作。若没有能够匹配上的标签，需要去 L2 或者内存中查询数据，速度相对慢很多。

2.2.2 Cache 的驱逐与替换

上述提到，缓存的大小要远远小于内存大小，当发现目标数据此时还没有被加载到 L1 数据缓存中，缓存控制器会去 L2 缓存中寻找目标数据。若此时仍未找到，处理器会从 RAM 中取出数据，并把数据放到缓存中，此时若物理地址对应的缓存组已满，必定会驱逐之前已经填到该缓存组中的某个缓存行。这就是缓存驱逐。同时，不管是安全世界还是普通世界的缓存行，在任意一个世界执行时都有可能被驱逐，以便为新的数据腾出空间。换句话说，安全缓存行的填充有可能会驱逐普通缓存行，反之亦然。这种设计是对 TrustZone 中受保护的执行进行侧信道攻击的关键。

在进行缓存替换时，CPU 对 32 位物理地址进行拆分，先计算这个地址的缓存组映射，然后在组内使用缓存替换算法，从这个缓存组的 4 个缓存项中挑出一个，把原有的数据驱逐，放入更新的数据，其中，需要更新 64 字节的缓存行部分，19 比特的标签部分，以及 NS 位等。

在现代的 ARM 架构中，内存系统通常有多个层次，缓存是最高级别的，且缓存通常分为多个级别。许多 ARM 处理器中的高速缓存既不是包容性的，也不是排他性的，也就是说最高级别缓存中的缓存行不一定会存在低级缓存中。ARM 处理器的缓存控制器的替换策略是随机替换策略。缓存替换策略指的是，当出现高速缓存竞争时，处理器从每组的 n 个（本文中是 4 个）缓存行中如何选择一个驱逐并进行新的缓存替换。随机替换策略给实验新增一个难点，即如何确保当前的数据加载进缓存时，之前的数据没有被驱逐，解决方法将会在第五章攻击方案中介绍。

虽然从内存加载数据和从缓存加载数据的时间差比较明显，但由于缺乏缓存包容性，若在攻击过程中以低级别的缓存作为目标会使分析过程变得比较困难。如果将高级别的缓存作为侧信道攻击的对象，数据加载的时间区别最明显。同时，只

要是处理器访问的内存数据，一定会被先放到 L1 数据缓存中。故在实验中，我们主要以 L1 数据缓存作为观测的对象，以此来捕捉高速缓存争夺中的信息泄露。

Linux 操作系统中，所有程序在运行时采用的都是逻辑地址, 而逻辑地址会被 MMU 变换为物理地址，再用这个物理地址向 RAM 发出内存读写请求，每个内存块的大小为 4KB。故物理地址的最低 12 位为页内偏移。因为 L1D cache 一共有 128 个缓存组，在将逻辑地址转化为物理地址时，页内偏移的 0-11 位是不会改变的，只有第 12 位为 0 或 1，也是攻击者无法完全控制的。故在求缓存组映射时，逻辑地址的第 12 位转化成物理地址后，决定会映射到前 64 还是后 64 个缓存组，加上不变的 6-11 位就可以完全确定映射到的缓存组号。在实验中，由于第 12 位转化成物理地址后是无法获取的，需要对该位为 0 或 1 的情况分别进行确定。而逻辑地址可以直接通过可执行连接文件.elf 或动态链接库.so 通过以下命令（示例中是获得 4 张 Te 表的偏移）查看。

```
1        nm target.elf | grep Te[0-3]
```

2.3　缓存侧信道攻击

2.3.1　现有的 cache 攻击手段

缓存侧信道攻击的本质是通过缓存创建一个通道，通过这个通道可以获得某些隐藏的信息，以检测程序是否访问了某个特定内存地址的数据。例如，攻击者能够在内存（缓存未命中）与缓存访问数据（缓存命中）的时间差值中了解缓存上的信息或数据的变动。下面是几种缓存攻击基本手段。

Flush+Reload:

Intel 处理器提供了 clflush 执行，针对于一个地址进行操作，可以将这个地址的数据从缓存中驱逐。Flush+Reload 方法基于共享内存实现，是一种跨内核的缓存探测方法。第一步，攻击者使用 clflush 指令把目标地址从缓存中驱逐，然后在下一阶段，受害者程序触发执行。最后一步，攻击者访问目标地址的数据，并记录访问时间。若访问过的内存块已经在缓存中而不需要 reload，时间会相对短一些，因为缓存命中的加载时间较短。根据记录的时间结合一定阈值，可以判断受害者程序是否使用过这部分数据。Flush-Reload 具体步骤说明：

- Step 1. Flush: 使用 clflush 指令把目标地址从缓存中驱逐
- Step 2. Trigger：触发目标（受害者）程序执行
- Step 3. Reload：重新访问内存中的那个数据，测量并记录时间

Flush+Flush：

该攻击方法同样是基于 clflush 指令执行时间的长短来实施攻击的。如果数据未在 Cache 中，clflush 指令执行时间会比较短，相反，有数据在 cache 中则执行时间会比较长。与其它 Cache 攻击不同，Flush+Flush 侧信道攻击技术在整个攻击过程中是不需要对内存进行存取的，所以该攻击技术也较为隐秘。Flush+Flush 具体步骤如下：

- Step 1. 通过 clflush 指令驱逐目标地址在缓存中的数据
- Step 2. 触发受害者程序运行
- Step 3. 再次利用 clflush 指令驱逐共享缓存行，测量刷新时间；根据测量时间判断原始数据是否被缓存，如果时间超过一个阈值，则认为此时对应的缓存中是有数据的，那么攻击者被看作访问了该地址的数据。

Evict+Time：

Evict+Time 具体步骤如下：

- Step 1. 攻击者将目标地址的数据填入缓存
- Step 2. 触发受害者程序运行，并记录其运行时间，运行过程有可能使用到第一步中的缓存数据
- Step 3. 使用 Evict 方法驱逐 Cache 上的数据
- Step 4. 然后再运行一次该函数，并第二次记录执行时间，如果时间不一致且执行时间变长则说明程序运行时读取了第一步所说的 Cache 数据（Cache 未命中）。

Prime+Probe：

该方法与上面三种不同的是,不需要共享内存的环境就可以进行攻击。Prime+Probe 方法具体步骤如下：

- Step 1. Prime: 攻击者用预先准备的内存数据填充目标的 cache 组（set）
- Step 2. Trigger: 触发受害者程序的执行，等待它将 cache 数据更新
- Step 3. Probe: 重新遍历 Prime 阶段填充的数据，测量并记录各个 cache 组读取时间，利用时间差来判断当前的数据加载是从内存中还是从高速缓存中直接读取

上述四种缓存侧信道攻击方法中，由于指令 clflush 在 JavaScript 和非 Intel CPU 中无法使用，所以最后一种方法较前几种方法更加常用。但同时，Prime+Probe 在攻击速度、攻击稳定性方面，不如 Flush+Reload 和 Flush+Flush 这两种方法。本文攻击环境是 ARM，故将采用最后一种攻击方法，具体原因及原理将在下一步阐述。

2.3.2　针对 TrustZone 的攻击方法

上述的 evict+time，flush+reload，flush+flush 这三种 cache 攻击方法必须基于共享内存，由于 TrustZone 的内存分离保护，这三种方法都无法使用。故必须使用 prime+probe 这种方法，因为它不需要共享内存的前提，只需要知道目标数据的物理地址或虚拟地址，即可进行缓存组的映射。TrustZone 中的缓存项虽然存在 NS 位来区分是安全世界还是普通世界使用的，但是缓存驱逐时是不考虑现有的 NS 位的，只要对应到目标缓存组，就会在缓存组里进行驱逐与替换。

Prime+Probe 攻击中，需要先构造一个驱逐集。为了在攻击后半部分的 probe 步骤中获得关于受害者进程的秘密信息，攻击者必须在 prime 阶段准确地将特定地址的内存内容填充到目标缓存区域，这样它就会引起与受害者进程的缓存争用。为了构造驱逐集，攻击者需要找到同样会被映射到受害者进程使用的缓存组的内存地址。以一定次数执行这部分内存地址的数据遍历，这些数据就会被填充到目标缓存组。当受害者程序访问了同样映射到这个缓存组的数据时，原本构造的占用整个缓存组的 4 个缓存项将会有一个被驱逐到内存中，新的数据就会替换。再执行一遍 prime 过程遍历的数据，会有某项数据的访问时间明显与其它数据的访问时间不同，这就是从内存访问数据和直接从缓存中访问数据的巨大的时间差造成的。通过比较这个时间阈值，可以得知当前的数据是否在安全世界执行某个任务时用到这个缓存行，这样一来，本不应该被获得的秘密信息就可以通过这个时间被攻击者获取，从而进行一些更深入的探测和攻击。

第三章　攻击模型及假设

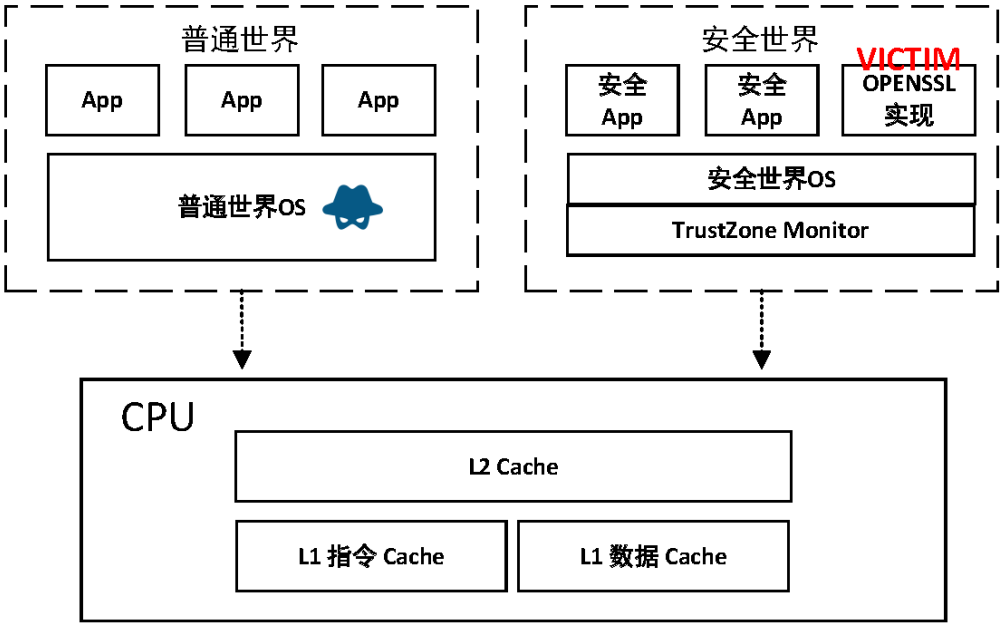

图 3.1　TrustZone cache 攻击模型

本章介绍文章中涉及的攻击方案的模型及假设。

正如 ARM TrustZone 白皮书[24] 所说，TrustZone 通常用来在高度隔离的执行环境中保护涉及安全数据的应用，例如加密操作。假设有一 ARM 架构的设备，其中普通世界内核为 Linux 系统，安全世界为 TrustZone，攻击目标是获取安全世界运行的某个程序中涉及的秘密值，如 AES 加密过程中的 16 位密钥。同时假设加密库在安全世界中得到保护，并且没有受到在普通世界中运行的可能被损坏的操作系统的干扰，也就是说无法被普通世界进行直接攻击。

攻击模型如图3.1所示。假设有一个 openssl 的 aes 加密过程在安全世界实现，并为普通世界的操作系统提供安全服务。普通世界的攻击者拥有操作系统的最高权限，即可以以 sudo 权限执行任意代码，这种假设在对硬件强制执行的可信环境发起攻击时是非常常见的。攻击目标就是安全世界的 openssl 的 AES 加密模块中的 16 位随机初始化的密钥。我们认为攻击者可以触发受害者进程中的加密，同时为受害者进程提供 16 位明文，经过加密后可以获得 16 位密文。同时，攻击者可以

利用内核中的性能监测单元（PMU）模块随时获取执行过程的 CPU 周期数，观察从不同位置加载进寄存器的时间在 CPU 周期上的差异性。

第四章　攻击对象

4.1　OpenSSL AES

高级加密标准（Advanced Encryption Standard, AES），也被叫做 Rijndael 加密法，若按算法流程直接进行加解密操作，可能会有计算量大、耗时长等问题。为解决这些问题，我们通过用简单的查表操作代替一些复杂的过程来降低算法的时间复杂性，以此更好地优化算法。整个加密过程，可概括为字节替代（SubBytes）、行移位（ShiftRows）、列混合（MixColumns）和轮密钥加（AddRoundKey）。其中，本文专注于研究 AES 的 128 位加密算法，并且主要攻击最后一轮的 16 字节密钥。因为密钥计算过程是可逆的，我们可以通过最后一轮的密钥倒推出最初的 16 字节密钥。整体的 AES 加密过程如图4.1所示。

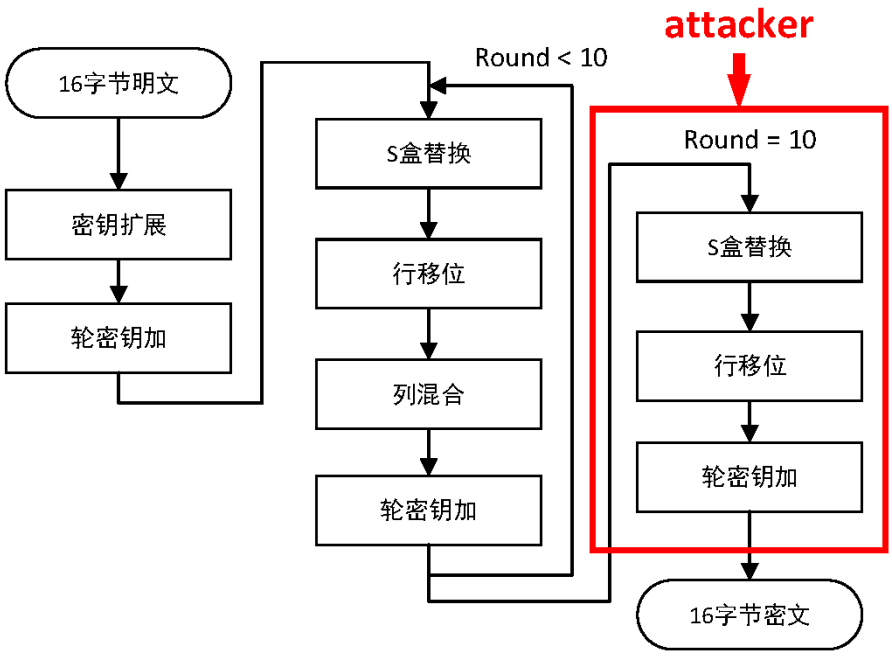

图 4.1　AES 加密过程

下面对 AES 的 T 表概念进行解释。规定以下符号概念：

表 4.1 符号解释

符号	解释
p_{ij}	明文字节
$p_{ij}^{'}$	前一轮输入字节经一轮后生成的中间字节
k_{ij}	轮密钥
c_{ij}	密文字节
$SBox\,[\cdot]$	S 盒查询
$\oplus$	异或
$\|\|$	连接符号
P_j	$p_{1,j}\|\|p_{2,j}\|\|p_{3,j}\|\|p_{4,j}, 0 \le j \le 3$
C_j	$c_{1,j}\|\|c_{2,j}\|\|c_{3,j}\|\|c_{4,j}, 0 \le j \le 3$
K_j	$k_{1,j}\|\|k_{2,j}\|\|k_{3,j}\|\|k_{4,j}, 0 \le j \le 3$
$TBox\,[\cdot]$	TBox
$Te\,[\cdot]$	Openssl 中简化计算后的 T 表

我们将输入的 16 字节明文看作一个 4*4 的矩阵，矩阵的每一格代表一个字节，并且竖着的一列表示连续的 4 个字节。即一个明文输入可以看作 $P_1||P_2||P_3||P_4$，共 16 个字节。

S 盒替换：

对于每一格输入 p_{ij}，把该字节的高 4 位看作行，后 4 位看作列，以行列为索引在 S 盒中找到替换后的字节，$p_{ij}^{'} = SBox\,[p_{ij}]$。

行移位：

左循环移位操作，上一步输出的 4*4 状态矩阵中，第 0 行左移 0 字节，第 1 行左移 1 字节，以此类推，$p_{ij}^{'} = p_{i,i+j}$。

列混合：

通过矩阵相乘的方式实现，通过将行移位后的状态矩阵和固定的矩阵相乘，得到列混合后的新状态矩阵，过程如下式给出。

$$\begin{bmatrix} p_{0,0}^{'} & p_{0,1}^{'} & p_{0,2}^{'} & p_{0,3}^{'} \\ p_{1,0}^{'} & p_{1,1}^{'} & p_{1,2}^{'} & p_{1,3}^{'} \\ p_{2,0}^{'} & p_{2,1}^{'} & p_{2,2}^{'} & p_{2,3}^{'} \\ p_{3,0}^{'} & p_{3,1}^{'} & p_{3,2}^{'} & p_{3,3}^{'} \end{bmatrix} = \begin{bmatrix} 02\ 03\ 01\ 01 \\ 01\ 02\ 03\ 01 \\ 01\ 01\ 02\ 03 \\ 03\ 01\ 01\ 02 \end{bmatrix} \begin{bmatrix} p_{0,0} & p_{0,1} & p_{0,2} & p_{0,3} \\ p_{1,0} & p_{1,1} & p_{1,2} & p_{1,3} \\ p_{2,0} & p_{2,1} & p_{2,2} & p_{2,3} \\ p_{3,0} & p_{3,1} & p_{3,2} & p_{3,3} \end{bmatrix} \tag{4-1}$$

状态矩阵中每一项的列混合结果可以按如下表示，元素下标的加法运算若超出 3 则按模 3 计算：

$$p_{ij}^{'} = [02\ 03\ 01\ 01]\,[p_{i,j}\ p_{i+1,j}\ p_{i+2,j}\ p_{i+3,j}] \tag{4-2}$$

其中，所有矩阵元素的乘法和加法都是定义在基于 GF（256）的二元运算，而不是一般含义上的乘法和加法。矩阵元素的加法可以看作简单的异或操作。将列混合看作列上的操作后，可以得到如下的式子：

$$\begin{aligned} P_j^{'} &= p_{0,j}^{'}||p_{1,j}^{'}||p_{2,j}^{'}||p_{3,j}^{'} \\ &= (2p_{0,j} \oplus 3p_{1,j} \oplus p_{2,j} \oplus p_{3,j})|| \\ &\quad (2p_{1,j} \oplus 3p_{2,j} \oplus p_{3,j} \oplus p_{0,j})|| \\ &\quad (2p_{2,j} \oplus 3p_{3,j} \oplus p_{0,j} \oplus p_{1,j})|| \\ &\quad (2p_{3,j} \oplus 3p_{0,j} \oplus p_{1,j} \oplus p_{2,j}) \end{aligned} \tag{4-3}$$

将其中的异或和连接符号进行顺序调换并不影响计算结果，故顺序调换后，可以将计算过程简化成 4 个 TBox 的查表替换行为:

$$\begin{aligned} TBox_0[x] &= 2x||x||x||3x \\ TBox_1[x] &= 3x||2x||x||x \\ TBox_2[x] &= x||3x||2x||x \\ TBox_3[x] &= x||x||3x||2x \end{aligned} \tag{4-4}$$

此时列混合的步骤可以由如下表示：

$$P_j^{'} = TBox_0\,[p_{0,j}] \oplus TBox_1\,[p_{1,j}] \oplus TBox_2\,[p_{2,j}] \oplus TBox_3[p_{3,j}] \tag{4-5}$$

其中，$0 \le j \le 3$，故列混合操作可以看作对 TBox 的快速查询操作，每一列的变换通过原列中的 4 个字节，取表中拼接而成，得到 4 个字节，作为新一列的输出。其中每个 TBox 有 256 项，每一项为 4 字节。

轮密钥加：

将上述结果的每一项 p_{ij} 与对应密钥进行异或操作，得到这一轮的输出。

以上为 AES 加密的四个步骤，而 OpenSSL 中，上述的四个步骤被整合为新的查找表 $Te_i\,[\cdot]$，其中 $0 \le i \le 3$：

$$Te_i[x] = TBox_i[SBox[x]] \tag{4-6}$$

也就是说，前 9 轮的输出结果都可以看作是一次表的查询操作，有利于计算效率的提高。每一轮的输出可以看作如下表达式：

$$P_j^{'} = Te_0[p_{0,j}] \oplus Te_1[p_{1,j+1}] \oplus Te_2[p_{2,j+2}] \oplus Te_3[p_{3,j+3}] \oplus K_j \tag{4-7}$$

对于最后一轮加密，因为少了列混合的步骤，故最后一轮可以用如下式子表示：

$$p_{i,j}^{'} = SBox[p_{i,i+j}] \oplus k_{i,j} \tag{4-8}$$

因为 Te 表就是由 S 盒经过一定变换得到的，故最后一轮的输出也可以由 Te 表项得到。首先，最后一轮的输出密文可以由如下式子表示：

$$c_{ij} = SBox[p_{i,i+j}] \oplus k_{ij} \tag{4-9}$$

其中，S 盒的内容可以通过 Te 表间接得到，以 Te_0 为例：

$$\begin{aligned} Te_0[x] &= TBox_0[SBox[x]] \\ &= 2SBox[x]||SBox[x]||SBox[x]||3SBox[x] \end{aligned} \tag{4-10}$$

由上式可以看出，$SBox[x]$ 可以由 $Te_0[x]$ 的倒数第二个字节提取得到，由此类推，$SBox[x]$ 可以由剩下的三个 Te 表项提取对应的字节位数得到。用以下四个公式可以看出 $SBox[x]$ 与四个 Te 表的对应关系：

$$\begin{aligned} SBox[x] &= Te_0[x] >> 8\&0\text{xff} \\ SBox[x] &= Te_1[x] >> 0\&0\text{xff} \\ SBox[x] &= Te_2[x] >> 24\&0\text{xff} \\ SBox[x] &= Te_3[x] >> 16\&0\text{xff} \end{aligned} \tag{4-11}$$

替换后可得：

$$c_{ij} = Te_{i+2}\left[p_{i,i+j}\right] >> (24-8i)\,\&0\text{xff} \oplus k_{ij} \tag{4-12}$$

故以上为最后一轮中，利用 Te 表得到最后密文的公式，以下为最后一轮利用 Te 表的算法伪代码：

算法 4.1 最后一轮加密算法

输入: 第九轮输入 P_i', 密钥 K_i,$Te_i\,(0 \le i \le 3)$

输出: 加密密文 $C_i\,(0 \le i \le 3)$

```
function LastRoundEncryption(P, K, Te)
    for i = 0 → 3 do
        C_i = Te_2 [(P'_i >> 24)&0xff] &0xff000000⊕
              Te_3 [(P'_{i+1} >> 16)&0xff] &0x00ff0000⊕
              Te_0 [(P'_{i+2} >> 8)&0xff] &0x0000ff00⊕
              Te_1 [(P'_{i+3}      )&0xff] &0x000000ff ⊕ K_i
    end for
    return C_0 ⊕ C_1 ⊕ C_2 ⊕ C_3
end function
```

4.2　最后一轮密钥的攻击

设定缓存攻击的目标是最后一轮的轮密钥，因为在 AES128 中，密钥操作是可逆的，因此可以用任何一个回合的密钥推导出原始的 16 字节加密密钥。由上式可以得到简化后的最后一轮的加密过程:

$$C_j = Te\left[P_j\right] \oplus K_j \tag{4-13}$$

最后一轮的加密密钥可以通过 Te 表项和最后的密文字节进行异或操作得到。在设定的加密模型中，加密过程是在安全世界中执行的，普通世界中的进程向安全世界提供明文，并从安全世界接受加密后的密文作为加密结果。因此，密文对于攻击者来说是已知的。等式中的另一个变量是 T 表项，它可以利用 prime+probe 攻击方法中获得的缓存访问模式中猜测出来。缓存访问模式可以理解为，安全世界在执行加密任务时具体使用了哪项缓存，因为 Te 表项是在缓存中保存的，我们

可以知道 Te 表在缓存中的具体布局，当得到具体访问的缓存项时，我们就可以知道对应的 Te 表内容。图4.2为计划的 prime+probe 攻击示意图。

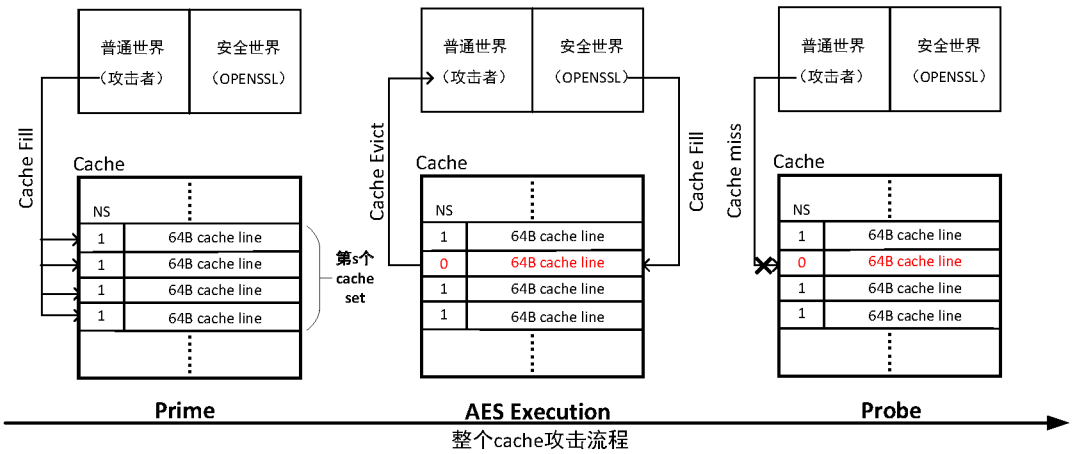

图 4.2　对 Te 表的 prime+probe 缓存攻击

Cortex-A53 的每个缓存行有 64 字节，Te 表的每一项为一个 int 类型数据，即 4 字节，因此每条高速缓存行拥有 16 个 Te 表项。一个 Te 表有 256 项，所以一个 Te 表所占用的大小为 1KB，所占用的地址空间对应着 16 个缓存组，四个 Te 表的地址空间刚好占用了 64 个 L1D 缓存组。也就是说，给定一个缓存组的组号，可以唯一的对应到某个 Te 表的连续 16 个字节。在执行缓存攻击，当受害者执行 AES 的过程中，攻击者之前占据的某个缓存行会被驱逐，攻击者再进行检测时就会发现，该缓存行中 16 个可能的 item 中有一个被用于加密过程，因此，一次 prime+probe 缓存攻击将会产生 16 个可能的候选密钥字节。

理想情况下，攻击者需要中断加密过程，只使用上一轮改变的缓存配置来推断一轮中使用的密钥。但是，由于安全世界收到 TrustZone 的保护，在其中执行的 AES 加密过程是无法中断的，也就是说，当一次加密中的 10 轮加密全部结束后，我们才能获取一些被驱逐的表项，并且无法知道这个表项是在第几轮加密中被使用到的。这种不确定性需要用大量的样本量进行观测来弥补。

下面对大样本量可以弥补密钥不确定性进行可行性分析。假设 Te_i 在十轮中一共使用了 m 次，一个缓存组可以存放连续的 16 个表项，假设 AES 是在 CBC 模式下使用的，那么算法的输入可以被看作随机数，则加密过程中没有访问 Te_i 的 16 个连续表项的概率是 $\left(1-\frac{16}{256}\right)^m$，因为一轮加密中需要用到 4 次 Te 表，故 10 轮加密则需要用到 40 次表项，平均到每个表即可以看作使用了 10 次，故代入 m=40，

可以得到概率值约为 8%。也就是说，任意一个缓存组有 92% 的概率在一次加密中被使用，有 8% 不会被用到，一次探测 64 个缓存组时，就有 $64 * 8\% = 5.12$ 的缓存组会检测不到 cache miss，这些缓存组的 Te 表项就可以被完全排除。因此，通过探测更多的样本，可以排除更多的不可能密钥，经过多轮迭代，最终有能力推断出密钥的值。

4.3　Neve&Seifert 排除法

Neve&Seifert 排除法[25] 是用上面得到的四个 Te 表的访问信息来得到加密密钥的算法。假设用 Prime+Probe 方法得到了一次加密中对四个 Te 表对应的缓存组的访问信息 $P_i\,(0 \leq i < 64)$，P_i 为 false 表示第 i 和 $i+64$ 个缓存组均没有被使用，由于噪音的原因，P_i 为 true 的情况不予考虑。设 $c_i\,(0 \leq i < 16)$ 为这一次加密的 16 字节密文。

规定 $Te'_{i+2}[\cdot] = (Te_{i+2}[\cdot] >> (24-8i))$ &$0x$ff 为 $GF(2^8) \mapsto GF(2^8)$ 的映射，由式 (4-12)，可以得到 $rk_i = c_i \oplus Te'_{i+2}[\cdot]$。以要恢复第 0 个字节的密钥为例，第 0 字节的密钥对应 Te_2 表的查表操作，若得知在加密过程中未访问到 Te_2 表以 t 开始的 16 个表项，则：

$$\forall a \in [t, t+16), rk_0 \neq c_0 \oplus Te'_2[a] \tag{4-14}$$

根据上式可以排除 rk_0 最多 16 个字节的取值，每一次攻击都累积记录这些非密钥的取值，当 16 个字节均只排除到只剩一个密钥时，证明所有非密钥已被排除，剩下的那个元素就是对应字节的密钥。

实现中，Te_0 的第 0 项到 Te_3 的最后一项连续占用了一个内存页的全部地址空间，由于每个连续的 16 个 T 表项存在被放进 0−63 缓存组和 64−127 缓存组两个可能性，并且这是普通世界无法通过 pagemap 获取的信息，所以需要开辟一个数组 probe[128] 来探测一共 128 个缓存组的驱逐情况，即若 probe[s] 和 probe[s + 64] 均未被访问，可以唯一地确定一个表的连续 16 项为非密钥，具体是 $Te_{[s/16]}$ 的第 $(s \bmod 16) * 16$ 至 $(s \bmod 16) * 16 + 15$ 项。

可以推出最后一轮中第 i 个密文字节的计算对应着 Te_{i+2} 的查表操作（下标中加法为 GF(4) 上的运算），设最后一轮密钥的 16 个字节的可能取值的集合为

$K_i = \{0, 1, ...255\}$，并且每次排除法之后 K_i 的结果可以继承。也就是说，每轮执行加密的时候，已经确定不是密钥的结果可以完全排除，不会出现在下一轮加密探测的可能密钥集中。在实现中，可以记录每一个 K_i 集合中元素的个数，若个数为 1，则说明排除密钥后只剩下一个可能的密钥了，也就是对应的密钥字节已经被恢复了，那么再接下来的轮数中，对这个密钥字节就可以直接跳过排除法步骤，这么做的目的是在攻击后期显著提升算法效率。

下面是 Neve&Seifert 排除法的简要步骤：

算法 4.2 Neve&Seifert 排除法

输入: probe 结果 $P_s(0 \leq s < 128)$, 密钥候选 $K_i\,(0 \leq i < 16)$, 加密密文 $c_i\,(0 \leq i < 16)$

输出: 当 $|K_i| = 1\,(0 \leq i < 16$ 时的 K_i

function ELIMINATION(P, K, C)
  for $s = 0 \rightarrow 127$ **do** ▷ 对 128 个缓存组分别进行遍历
    if $P_s == false$ **then** ▷ 未检测到 cache miss
      $i \leftarrow \lfloor s/16 \rfloor + 2 \bmod 4$
      while $i < 16$ **do** ▷ 对每一位的密钥候选项进行排除
        for $j = 0 \rightarrow 15$ **do** ▷ 每个 cache set 有 16 个 Te 表项
          $K_i \leftarrow K_i \backslash \left\{c_i \oplus Te'_{i+2}\left[(s \bmod 16) \times 16 + j\right]\right\}$
        end for
        $i \leftarrow i + 4$
      end while
    end if
  end for
  $flag \leftarrow true$
  for $i = 0 \rightarrow 15$ **do**
    if $|K_i \neq 1|$ **then** ▷ 每个 K_i 只剩一个值时即顺利攻击出密钥
      $flag \leftarrow false$
    end if
  end for
  return $flag, K$
end function

第五章　攻击方案

5.1　分配内存和缓存驱逐

为了获得关于受害者进程对于缓存的具体访问情况，攻击者必须准确地将特定的内存填充到特定的缓存区域，这样它就会引起与受害者进程的缓存争夺。受害者的 AES 加密程序中的 T 表用到了 128 中的 64 个缓存组，需要首先填满 128 个缓存组，对于一个缓存组中的 4 路，我们需要准备 4 个互不相同且有效的物理地址，且都对应着这个缓存组，并且从物理地址起始的连续 64 个字节都可读，就可以填满这个缓存组的 4 个缓存行。可以利用 pvalloc 开辟出 8 个 4K 页面，同时利用/proc/self/pagemap 控制其中的 4 个页面映射到前 64 个缓存组，另外四个页面映射到后 64 个缓存组。设基地址为 $base_i$，低 12 比特都为 0，且从 $base_i$ 开始的连续 4096 个字节都是可读的。设对应缓存组号为 s 的对应驱逐地址为 $evict_{s,i}\,(0 \leq i < 8, 0 \leq s < 64)$，则可以根据如下式子设计 prime 算法：

$$evict_{s,i} = base_i + (s << 6) \tag{5-1}$$

64 个缓存组刚好对应一整个 4KB 的页面的地址空间，故所开辟的 8 个页刚好可以直接填充 128 个缓存组。构造了内存驱逐集后，需要把所构造的数据放入缓存中。由于 ARM 处理器的缓存替代算法为随机替换，所以无法直接提出一个完全接近的替换算法，本文中，若要填充缓存组 s，则把 $evict_{s,i}$ 从 $i = 0 \rightarrow 7$ 正向遍历若干次，实验中设置为 10 次，重复的次数若过多，则可能会让攻击效率下降，若过少，可能无法导致缓存组的数据被完全填充导致遗漏。遍历后，需要检查数据是否都被填充至 L1 缓存中，故在每次 prime 后都设置一次 prime_check，对所有数据检查访问的时间是否超过一定阈值，若没超过，则可以认为所有数据都已经被加载入 L1 cache 中，prime 步骤就可以顺利完成。一旦有数据超过阈值，则认为在替换过程将之踢出缓存了，故需要重新进行 prime 过程，直到所有数据访问时间均未超过阈值。检查访问时间需要用到周期计数寄存器，将在下一节介绍。

5.2 周期计数器探测缓存

在执行了受害者的 AES 加密操作后，攻击者需要在普通世界中观测缓存的具体配置改变情况。由于安全世界的执行，需要探测此时缓存状态的变化。更具体地说，攻击者需要确定在 prime 步骤中填充的缓存行是否还在缓存中。我们从最高级缓存 L1 缓存的争夺中提取侧信道信息。因此，需要一个高精度的定时器来区分 L1 高速缓存或低级别的内存加载到寄存器中的时间上的不同。在 x86 系统中，可以使用 rdtsc 指令来提供纳秒级别的时间计时，在 Skylake 攻击平台上，甚至存在 MEM_LOAD_RETIRED.L1D_MISS 的事件可以直接记录 L1d 缓存发生的 cache miss 次数。但是，在 ARM 处理器中，没有这样的计时器，但 ARMv7 架构提供了一个性能监控单元 PMU，它有一个周期计数寄存器（PMCCNTR）。我们利用该周期计数寄存器来测量将内存或缓存载入寄存器所需的时间，以区分缓存层次结构中不同级别的缓存命中。

首先，为了在 userspace 能够正常使用 PMCCNTR，需要构造一个内核模块，并使用 sudo insmod enable.ko 命令激活。然后，我们以所示代码进行一个如何使用 PMCCNTR 的举例。指令流水线是许多现代处理器中用来实现指令级并行的技术，它使处理器的吞吐量更高，但它可能影响内存加载指令时间的测量，所以在周期计数寄存器被读取之前，使用 isb 指令（指令屏障）来完成流水线上的所有指令。此外，dsb（数据同步屏障）指令用来在定时器测量之前完成前面所有的操作。然后用 ldr 指令将存储在 r3 的内存地址加载到 r0。dmb(数据内存屏障指令), 用来等待前面访问内存的指令完成后再执行后面的指令。然后再读取周期计数器。两个定时器读数之间的差值被存储到 r0 中。按照类似的操作，可以完成所有指定的内存数据的探测。

```
1    dsb
2    isb
3    mrc p15, 0, r1, c9, c13, 0
4    ldr r0, [r3]
5    dmb
6    mrc p15, 0, r2, c9, c13, 0
7    sub r0, r2, r1
```

一旦时间被记录下来，就可以与提前测好的已知阈值进行比较，从而对内存访问的级别进行分类。不同级别的内存访问（L1、L2、DRAM）可以通过 Cortex-A53 的性能单元中的周期计数寄存器区分出来。经过实验，L1 缓存上的缓存命中平均有 90 个 CPU 周期，而从 L2 缓存加载使用 127 个 CPU 周期，从内存加载数据则需要 311 个 CPU 周期。因此，周期计数寄存器使攻击者不仅能可靠地分辨出缓存缺失和缓存命中之间的区别，还能分辨出缓存命中的缓存级别。利用其中的时间差，在 probe 阶段，我们可以判断每加载一个 64B 的数据时，所耗费的 CPU 周期，如果没有超过 90 这个 CPU 周期阈值，我们就认为此时的数据仍是从 L1D cache 中加载的，也就是说受害者进程在执行 AES 加密时，并没有用到这个地址对应的缓存组中的数据。因为我们提前可以知道这个缓存组中的 16 个 T 表项的具体内容，结合获得的密文，对于每个字节，我们就可以直接排除对应的 16 个可能的密钥值。

5.3　普通世界与安全世界交互设计

前文中提到，普通世界提供 AES 加密的 16 字节明文，唤醒安全世界的加密程序执行，并能够获得加密之后的 16 字节密文，本节将阐述实验中如何设计两个世界的程序交互过程。

普通世界和安全世界的程序可以看作 CA(Client Application, 运行在普通世界) 和 TA(Trust Application, 运行在安全世界) 两端的交互或者通话过程。典型的 CA 有支付应用、指纹对比等，利用特定接口调用可信执行环境中的服务，其中的具体过程对于 CA 端是透明且无法得到的。TA 是可信执行环境中完成特定功能的应用，这些应用通常具有较高安全性，执行特定服务后将计算结果返回给调用自己的 CA。

CA 在用户层面调用 CA 接口以后就会产生系统调用 (system call) 操作，系统调用会使 Linux 进入内核状态 (kernel space)，此时操作系统将处于内核区域，然后通过传入的参数，当 SMC 中断处理函数实现了将 cortex 的状态转换到安全世界和对相关参数的拷贝之后，安全世界中的操作系统将进行下一步的处理。操作系统首先会得到传过来的数据信息以及想要的 TA 的 UUID，并加载对应的 TA image。然后，安全世界中的操作系统会转换到 TEE 用户态，并将 CA 传递过来的其他参数传给 TA。当 TA 获取到参数之后，首先解析出参数中的 commond ID 值，根据具体的 command ID 值来做对应的操作，例如指纹对比、加解密操作。等操作执行

完成后，会根据之前的过程，依次回退，最后的结果将会传递给 CA 端，CA 端将获取可信服务的结果。整体流程可以参考图5.1。

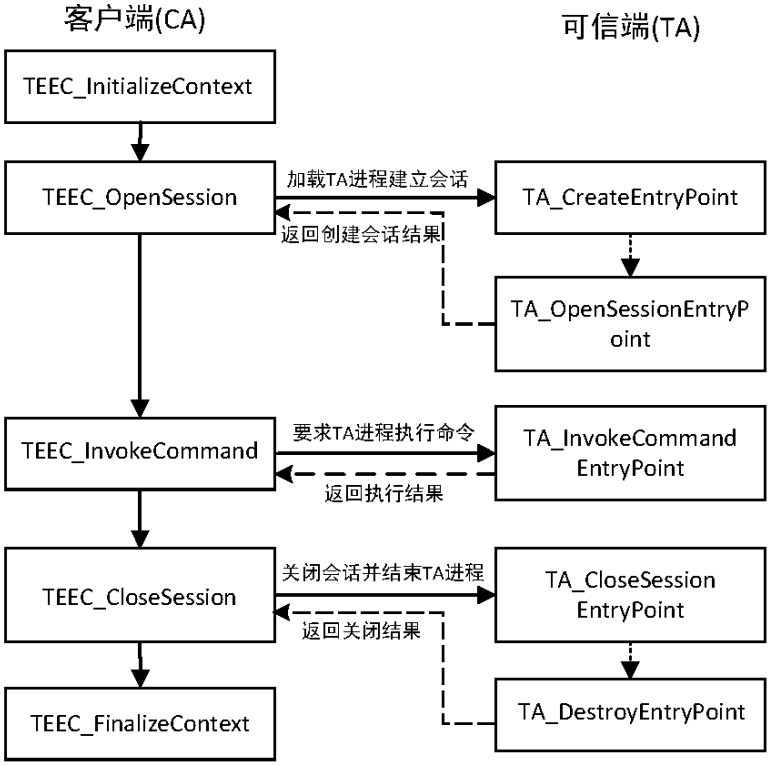

图 5.1　TA 与 CA 交互设计

CA 就是在 REE 中用来控制 TA 的应用程序，典型的 CA 程序流程遵循 GP 客户端 API。主要包含五个与 TA 中对应的 api 来控制 CA-TA 连接与指令操作。一次完整的 CA 调用过程需要分别实现以下五个接口操作：

- TEEC_InitializeContext: 通过初始化 TEE 上下文变量（context），以建立与 TEE 之间的通话通道
- TEEC_OpenSession: 建立 CA 与 TA 之间的会话窗口
- TEEC_InvokeCommand: 向 TA 发送 command 请求来执行具体的操作
- TEEC_CloseSession: 关闭 CA 与 TA 之间的对话窗口
- TEEC_FinalizeContext: 清空会话通道，清除 context 变量

结合本文实验的加密操作，具体来说，完成一次完整的 CA 请求时，程序需要执行的操作分别是：首先调用 TEEC_InitializeContext 函数，获取 ctx 变量并存放到

TEE_Context 类型的变量中。然后调用 TEEC_OpenSession 函数，创建一个 CA 与特定 TA 之间进行通信的通道。然后需要初始化 TEEC_Operation 类型的变量，由于 CA 端需要提供明文，故该变量第一个参数类型需设置为 TEEC_MEMREF_TEMP_INPUT，并且在变量中存放明文内容和字节数。使用规定的 command ID 作为参数，调用 TEEC_InvokeCommand 函数来真正发起对应的加密请求。之后的具体加密实现将由 OP-TEE 和 TA 进行处理，并将得到的密文返回给 CA 端。调用成功之后需要注销会话变量 sess，并且释放掉 ctx，这两个操作一次通过调用 TEEC_CloseSession 函数和 TEEC_FinalizeContext 函数来实现。TA 端的实现同样使用五个与上述对应的 API 来控制 CA-TA 连结与指令操作：

- TA_CreateEntryPoint：建立进入点，让 TA 可以被呼叫
- TA_OpenSessionEntryPoint：建立 CA 呼叫 TA 的通道 session
- TA_InvokeCommandEntryPoint：接收 CA 传送的指令，执行对应函数
- TA_CloseSessionEntryPoint：关闭 CA-TA 的通道
- TA_DestroyEntryPoint：移除进入点，结束 TA 的功能

在本文设置的实验中，完成一次加密操作的 TA 端，程序需要定义以下的内容：TA_CreateEntryPoint 和 TA_OpenSessionEntryPoint 分别对应了 CA 端的前两个 api，完成这两步后，TA 端就已经和 CA 端进行了加密操作的连接。TA_InvokeCommand EntryPoint() 函数则为 CA 端发起加密请求时对应的操作接口，因为本文中的 TA 端设置了两个加密服务，分别是初始化密钥以及加密操作，TA 根据传入的值，决定跳转到 TEE_Result aes_init() 或 TEE_Result aes_encrypt() 函数进行处理。初始化密钥函数中随机生成一个 16 字节密钥，利用密钥扩展操作初始化加密密钥。加密函数接收传过来的明文以及对应字节数，在函数内部对明文进行 openssl 的加密调用，并且生成 16 字节密文，同样放入参数中传回给 CA 端。TA_CloseSession EntryPoint 和 TA_DestroyEntryPoint 同样对应了 CA 端的后两个 api，当被调用后，整个连接将被终端，剩下的操作将交给普通世界中的程序进行缓存侧信道攻击。

5.4 攻击算法流程

为了模拟真实的攻击环境，AES 加密的密钥由 tee 内 TA 产生，外部攻击者无法直接获取，外部攻击者 CA 可以通过 TEEC_Operation 类型的参数来传递加密的明文和获取加密的密文。为了进一步减少噪音，本攻击设置了 Linux 进程调度优先级，将攻击者线程的优先级调至最高，以保证攻击者在整个攻击阶段的程序不会被其他线程干扰，此过程需要 root 权限。

具体的普通世界与安全世界的交互设计见图5.2。在攻击前，攻击者指定一个特定的 16 字节明文，获取它的 TrustZone 加密结果，存储起来。然后攻击者正式开始攻击过程，在每一次的 prime+probe 过程中，都随机生成一个明文，发送给受害者进行加密，并通过密文及缓存信息结合排除法进行密钥的筛选。在攻击者通过排除法把所有的密钥取值候选值都只剩余 1 个时，攻击者通过最后一轮的密钥，计算出最初的加密密钥，用这个密钥去解密前面存储的密文，若与指定的明文相同，则说明攻击密钥恢复成功。

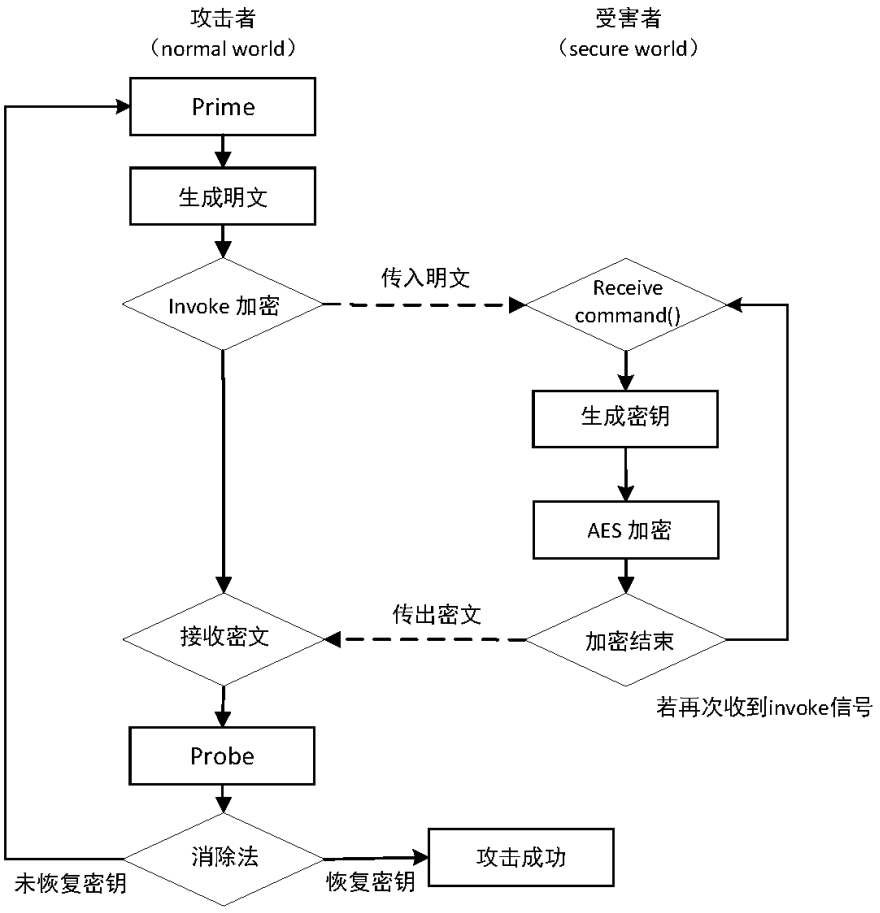

图 5.2 攻击算法流程

第六章　实验与分析

6.1　攻击结果与性能分析

本小节介绍实验中的攻击结果以及相关性能分析。实现攻击的硬件平台为Cortex-A53 处理器 armv7l 架构，有四个物理核。攻击环境是 Linux Raspbian GNU/Linux 10 (buster)，内核版本为 4.14.98-v7，所有 ta 端编译得到的.ta 文件均放在/lib/optee_armtz 文件夹中。

在 teecache 目录下输入 sudo LD_LIBRARY=$(HOME)/teeaes_aes ./spy, 并且循环 1000 次攻击，即可得到以下攻击结果。可以看到受害者提前生成一个密钥，这个密钥攻击者是不可知的，只是为了可视化展示输入在屏幕上。攻击者发送明文并且收到加密密文。利用上一章介绍的算法流程，对缓存进行 prime+probe 攻击，最终恢复了最后一轮密钥，并由 aes128_key_schedule_inv_round 函数得到了最初的密钥，利用这个密钥对收到的密文进行解密，得到的明文与最初发送的明文一致，故攻击成功。

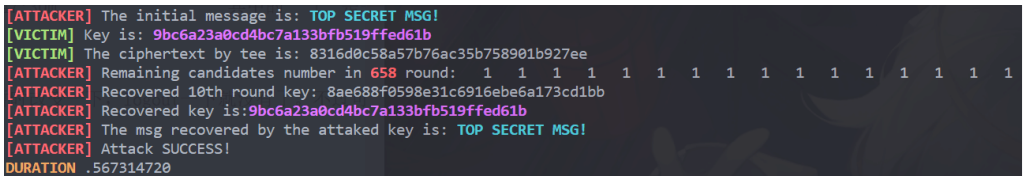

图 6.1　攻击结果

除了观察攻击算法的整体效率，实验中还另外对 probe 过程进行了性能分析。一轮探测（probe round）指一次 prime+probe 过程中获取对所有的 Te 表所占的 64 个缓存组的使用信息，可以理解为，一次成功的攻击由多次一轮探测组成，经过每次一轮探测都会排除一些不可能的密钥项，而每个一轮探测中，有可能由多次 prime+probe 过程组成，具体次数将取决于 prime 和 probe 缓存时设置的缓存组数。一轮探测可能会触发多次加密，例如，若每次 prime 和 probe 过程探测的缓存组为 2^l 个，则一轮探测需要触发 2^{6-l} 次加密，这样才能保证一轮探测过程能够获取所有缓存组的信息。由于外界噪音的原因，实验探究了每次 probe 过程探测的缓存

组数量与攻击效率的关系。分别设置一次探测的缓存组数为 $2^0, 2^1, 2^2, 2^3, 2^4, 2^5, 2^6$，每种组数设置 20 次攻击并记录求平均值，分别观察攻击时间和探测轮数的变化情况。

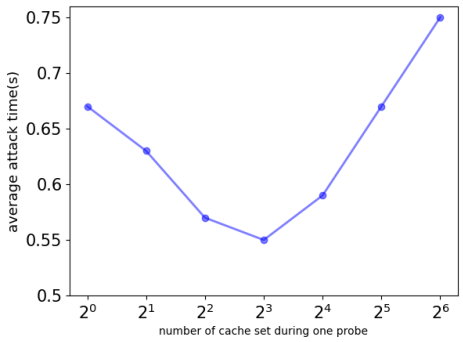

图 6.2 攻击时间随单次 probe 组数变化

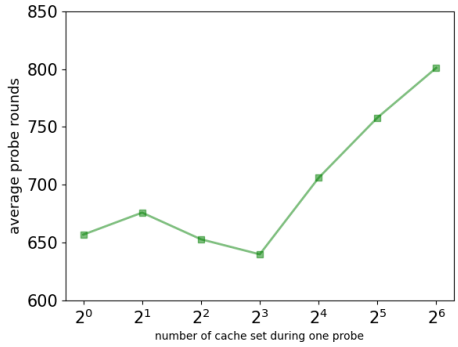

图 6.3 总探测轮数随单次 probe 组数变化

从图6.2可以得到，不同探测缓存组数拥有着不同的平均攻击时长，从每次探测 8 个缓存组开始，每次探测的缓存组数量越多，就越容易被噪声影响，平均每一轮探测得到的信息也就越少，平均耗时也相对较多。从图中可以得出，攻击耗费时间最少的是每次探测 8 个缓存组，此时为了探测所有的 64 个缓存组，一轮探测中就需要 8 次加密，结合图6.3，单次 probe 8 个缓存组时平均需要 640 次一轮探测，故平均恢复出密钥需要 8*196=5120 次加密。

由于需要得到一次加密过程对 64 个缓存组的全部使用信息，故一轮探测需要触发的加密次数与单次探测的缓存组数成反比，所以，单次 probe 的缓存组越多，一轮探测需要触发的加密次数就越少，为了加强攻击的实用性，我们在这里以恢复密钥所需要的最少加密次数作为主要性能指标，因为在真实攻击场景中，TEE 中的程序很有可能监测到一定次数的运行异常后，就暂时自锁程序运行，故我们要在尽可能少地触发可信程序运行的基础上，攻击出密钥，而攻击时间反而没有这么重要。而在每次探测的缓存组为 64 个时，虽然耗时较最少时间多了 200ms 左右，但是只需要加密约 800 次就可以完整恢复加密密钥。故将每次探测缓存组数量设置为 64，此时，单次探测会探测所有的 64 个缓存组，每个一轮探测只触发一次加密。在确定了单次 probe 的缓存组数为 64 个之后，共实施了 1000 次攻击，最终得到正确恢复出密钥的比例为 99.92%。

图6.4展示了一次攻击过程中密钥候选项数量的变化。基于 1000 次的攻击数据，可以得出当前的一轮探测次数与 16 个剩余密钥字节取值数量的最大值、最小

值和平均值。假设任意一次攻击的第 t 次探测后，最后一轮密钥的 16 个字节的未排除的取值个数分别为 $r_i(t)\,(0 \leq i < 16)$，根据大数定律，从数据计算出来候选取值个数的最大值对应于 $E[\max(r_i(t))]$，最小值对应于 $E[\min(r_i(t))]$，平均值对应于 $E[(r_i(t))]$，其中 E 是期望符号。

从图6.4可以得出在 200 次加密左右，最小值接近于 1，说明在平均不到 200 次加密内就可以恢复出最后一轮密钥的少数几个字节。在 800 次加密左右，最大值、平均值、最小值都接近于 1，说明绝大多数的最后一轮密钥字节在 800 次加密内已经被恢复，只剩下少数几个还未恢复的。而剩余的少数几个还未恢复的密钥剩余的候选数量也非常少。然而，我们使用的 Neve&Seifert 排除法中，最后剩余的几个密钥取值往往还再需要很多轮才能够排除掉，这也就是图6.4有较长的拖尾的原因。

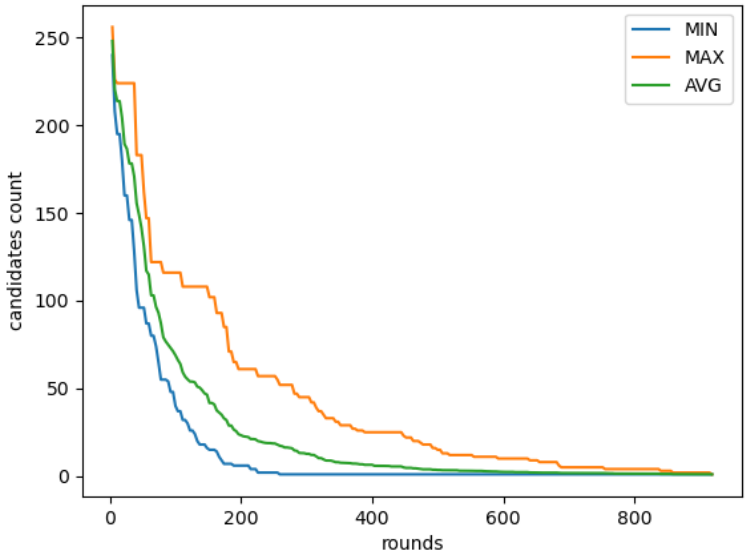

图 6.4　密钥候选变化情况

图6.5, 图6.6分别是基于 1000 次实验中，一次攻击中探测轮数的实验次数直方图和攻击时长的各段实验数量直方图，在设置的实验平台上，1000 次攻击的平均耗时为 0.75 秒，恢复出加密密钥平均需要触发 801.07 次 AES 加密。可以看到 1000 次模拟攻击中，大多数需要的加密轮次集中在 650 到 900 之间。一小部分数据是 1000 轮左右攻击出密钥的，可能是缓存噪声等各种突发因素，具有一定的偶然性。从攻击时间来看，大多数攻击可以在 1.0s 内得到密钥，70% 的攻击甚至可以在 0.8s 内成功获得密钥，整体算法效率较高。

除了上述的排除法，还可以利用投票法计算猜测熵（Guessing Entropy，GE），

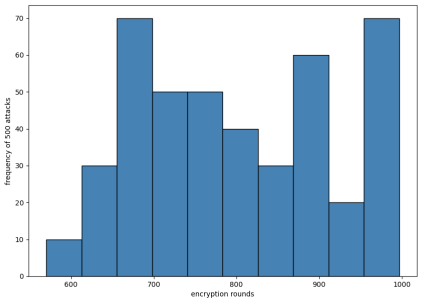

图 6.5　探测轮数的实验次数

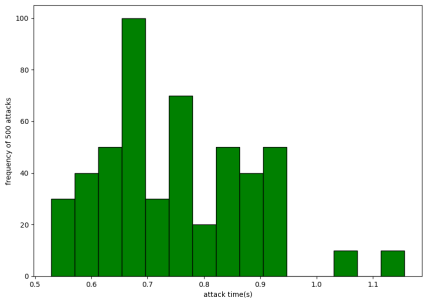

图 6.6　攻击时长的试验次数

以此估计该侧信道攻击的可靠性。猜测熵在侧信道攻击中，是对攻击者猜测系统效果的常用评估手段。在本文的实验中，在 probe 步骤之后，得到了没有发生 cache miss 的缓存组对应的 Te 表项，对这些表项的 vote 值进行投票累加，同时对整个 prime+probe 过程规定固定次数。当遍历完规定次数后，观察每个字节的 256 个 vote 值，其中投票次数最少的为该字节最有可能的密钥。因为每种攻击算法最后不一定能攻击出正确的密钥，故可以将正确的密钥对应的投票数在所有投票数中进行排序，若最后得到投票数最少，记猜测熵为 1，表明攻击效果较好，能成功攻击出密钥，猜测熵越大，可以看作攻击效果越差，即 GE $(byte_i) = rk_i$。

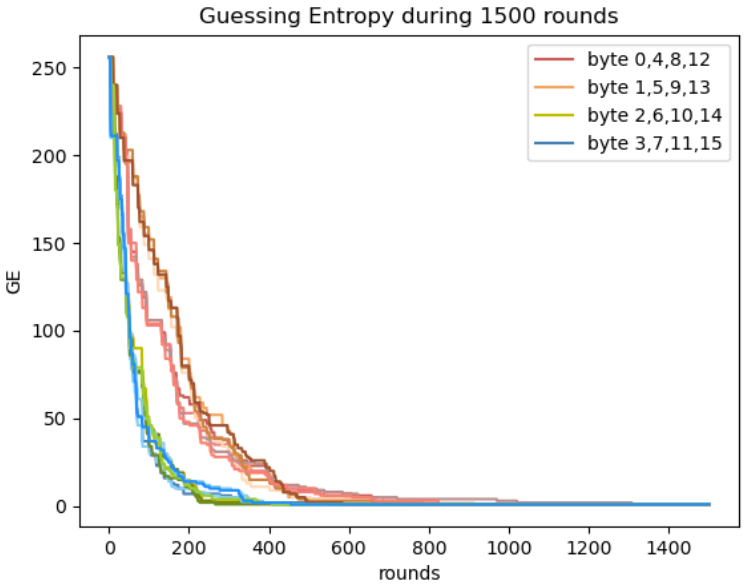

图 6.7　猜测熵随轮数变化情况

图6.7为任意 50 次攻击的平均猜测熵下降图，图中将同色系作为不同位置字节的标识。可以看到，整体的下降趋势和图6.4类似，但反应的信息更详细。首先，每

个字节在规定的 1500 次遍历后，GE 均下降到 1，证实了攻击的有效性。每个色系的字节在加密过程中的位移或占用的 Te 表都是相等的，所以攻击速度也相互接近。蓝色和绿色系的字节在 400 轮左右猜测熵已经降为 1，红色系对应的字节虽然一开始下降较快，但是直到 1000 轮左右才完全猜测出密钥，证明 $P_i\,(0 \leq i \leq 3)$ 中的第一个字节是攻击的关键，决定了攻击的整体速度，具体原因有可能是调用 tee 过程中的内存所对应的缓存组正好与之相同，导致攻击者在前期一直无法探测出。

6.2　cache 攻击应对方法

自从近二十年前第一个基于缓存的时间侧信道攻击被提出，学术界已经做了很多研究来推进对缓存信息侧信道的攻击和防御。与以前的工作类似，本文研究的最终目标是进一步了解 TrustZone 安全世界中的侧信道信息泄露，并优化未来的设计。对侧信道攻击的防御通常遵循两个方向。第一个方向试图在加密操作本身的实现上下功夫，而第二个方向则侧重于加固加密操作所运行的系统，以消除在时间上的侧信道信息。这部分主要介绍一些可能的解决缓存侧信道攻击的方法。

在改进加密算法的方法中，最直接的方法是关注加密算法本身。OpenSSL 采用了一种缓解措施来预装 AES 的 T 表，这样，假设加密算法的执行不能被打断，对该表的访问就不能再被攻击者发现，也就是说，攻击者连 T 表的逻辑地址等基本信息都无法获得，更不能得到对应的缓存组信息了。还有一些防御性的措施，例如将软件控制流随机化[26]，这样执行路径和缓存组之间就没有固定的关系，一旦没有这种映射信息，攻击者就不可能从高速缓存配置中推断出重要数据，也就不能汇总统计数据来推断密钥。

在加固加密操作所运行的系统这个方向上，我们需要先着重了解破解高速缓存时间侧信道攻击的两个要求。首先，攻击者必须能够用内存填满缓存，以造成与受害者进程的资源争夺，因此可以试着消除这种资源争夺。由于内存分配是由内核指定的，一个操作系统级攻击者，在完全控制所有普通世界内存的情况下，可以成功攻击秘密值。那么，为了组织这种缓存争夺，我们可以规定程序对缓存的使用，最简单的就是关闭缓存，完全关闭缓存肯定会影响程序的运行效率，但如果操作系统可以提供一些特殊的操作，允许程序自行决定是否可以启用缓存，受害者程序可以选择在执行涉及到秘密数据的时候关闭缓存，这样对整体的运行效率不会产生太大的影响。类似原理的操作还可以是，不关闭缓存的使用，但是给安

全世界内的程序划分特定的缓存隔离区域，也就是跟内存一样的分隔机制，这些缓存内容是不能被其他程序或操作系统读写的，并且在该程序执行完成后会自动销毁并回收。

创建高速缓存侧信道的第二个要求是能够检测缓存状态的变化。因此，攻击者需要一个高精度的定时器来区分来自内存系统不同层次的内存访问。如果我们能限制访问内核性能事件接口的权限，有可能可以防止非内核权限的应用对安全世界侧信道信息的访问。

第七章　总结与展望

7.1　方案总结

本文先介绍了 ARM TrustZone 可信执行环境的运行原理，然后分析了 4 种现代常用的缓存攻击方法。由于 TrustZone 的内存分离机制，只有 prime+probe 攻击是无法抵抗的。我们实践了基于时间的缓存侧信道攻击，能够从 TrustZone 的安全世界中提取密钥，即开发了一个攻击模式，其中攻击者拥有操作系统最高权限，通过触发在安全世界里执行的 AES 加密，最终证实可以在 0.75 秒左右进行大约 801.07 轮观察的 AES 加密，并且成功从安全世界中提取完整的 16 位 AES 加密密钥。由于我们的攻击依赖于启用 TrustZone 的缓存架构的基本设计，因此理论上它对各种版本的 ARM 处理器均有效。最后，本文提出了可能的缓存攻击预防策略。

7.2　未来工作

本文使用 AES 作为从信任区的安全世界中提取重要或敏感信息（如秘密密钥）的能力的示范。然而，它的适用性绝不仅限于 AES。在 AES 中跟踪 T 表访问的方法也可以应用于其他基于表的加密实现[27,28]。同时，当安全世界内的程序使用内核地址空间布局随机化（KASLR）[29] 时，同样可以使用类似的缓存侧信道手段进行攻击。

从其他平台的应用来考虑，尽本文的原理并不局限于单核处理器，因为我们目前的实现使用最高级的 L1 数据高速缓存来减少噪音，但还需要一些进一步的研究来扩展这种方法在多核处理器上的机制。在许多现代多核处理器中，每个单独的处理器内核都有自己的专用 L1 高速缓存。为了继续利用 L1 上的缓存争夺作为侧信道信息的来源，攻击需要被安排和受害者处在同一个处理器核心上运行，这通常很难实现。

另一方面，最后一级缓存通常由所有内核共享，以前的一些工作主要是利用最后一级缓存作为侧信道信息的来源[17,27,28]。当把这种方法扩展到多核处理器（如

ARMCortex-A9）时，攻击需要使用最后一级共享缓存来应用 prime+probe 技术。然而，在一些 ARM 处理器上，如 CortexA-9，缓存控制器拥有 inclusive 的特点，以确保存在高层缓存中的任何缓存行也存在于低层缓存中，那么，对最后一级缓存的探究将可能受到其它核的高级缓存的操作影响，这种情况下，低级缓存中可能存在额外的噪音，将给攻击提升一定的难度。因此，对其它加密方法的应用、对多核处理器的应用以及对具有 inclusive 特性的最后一级缓存的应用，都可以成为将来 prime+probe 缓存侧信道攻击在 ARM 架构处理器上的研究对象。

致　谢

毕业设计到这里要画一个句号了，从去年 10 月出题，到今年三月份因为疫情临时换了个题目，再到现在完成了题目，整个过程现在回看其实还挺感慨的。去年 10 月份师兄给我出了第一个题目时，当时虽然对出的题目的知识点不太熟悉，可是学习的劲头还是比较强的，很快就将一些基础知识学会了，一直到今年二月去到上海进行校外毕设的实习的前两周，过程都比较顺利，虽然还是有很多不会的，但是真的非常感谢组里的师兄们的耐心教导，从一开始的配环境到采集曲线，一步步都很耐心的带着我，让我得到了很大的提升。本以为过程可以很顺利的进行，但是突入其来的疫情让我无法前往实验室进行线下实验，于是三月底只能临时换了个可以线上进行的题目。从尝试阅读这方面的论文，再到进行实践，留给我的时间非常紧张，压力其实也挺大的，一度担心自己能不能在五月份完成工作。

我要衷心感谢指导我的老师和师兄，尤其是姬宇航师兄，在我茫然的时候会帮助我想方法，在我无法理解论文时耐心地替我讲解。在论文中，从开题报告到中期报告，再到最后的论文，整个过程挺曲折的，但最终还是顺利地完成了任务。

同时，我也要感谢父母，在我做毕设的时候经常关心、鼓励我，在我因为疫情被困的时候给我很大的帮助和支持，从而让我有更大的动力完成论文。

本科就这样结束了，新的起点就要从零开始，我一定要继续努力。

参考文献

[1] Zhang N, Sun K, Lou W, et al. Case: Cache-assisted secure execution on arm processors[C]// 2016 IEEE Symposium on Security and Privacy (SP). IEEE, 2016: 72-90.

[2] Azab A M, Ning P, Shah J, et al. Hypervision across worlds: Real-time kernel protection from the arm trustzone secure world[C]//Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security. 2014: 90-102.

[3] Zhou Y, Wang X, Chen Y, et al. Armlock: Hardware-based fault isolation for arm[C]// Proceedings of the 2014 ACM SIGSAC conference on computer and communications security. 2014: 558-569.

[4] Jang J S, Kong S, Kim M, et al. Secret: Secure channel between rich execution environment and trusted execution environment.[C]//NDSS. 2015: 1-15.

[5] Kocher P, Jaffe J, Jun B. Differential power analysis[C]//Annual international cryptology conference. Springer, 1999: 388-397.

[6] Kocher P C. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems [C]//Annual International Cryptology Conference. Springer, 1996: 104-113.

[7] 王崇, 魏帅, 张帆, 等. 缓存侧信道防御研究综述[J]. 计算机研究与发展, 2021, 58(4):794.

[8] Genkin D, Pachmanov L, Pipman I, et al. Ecdsa key extraction from mobile devices via nonintrusive physical side channels[C]//Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. 2016: 1626-1638.

[9] Genkin D, Shamir A, Tromer E. Rsa key extraction via low-bandwidth acoustic cryptanalysis [C]//Annual cryptology conference. Springer, 2014: 444-461.

[10] Bernstein D J. Cache-timing attacks on aes[J]. 2005.

[11] Acıiçmez O, Koç Ç K. Trace-driven cache attacks on aes (short paper)[C]//International Conference on Information and Communications Security. Springer, 2006: 112-121.

[12] Percival C. Cache missing for fun and profit[M]. BSDCan Ottawa, 2005.

[13] Osvik D A, Shamir A, Tromer E. Cache attacks and countermeasures: the case of aes[C]// Cryptographers' track at the RSA conference. Springer, 2006: 1-20.

[14] Gullasch D, Bangerter E, Krenn S. Cache games–bringing access-based cache attacks on aes to practice[C]//2011 IEEE Symposium on Security and Privacy. IEEE, 2011: 490-505.

[15] Yarom Y, Falkner K. {FLUSH+ RELOAD}: A high resolution, low noise, l3 cache {Side-Channel} attack[C]//23rd USENIX security symposium (USENIX security 14). 2014: 719-732.

[16] 袁稚炜. 基于差分 F_ (LUSH)+ R_ (ELOAD) 的缓存侧信道攻击方法研究[D]. 南京航空航天大学, 2019.

[17] Ristenpart T, Tromer E, Shacham H, et al. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds[C]//Proceedings of the 16th ACM conference on Computer and communications security. 2009: 199-212.

[18] Zhang Y, Juels A, Reiter M K, et al. Cross-vm side channels and their use to extract private keys[C]//Proceedings of the 2012 ACM conference on Computer and communications security. 2012: 305-316.

[19] Weiß M, Heinz B, Stumpf F. A cache timing attack on aes in virtualization environments[C]// International Conference on Financial Cryptography and Data Security. Springer, 2012: 314-328.

[20] Lipp M, Gruss D, Spreitzer R, et al. {ARMageddon}: Cache attacks on mobile devices[C]//25th USENIX Security Symposium (USENIX Security 16). 2016: 549-564.

[21] Zhang X, Xiao Y, Zhang Y. Return-oriented flush-reload side channels on arm and their implications for android devices[C]//Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. 2016: 858-870.

[22] Zhang N, Sun K, Shands D, et al. Truspy: Cache side-channel information leakage from the secure world on arm devices[J]. Cryptology ePrint Archive, 2016.

[23] Guanciale R, Nemati H, Baumann C, et al. Cache storage channels: Alias-driven attacks and verified countermeasures[C]//2016 IEEE Symposium on Security and Privacy (SP). IEEE, 2016: 38-55.

[24] Holdings A. Arm security technology: Building a secure system using trustzone technology[J]. Retrieved on June, 2009, 10:2021.

[25] Neve M, Seifert J P. Advances on access-driven cache attacks on aes[C]//International Workshop on Selected Areas in Cryptography. Springer, 2006: 147-162.

[26] Crane S, Homescu A, Brunthaler S, et al. Thwarting cache side-channel attacks through dynamic software diversity.[C]//NDSS. 2015: 8-11.

[27] 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[C]//2015 IEEE Symposium on Security and Privacy. IEEE, 2015: 591-604.

[28] Liu F, Yarom Y, Ge Q, et al. Last-level cache side-channel attacks are practical[C]//2015 IEEE symposium on security and privacy. IEEE, 2015: 605-622.

[29] Hund R, Willems C, Holz T. Practical timing side channel attacks against kernel space aslr[C]// 2013 IEEE Symposium on Security and Privacy. IEEE, 2013: 191-205.