定稿修改稿:

HTrustZone:借助 Hypervisor 强化 TrustZone 对非安全世界的监控能力*

章张锴[1], 李舟军[1], 夏春和[1], 马金鑫[2], 崔津华[3]

[1](北京航空航天大学大学计算机学院,北京 100191)

[2](中国信息安全测评中心,北京 100085)

[3](新加坡管理大学,新加坡 178895)

通讯作者: 李舟军, E-mail: lizj@buaa.edu.cn

摘 要: ARM TrustZone 技术已经在 Android 手机平台上得到了广泛的应用,它把 Android 手机的硬件资源划分为两个世界,非安全世界(Non-Secure World)和安全世界(Secure World).用户所使用的 Android 操作系统运行在非安全世界,而基于 TrustZone 对非安全世界监控的系统(例如,KNOX,Hypervision)运行在安全世界.这些监控系统拥有高权限,可以动态地检查 Android 系统的内核完整性,也可以代替 Android 内核来管理非安全世界的内存.但是由于 TrustZone 和被监控的 Android 系统分处于不同的世界, world gap(世界鸿沟)的存在导致处于安全世界的监控系统不能完全地监控非安全世界的资源(例如,Cache).TrustZone 薄弱的拦截能力和内存访问控制能力也弱化了它对非安全世界的监控能力.首次提出一种可扩展框架系统 HTrustZone,能结合 Hypervisor 来协助 TrustZone 抵御利用 world gap 的攻击,增强其拦截能力和内存访问控制能力,从而为非安全世界的操作系统提供更高的安全性保障.并在 Raspberry Pi2 开发板上实现了 HTrustZone 的原型系统,实验结果表明: HTrustZone 的性能开销仅仅增加了 3%左右.

关键词: TrustZone;Hypervisor;监控系统;虚拟化

中图法分类号: TP311

中文引用格式: 章张锴,李舟军,夏春和,马金鑫,崔津华.HTrustZone:借助 Hypervisor 强化 TrustZone 对非安全世界的监控能力.软件学报.

英文引用格式: Zhang ZK, Li ZJ, Xia CH, Ma JX, Cui JH. HTrustZone: Utilizing Hypervisor to Extend TrustZone's Introspection Capabilities. Ruan Jian Xue Bao/Journal of Software, 2017 (in Chinese).

HTrustZone: Utilizing Hypervisor to Extend TrustZone's Introspection Capabilities

Zhang Zhang-Kai[1], Li Zhou-Jun[1], Xia Chun-He[1], Ma Jin-Xin[2], Cui Jin-Hua[3]

[1](School of Computer Science and Engineering, Beihang University, Beijing 100191, China)

[2](China Information Technology Security Evaluation Center, Beijing 100085, China)

[2](Singapore Management University, 178895, Singapore)

Abstract: The technology of ARM TrustZone has been widely used on the Android phone, which divides the hardware resources of Android phone into two worlds: Non-Secure World and Secure World. The Android operating system used by user is running on the Non-Secure World, while the Non-Secure World's introspection systems (e.g., KNOX, Hypervisor) that are based on TrustZone are running on the Secure World. These introspection systems have the high privilege. They can dynamically check Android kernel integrity

基金项目: 国家 863 高技术研究发展计划项目(2015AA016004); 国家自然科学基金(61370126, 61672081, 61502536, U1636208)

Foundation item: National High Technology Research and Development Program of China (2015AA016004); National Natural Science Foundation of China (61370126, 61672081, 61502536, U1636208)

and perform memory management of Non-Secure World instead of Android kernel. But TrustZone can not completely introspect the hardware resources (e.g., Cache) of Non-Secure World because of the world gap (introspection systems and Android system are in the different worlds). TrustZone's inferior interception capabilities and memory access control capabilities make its introspection capabilities weaker. To the best of our knowledge, we are the first to propose an extendable frame system HTrustZone that utilizes Hypervisor to extend TrustZone's introspection capabilities to defeat world gap attacks, strengthen interception capabilities and memory access control capabilities. HTrustZone can help TrustZone make great progress on system introspection and give more security guarantees to the operating system on Non-Secure World. We have implemented HTrustZone system on Raspberry Pi2 development board and the experiment results show that the overhead of HTrustZone is about 3%.

Key words: TrustZone; Hypervisor; Introspection System; Virtualization

随着智能手机的高速发展,Android 内核变得越来越复杂,可信计算基(TCB:Trust Computing Base)也越来越大.这使得 Android 内核会不可避免地存在一些已知的或未知的漏洞.攻击者可以利用这些漏洞来完全控制内核,从而掌控整个 Android 系统来发起拦截短息,盗取隐私信息(通讯录,照片)等攻击[1-3].这类攻击拥有内核权限且几乎不可能被用户察觉到,危害极大.由于这类攻击的存在,Android 内核这个绝大多数智能手机上的可信根不再可信.寻找权限更高的新的可信根来检测和抵御这类攻击的需求是非常迫切的,而 TrustZone 就是新可信根的一个很好的选择[4-11].通过 TrustZone 来监控 Android 内核的运行是抵御内核级攻击,提高 Android 系统安全性的一种方法.

然而 TrustZone 并不能主动地打断被监控操作系统的运行来进行实时的安全性检查,通常需要依靠非安全世界操作系统的内核模块协助或者修改内核代码的方法来实现这一打断操作,这些方法又会引入内核模块和被修改的代码的安全性问题.而处于非安全世界的 Hypervisor 可以利用硬件特性有效地拦截操作系统的系统调用,上下文切换和一些重要寄存器的访问等事件,在监控点打断操作系统的运行.更重要的一点是,在硬件虚拟化技术的帮助下,Hypervisor 可以很好地保护自身空间来抵御内核级的攻击.HTrustZone 利用 Hypervisor 的这些特点来协助 TrustZone 对非安全世界的操作系统进行实时的安全性检查.

8 硬件特性介绍

8.1 TrustZone

ARM TrustZone 技术作为安全拓展最早是在 ARMv6 的版本中被引入的[12],它把硬件的资源划分为两个世界,非安全世界和安全世界,其中 Android 系统工作在非安全世界,TrustZone 工作在安全世界.当 CPU 工作在安全世界模式的时候,它可以访问安全世界的资源和非安全世界的资源.但是当 CPU 工作在非安全世界模式的时候,安全世界的资源访问是被禁止的.例如,当 CPU 工作在安全世界模式的时候,通过操作 TZASC(TrustZone Address Space Contoller)寄存器和 TZMA(TrustZone Memory Adapter)寄存器可以把一块物理内存设置为"安全内存"(Secure Memory),非安全世界对这块内存的访问将出现不可预知的错误.而安全世界是允许访问非安全世界的物理内存的.通过这种方式,安全世界可以有效地保护特定的物理内存并监控非安全世界的物理内存的使用.正因为 TrustZone 拥有比工作在非安全世界的 Android 系统更高的权限,它可以用来检测和监控 Android 系统的安全性.

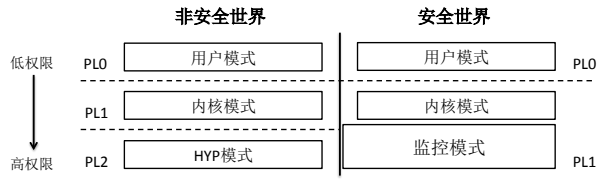

Fig.1 ARMv7 Architecture

图 1 ARMv7 体系结构

图 1 描述了 ARMv7[13]的硬件架构,非安全世界有三个模式:用户模式(USR mode),内核模式(SVC mode)和 HYP 模式(HYP mode),其中为了支持 CPU 的硬件虚拟化,ARMv7 新增了 HYP 这个新的 CPU 模式.Android 的应用程序运行在用户模式下,Android 内核运行在内核模式下,Hypervisor 运行在 HYP 模式下,拥有非安全世界最高的权限.安全世界也有三个模式,用户模式,内核模式和监控模式(Monitor mode).监控模式是连接非安全世界和安全世界的桥梁.在非安全世界的内核模式或者 HYP 模式下,执行 SMC(Secure Monitor Call)指令,可以主动地从非安全世界切换到安全世界的监控模式.在监控模式下执行 ERET 指令,CPU 通过检查 SCR(Secure Control Register)寄存器的 NS 位来决定返回非安全世界还是继续停留在安全世界.监控模式拥有最高的权限,它可以访问 CPU 上的所有寄存器,因此 TrustZone 可以配置 Hypervisor 相关的控制寄存器来初始化并激活 Hypervisor.

8.2 Hypervisor

Hypervisor 作为 ARM 的虚拟化拓展最早是在 ARMv7 的版本中被引入的[13].与 TrustZone 不同,Hypervisor 运行在非安全世界最高权限模式下(HYP 模式).在非安全世界,应用程序运行在权限等级为 0 的模式下,内核运行在权限等级为 1 的模式下,Hypervisor 运行在权限等级为 2 的模式,权限等级越高,权限越大.因此 Hypervisor 可以访问非安全世界包括寄存器,内存和 Cache 在内的所有硬件资源.在非安全世界的内核模式执行 HVC(Hypervisor Call)指令是进入 Hypervisor 的一种常见方式.

利用内存虚拟化技术,Hypervisor 可以激活第二层内存地址翻译(Stage-2 translation)来管理内存.原本虚拟地址(VA)到物理地址(PA)的内存地址翻译就转变为虚拟地址(VA)到中间地址(IPA),再从中间地址(IPA)到物理地址(PA)的内存地址翻译.其中第二层内存地址翻译(IPA 到 PA)的这个过程对于操作系统来说是透明的,在操作系统看来,IPA 就是它所看到的"物理地址".通过设置第二层地址翻译页表项(page table descriptor)访问监控位,Hypervisor 可以控制操作系统对内存物理页的访问属性.设置读写控制位,可以把物理页的访问权限定义成只读,只写,读写和不可读不可写四种类型.设置执行控制位,可以把物理页的执行权限定义成可执行或不可执行.操作系统并不知道这种细粒度物理页访问监控的存在,一旦操作系统违反了物理页的访问属性,CPU 将产生异常陷入 HYP 模式并由 Hypervisor 去处理这个异常.利用第二层内存地址翻译,Hypervisor 可以很好地保护自己的内存地址空间和监控操作系统对内存物理页的访问.不过,一些外设(例如,鼠标和键盘)可以通过 DMA(Direct Memory Access)的方式访问物理内存,而 DMA 是不需要经过第二层内存地址的翻译的,因此 Hypervisor 不能直接控制 DMA 对物理内存的访问.在支持 SMMU(System MMU,与 x86 上的 IOMMU 类似)的设备上,每一次 DMA 访问都需要 IOVA 到 IOPA 的翻译.检查并保护 SMMU 的页表,Hypervisor 可以限制 DMA 物理内存访问的范围,从而抵御 DMA 对敏感物理内存的攻击.

8.3 TrustZone与Hypervisor的优缺点分析

所属世界: Hypervisor 和被监控的操作系统都运行在非安全世界,这有利于 Hypervisor 监控操作系统的各种硬件资源;而 TrustZone 和被监控的操作系统运行在不同的世界,有可能受到利用 world gap 的攻击.

拦截能力: Hypervisor 拥有强大的拦截操作系统指令和事件的能力,而 TrustZone 的拦截能力比较弱.

物理内存访问监控: Hypervisor 拥有细粒度的物理内存访问监控能力;而 TrustZone 无法做到物理内存的访问监控.

隔离能力: Hypervisor 的隔离能力主要基于 Staeg-2 地址翻译对物理内存的隔离,并需要解决 DMA 攻击的问题;相比之下,TrustZone 隔离能力更强大,可以安全地隔离物理内存和外设,隔离操作简单并且能完全抵御 DMA 攻击.

性能开销: Hypervisor 只要打开 Stage-2 地址翻译将给操作系统引入额外的性能开销;而 TrustZone 则不存在额外性能开销的问题.

商业应用场景:目前还没有利用 ARM 硬件虚拟化(Hypervisor)技术来加固商用智能手机的产品;而 TrustZone 作为比 ARM 硬件虚拟化技术出现更早的技术,已经广泛地运用于商用智能手机中,但它更多的是利用强大的隔离特性为商用智能手机提供安全服务(例如,指纹解锁),而监控商用智能手机操作系统的产品还相对较少,因为 TrustZone 的监控能力较弱.

TrustZone 的特点是隔离能力强、性能开销小和应用广;而 Hypervisor 的特点是处于非安全世界、拦截能力强、灵活的内存访问监控.本文提出的 HTrustZone 把 TrustZone 和 Hypervisor 的特点结合到了一起,充分利用它们的硬件特性,大大强化了 TrustZone 的监控能力.

9 HTrustZone 的设计

虽然 TrustZone 拥有比非安全世界的 Android 系统更高的权限,能够监控非安全世界操作系统的安全性,但是从安全监控的角度来看,TrustZone 存在三点不足:1) World gap, TrustZone 与被监控的 Android 系统之间存在 world gap,这使得 TrustZone 不能准确地获取非安全世界的信息;2) 指令和事件的拦截能力薄弱; 3) 物理内存的访问监控能力薄弱.而 HTrustZone 利用工作在非安全世界 Hypervisor 的特性,弥补了 TrustZone 的这三点不足.

图 2 描述了 HTrustZone 的系统结构(虚线方框),它横跨非安全世界和安全世界,包含 Hypervisor 和 TrustZone.非安全世界的 Hypervisor 工作在 HYP 模式下,安全世界的 TrustZone 工作在监控模式下.工作在监控模式 TrustZone 在 HTrustZone 系统中占主导,它可以动态地激活和关闭 Hypervisor.这种设计的好处是:1)消除了 world gap,因为 HTrustZone 既可以工作在非安全世界,又可以工作在安全世界;2)TrustZone 可以借助 Hypervisor 的特性来提高自身监控能力:当需要监控时,激活 Hypervisor 拦截 Android 系统的操作;当监控任务完成后,关闭 Hypervisor 以提高 Android 系统的整体性能.不过,在监控能力增强的同时,HTrustZone 也引入了两个安全性的问题.一是如何安全地启动 HTrustZone,即如何利用 TrustZone 安全地启动 Hypervisor;二是 Hypervisor 和 TrustZone 相互之间如何安全地切换.

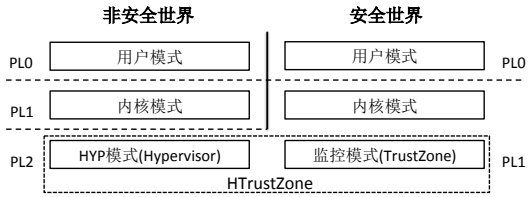

Fig.2 The Architecture of HTrustZone

图 2 HTrustZone 的架构

9.1 启动HTrustZone

HTrustZone 包含 TrustZone 和 Hypervisor 两个部分.TrustZone 从系统上电之后就一直处于工作状态下,而为了提高系统的整体性能,Hypervisor 在需要协助 TrustZone 监控 Android 系统操作时才被激活.因此,启动 HTrustZone 即利用 TrustZone 动态启动 Hypervisor.

动态启动 Hypervisor 需要完成以下两个工作:一是加载 Hypervisor 的代码到内存,代码加载可以由 TrustZone 或者 Android 内核来完成.前者具有更高的安全性,因为 TrustZone 是可信的,它存储和加载的操作也是可信的,但这种方法加重了 TrustZone 的负担;后者可以减轻 TrustZone 的负担,但由于 Android 内核是不可信的,它加载的操作也是不可信的,需要 TrustZone 来对加载的代码的完整性进行验证.本文选择 Android 内核来加载 Hypervisor 的代码;二是配置 Hypervisor 相关控制寄存器,因为 TrustZone 可以访问 Hypervisor 的控制寄存器,本文选择 TrustZone 直接操作 Hypervisor 的控制寄存器来初始化 Hypervisor.

图 3 描述了动态启动 Hypervisor 的步骤:首先,Android 内核为 Hypervisor 分配内存(步骤 1);接着 Android 内核把 Hypervisor 的代码加载到所分配的内存的指定位置(步骤 2);至此非安全世界的启动 Hypervisor 的准备工作完成.在步骤 3 中,非安全世界发起 SMC 请求进入监控模式(TrustZone),由 TrustZone 来初始化 Hypervisor、保护步骤 1 分配的内存和验证步骤 2 加载的 Hypervisor 代码的完整性.TrustZone 先往 Android 内核分配好的内存中填写 Stage-2 的页表数据(步骤 4)并设置需要保护的物理页所对应页表项的属性.然后 TrustZone 初始化 Hypervisor 相关寄存器,激活 Stage-2 地址翻译保护 Android 内核分配的内存(Hypervisor 空间)和 SMMU 页表(步骤 5). 最后,在步骤 6 中,由于 Hypervisor 的代码和 SMMU 页表数据都是固定的,TrustZone 分别计算它们的 HMAC 值,再与正确的 HMAC 值比较,以此来检查 Hypervisor 代码和 SMMU 页表数据的完整性.

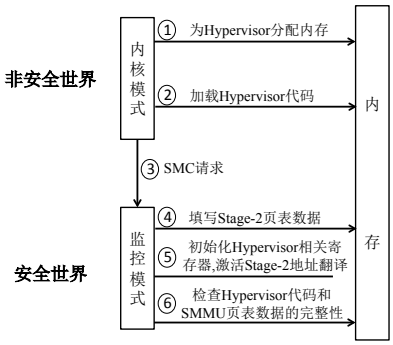

Fig.3 The Steps Of Dynamical HTrustZone Launch

图 3 动态启动 HTrustZone 的步骤

当这 6 个步骤完成并且完整性检查通过后,Hypervisor 就被激活了,Android 系统开始运行在虚拟机环境中.Stage-2 地址翻译的机制确保了 Hypervisor 所使用的物理内存不会被不可信的 Android 内核恶意修改,Hypervisor 自身的安全性得到了保障.当 TrustZone 需要监控 Android 系统运行或者检查 Android 系统运行状态时,它可以借助 Hypervisor 去设置监控点和检查点,以此提高监控能力.

9.2 TrustZone与Hypervisor的切换

HTrustZone 中,TrustZone 工作在安全世界的监控模式下,Hypervisor 工作在非安全世界的 HYP 模式下.TrustZone 和 Hypervisor 之间切换的安全性直接影响到 HTrustZone 系统的安全性.通常,进入 TrustZone 的方式是 Android 内核发起 SMC 请求;而进入 Hypervisor 的方式是内核发起 HVC 请求或者内核的操作产生 HYP 异常陷入 Hypervisor.

当 CPU 需要从 Hypervisor 的 HYP 模式切换到 TrustZone 的监控模式时,Hypervisor 可以发起 SMC 请求直接进入监控模式.这个过程中,不可信的 Android 内核无法介入,不能打断或者干涉这个操作,因此利用 SMC 请求从 Hypervisor 切换到 TrustZone 的过程是安全的.

当 CPU 需要从 TrustZone 的监控模式切换到 Hypervisor 的 HYP 模式时.由于 TrustZone 不能像 Android 内核一样通过 HVC 请求直接进入 Hypervisor,TrustZone 需要通过其他方式切换到 HYP 模式,方法有两种:第一种是先返回 Android 内核,再从内核发起 HYP 请求进入 Hypervisor;第二种是修改返回地址和 CPU 状态相关寄存器,直接返回到 Hypervisor.显然,第一种方法没有第二种方法安全,因为第一种方法中,不可信的 Android 内核参与了 TrustZone 切换到 Hypervisor 的过程,攻击者可以在 TrustZone 返回 Android 内核与内核发起 HYP 请求的这个间隙发起对 HTrustZone 的攻击(例如,跳过 HVC 请求).而第二种方法的模式切换对 Android 内核来说是一个原子操作,它没有办法干预,本文采用第二种更为安全的切换方式.

10 HTrustZone 的监控能力

HTrustZone 融合了 TrustZone 和 Hypervisor 两者的硬件特性,大大增强了 TrustZone 监控 Android 系统的能力.

10.1 World Gap

TrustZone 运行在安全世界,而被监控的 Android 系统运行在非安全世界,攻击者可以利用这个 world gap 发起一些特殊的攻击. CacheKit[14]是一个典型的利用 world gap 攻击的例子,它利用 Cache 的不一致性,即安全世界和非安全世界所使用的 Cache line 是完全隔离的,把恶意代码隐藏在非安全世界的 Cache 中,而非安全世界的 Cache 对 TrustZone 来说是透明的,因此它无法察觉攻击者隐藏在非安全世界 Cache 中的恶意代码. World gap 的存在从一定程度上限制了 TrustZone 对非安全世界的监控能力.

而 HTrustZone 可以完全抵御 world gap 的攻击,因为它横跨非安全世界和安全世界.当安全世界的 TrustZone 需要扫描非安全世界的 Cache 或者内存的时候,HTrustZone 中的 TrustZone 可以动态地启动 Hypervisor,再安全地切换到 Hypervisor 的模式并由 Hypervisor 去扫描非安全世界的 Cache 或者内存.而 Hypervisor 和 Android 系统都处于非安全世界且 Hypervisor 拥有更高的权限,Android 系统无法在 Cache 或者其他硬件中隐藏恶意代码,CacheKit 的攻击很容易被检查到.

10.2 拦截能力

TrustZone 对非安全世界敏感事件(例如,中断,系统调用)或者特殊指令的拦截能力的强弱与它的监控能力密切相关.而 TrustZone 的拦截能力主要体现在 SCR 寄存器的功能上.图 4 是对 SCR 功能的描述[13],从中可以看到,TrustZone 的拦截能力很弱,仅仅只能拦截非安全世界的各类中断(IRQ 和 FIQ)和外部错误(External Abort),这大大影响了 TrustZone 对非安全世界的监控能力.

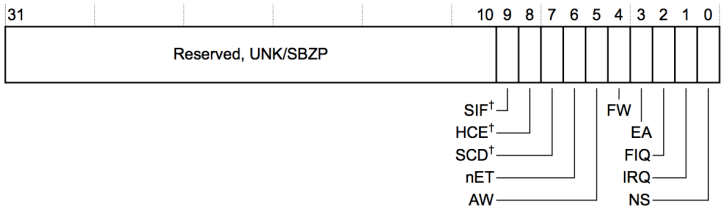

Fig. 4 SCR Register Bit Assignments

图 4 SCR 寄存器的位说明

Hypervision[6]提出用 TrustZone 取代传统的操作系统内核去管理非安全世界的内存模块.TrustZone 在为新的进程创建页表时,会检查内核所在的物理内存的访问权限和执行权限,检查是否存在多重页表映射等方式来保护物理内存.由于这些操作都是在 TrustZone 中完成的,攻击者难以介入,也难以绕过这些检查,因此基于内存的很多内核攻击手段就失效了.Hypervision 强化了运行时内核的保护,实用高效,被广泛地运用于三星公司生产的手机的 Android 内核中.但是,TrustZone 薄弱的拦截能力导致它无法拦截内核操作页表控制寄存器的相关指令.在 Hypervision 的实现方案中,内核中所有的操作页表控制寄存器的相关指令都被替换成了 SMC 指令,这样每当这些指令需要执行的时候都会陷进 TrustZone,TrustZone 会检查这些指令相对应参数的准确性并模拟这些指令的执行.指令替换从一定程度上弥补了 TrustZone 拦截能力不足的缺陷,然而这个方案存在三个缺点:一是需要修改内核,弱化了内核的兼容性和可移植性,在不支持 Hypervision 的设备上,系统就无法工作.当一个新的内核需要增加 Hypervision 的功能时,修改内核并替换指令是必不可少的工作.而且找出内核中所有的操作页表

控制寄存器相关的指令也不是一件容易的事;二是 TrustZone 需要识别出每个 SMC 指令所对应的页表控制寄存器操作指令,内核中原本可能就存在 SMC 指令,当 TrustZone 收到内核 SMC 请求的时候,首先它需要判断这是被替换之后的 SMC 指令,还是内核中原本就存在的 SMC 指令,再把替换的 SMC 指令解析成页表控制寄存器的操作指令,这无疑增加了 TrustZone 的负担;三是替换后的 SMC 指令安全性,即如何保证这些指令不被不可信内核恶意修改和这些指令不被跳过,需要额外的检查来保护 SMC 指令.

与 TrustZone 相比,Hypervisor 拥有强大的拦截能力,HCR(HYP Configuration Register)描述了 Hypervisor 的拦截功能(图 5). 除了中断拦截功能之外,Hypervisor 还可以通过配置 HCR 寄存器[13]拦截各类异常(包括系统调用),拦截页表相关控制寄存器的操作指令,拦截 TLB 和 Cache 的操作等等.对比图 4 和图 5,可以非常明显地看

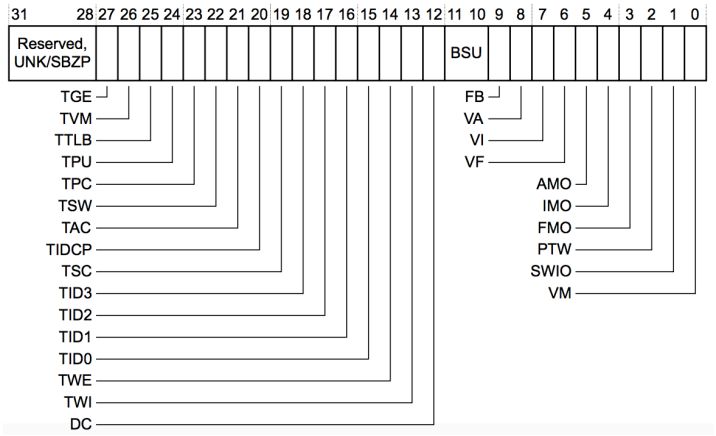

Fig. 5 HCR Register Bit Assignments

图 5 HCR 寄存器的位说明

到 HCR 的功能比 SCR 丰富很多.

而 HTrustZone 拥有"SCR+HCR"的功能,可以更有好地拦截 Android 系统的操作.HTrustZone 可以利用 HCR.TVM 的功能来实现拦截页表控制寄存器的操作.当 HCR.TVM 位置为 1,页表控制寄存器操作将产生一个 HYP 的异常而陷入 Hypervisor. Hypervisor 可以发起 SMC 请求,再由 TrustZone 中的 Hypervision 来处理这个操作.TrustZone 处理完这个操作后,如果直接执行 ERET 返回,那么 CPU 模式又会切换回 HYP 模式,最后由 HYP 模式返回内核,继续执行内核的指令.但是这样的做法显得不够高效.本文的方案是:TrustZone 在执行 ERET 返回之前,修改 spsr_mon 和 lr_mon 寄存器直接返回到内核模式,这样可以减少一次上下文的切换.由于 HCR.TVM 置 1 之后,任何有效的页表控制寄存器的写操作都会陷入 Hypervisor,因此 HTrustZone 可以简单高效地拦截页表控制寄存器的操作,避免了 Hypervision 实现过程中繁杂的指令替换操作.HTrustZone 中的 TrustZone 可以识别出 SMC 请求是来自 Hypervisor 还是非安全世界的内核;而且它能够识别触发异常指令的编码并解析出对应的页表控制寄存器操作指令.这大大减轻了 TrustZone 的负担.HCR.TVM 置 1 的拦截方式充分利用了硬件虚拟化技术的特点,对于 Android 系统来说透明的,不存在指令替换方案中被攻击的问题.除此之外,HTrustZone 也可以拦截非安全世界的系统调用,用户态和内核态的切换等事件.例如,通过拦截系统调用,解析 binder 数据传输,HTrustZone 可以通过类似 CopperDroid[15]的方法推测 Android 软件的行为.

10.3 物理内存访问的监控

通过设置 TZASC 和 TZMA 寄存器,TrustZone 可以把连续的几块物理内存设置为安全内存.而安全的物理内存是不允许非安全世界访问的,即便是非安全世界以 DMA 的方式去访问安全内存也会产生不可预知的错误.因此 TrustZone 可以实现对物理内存的隔离保护,但是它不能通过设置安全内存的方式来监控非安全世界对该内存的访问,需要非安全世界操作系统的内核模块来协助监控内存的访问,这种方式又引入了内核模块安全性的问题.而内存为系统的运行提供了代码和数据,记录了当前系统的操作和行为.物理内存的访问监控也是 TrustZone 对非安全世界监控能力的一种直接体现.

HTrustZone 拥有强大的物理内存访问监控能力.正如第一章所提到的,Hypervisor 激活 Stage-2 页表翻译机制之后,可以支持页粒度物理内存的访问监控.HTrustZone 的 TrustZone 可以借助 Hypervisor 这一特性实现物理内存的访问监控.不仅如此,Hypervisor 还可以通过配置 debug 寄存器来实现指令级物理地址的访问监控.不过,ARM 设备上的 debug 寄存器数目相对有限,同一时间能指定的物理地址数目也是有限的(例如,Raspberry Pi2 上 debug 寄存器的数目是 8 个).Hypervisor 中的这些监控物理内存访问的设置对于非安全世界的操作系统来说也是透明的.

当 HTrustZone 中的 TrustZone 需要监控 Android 系统对物理内存的访问时,它可以动态启动 Hypervisor 并在 Stage-2 页表项上设置好需要监控的物理页的属性.当 Android 系统违反访问物理页面访问属性的时候,会产生 Stage-2 的页错误(page fault)异常直接陷进 Hypervisor,在页错误的的处理函数中,Hypervisor 执行 SMC 指令进入监控模式,让 TrustZone 来处理这个异常.

通过配置 debug 的相关寄存器,TrustZone 可以让 debug 异常直接陷进 Hypervisor.当 debug 的目标地址被访问时,产生的 debug 异常将陷进 Hypervisor,同样的在 Hypervisor 的异常处理函数中,执行 SMC 指令进入监控模式,由 TrustZone 来处理 debug 异常.在 ARM 的架构中,异常向量表作为内核的入口通常是在固定的虚拟地址上,系统调用的入口位于偏移为 0x8 的位置,中断的入口位于偏移为 0x14(IRQ)和 0x18(FIQ)的位置.如果把 debug 寄存器设置为系统调用入口的虚拟地址,那么 TrustZone 可以拦截所有的系统调用;如果两个中断入口的虚拟地址都设置断点,那么 TrustZone 可以拦截所有的中断.

11 系统实现

我们在 Raspberry Pi2 开发板[16]上实现了 HTrustZone 原型系统.在 Raspberry Pi2 开发板上,Android5.1 运行在非安全世界的用户模式和内核模式下.HTrustZone 包含 Hypervisor 和 TrustZone 两部分,其中 Hypervisor 运行在非安全世界的 HYP 模式,TrustZone 运行在安全世界的监控模式下.HTrustZone 原型系统的实现主要包含三部分内容:启动 HTrustZone、HYP 模式和监控模式的相互转换和 Hypervisor 的拦截功能.本章将介绍这三部分的实现细节.

11.1 启动HTrustZone

在 2.1 中,本文提出了启动 HTrustZone 的方法,其中核心内容是 TrustZone 动态启动 Hypervisor.首先,Android 内核为 Hypervisor 分配内存,用来加载 Hypervisor 的代码、分配 Hypervisor 栈和存储 Stage-2 页表数据.Hypervisor 代码的加载是由 Android 内核来完成的,而栈分配和页表数据的填写由 TrustZone 来完成.

因为在动态启动 Hypervisor 之后仍然需要保证 Android 系统的正常运行,所以 Stage-2 页表管理的中间地址(IPA)到物理地址的映射必须是一一映射以维持虚拟地址依然能够映射到正确的物理地址.Stage-2 的页表采用 3 级长描述型页表(long-descriptor translation format)[13].由于 Android 内核所能分配的最大的连续内存大小是 4M, 考虑到 Stage-2 页表数据大小,内核需要为 Hypervisor 分配 3 块连续的 4M 内存.如图 6 所示,内存的布局为:第一块内存存放第一级和第二级 Stage-2 的页表数据,Android 内核可以把 Hypervisor 的代码也加载到第一块内存中,剩下的内存可以作为 Hypervisor 运行时的栈来使用.第二块和第三块内存存放第三级 Stage-2 的也表数据.虽然 Raspberry Pi2 的内存大小是 1G,但是由于 IO 内存的存在,最高的物理地址远大于 0x40000000,为

了确保所有的中间地址(IPA)都有映射,我们映射了 4G 的 IPA 地址空间.因此第三级 Stage-2 的页表数据大小是 8M.

当 Android 内核为 Hypervisor 分配好内存之后,执行 SMC 指令通知 TrustZone 初始化 Hypervisor.TrustZone 先根据 Android 内核所分配内存的物理地址,填写 Stage-2 的三级页表数据到对应的物理内存位置.接着初始化 Hypervisor 相关控制寄存器[13],并通过配置 VTCR(Virtualization Translation Control Register), VTTBR(Virtualization Translation Table Base Register),HVBAR(HYP Vectors Base Address Register)和 HCR 寄存器,激活 Stage-2 地址翻译,设置 Stage-2 的页表项来保护 Hypervisor 的空间(Android 内核分配的 12M 内存).由于 Raspberry Pi2 硬件上不支持 SMMU,因此 Hypervisor 需要其他方式来抵御 DMA 攻击(例如,拦截 IO 操作,检查 DMA 操作).最后 TrustZone 计算 Hypervisor 代码的 HMAC 值来检查代码和数据的完整性.如果检查通过了,TrustZone 可以选择返回 Android 内核,也可以选择返回到 HYP 模式来执行 Hypervisor 的代码,至此动态启动 Hypervisor 的工作就完成了,HTrustZone 开始工作.

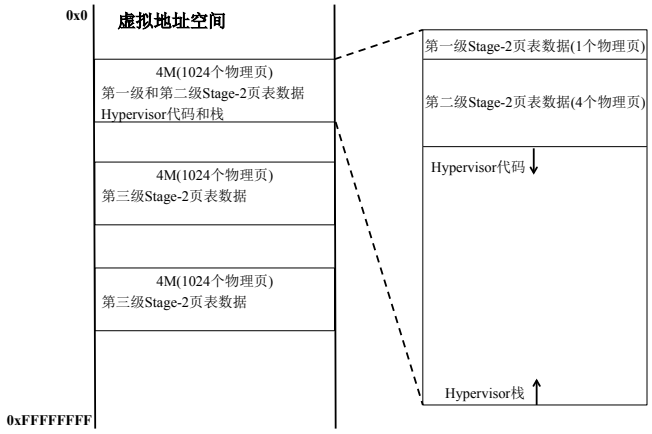

Fig.6 Hypervisor Memory Layout

图 6 Hypervisor 的内存布局

11.2 TrustZone与Hypervisor的切换

当 HTrustZone 工作时,TrustZone 和 Hypervisor 之间的相互切换是必不可少的.例如,在 CacheKit 的检测方案中,TrustZone 需要从监控模式切换到 Hypervisor 的 HYP 模式来扫描内存和 Cache 中是否存在恶意代码.在 ARMv7 中,当 Android 内核执行 SMC 指令的时候,CPU 的模式由非安全世界的内核模式切换到安全世界的监控模式,CPSR(Current Program Status Register)寄存器备份在 spsr_mon 寄存器中,返回地址(SMC 的下一条指令)保存在 lr_mon 寄存器中.当在监控模式下,执行 ERET 指令返回时,CPSR 从 spsr_mon 寄存器中恢复,其中 CPSR.M[4:0]表示当前 CPU 的工作模式.由于 SMC 指令是从 Android 内核发起,因此 CPSR.M 被恢复成非安全世界内核模式.PC 寄存器指向 lr_mon 寄存器中保存的地址.为了实现从 TrustZone 返回 HYP 模式,需要备份 spsr_mon 寄存器并把 spsr_mon.M[4:0]的内核模式修改为 HYP 模式,再备份 lr_mon 寄存器并把 lr_mon 修改成 Hypervisor 对应扫描内存的函数入口地址,然后再执行 ERET 指令.Hypervisor 响应完 TrustZone 的请求之后,把备份的 spsr_mon 和 lr_mon 寄存器恢复到到 spsr_hyp 和 elr_hyp 寄存器中,执行 ERET 指令返回到内核模式继续执行内核的指令.

图 7 是对这个过程的小结:1)内核执行 SMC 指令进入监控模式;2)监控模式下修改 spsr_mon 和 lr_mon 寄存器返回 HYP 模式;3)HYP 模式下修改 spsr_hyp 和 elr_hyp 寄存器回到内核模式.

在 HTrustZone 中,从 Hypervisor 切换到 TrustZone 也是非常常见的.例如,Hypervision 的优化方案和物理内存的访问监控,Hypervisor 需要先拦截特殊指令的执行或者物理内存访问的事件,再从 HYP 模式切换到监控模式,进入 TrustZone,并由 TrustZone 去进行相应的处理.HTrustZone 在初始化 Hypervisor 的时,激活特殊指令的和物理内存访问事件的拦截功能,一旦 Android 内核执行特殊指令或者违反物理内存的访问规则,CPU 将陷入 HYP 模式.而在 HTrustZone 中,Hypervisor 的功能是辅助 TrustZone 拦截 Android 内核的操作,因此 Hypervisor

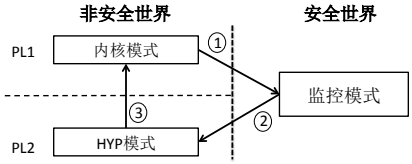

Fig.7 Context Swith From Monitor Mode To HYP Mode

图 7 从监控模式到 HYP 模式的上下文切换

在处理函数中,执行 SMC 指令进 TrustZone,同时把备份的 spsr_hyp 寄存器的值和 elr_hyp 寄存器的值保存到通用寄存器中.通过 HSR(Hyp Syndrome Register)寄存器,TrustZone 能够解析出 Hypervisor 所拦截的事件是对指令的监控还是对物理内存访问的监控.如果是指令监控,TrustZone 解析出该指令并模拟指令的执行.如果和物理内存的访问监控相关,通过 HDFAR(HYP Data Fault Address Registr)寄存器,TrustZone 能够得到内存放问的目标地址,从而采取相对应的处理.当 TrustZone 处理完成后,把 spsr_mon 和 lr_mon 分别修改成保存的 spsr_hyp 和 elr_hyp 的值,执行 ERET 指令回到 Android 内核继续执行其他指令.

图 8 是对这个过程的小结:1)Hypervisor 拦截内核事件或者指令;2)Hypervisor 执行 SMC 指令进入监控模式;3)监控模式下修改 spsr_mon 和 lr_mon 寄存器回到内核模式

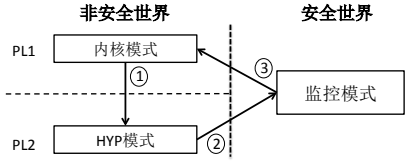

Fig.8 Context Switch From HYP Mode To Monitor Mode

图 8 从 HYP 模式到监控模式的上下文切换

11.3 HTrustZone的拦截能力

在 2.2 中,我们比较了 TrustZone 和 Hypervisor 的拦截能力,发现 Hypervisor 拥有比 TrustZone 更为强大的拦截能力,因此 HTrustZone 的拦截能力主要体现在 Hypervisor 上,而 Hypervisor 拦截 Android 内核操作的方法主要有三种:1)设置 HCR 寄存器的相关功能位;2)设置 Stage-2 页表项的物理页访问属性;3)设置 debug 寄存器.TrustZone 可以访问相关寄存器和 Stage-2 所在物理页的页表数据来激活这三种拦截功能.

设置 HCR 寄存器:图 3 介绍了 HCR 寄存器的每一位,从中可以看到 HCR 寄存器的功能非常丰富.例如,HCR.TGE 置 1 可以拦截用户模式到内核模式的切换;HCR.TVM 置 1 可以拦截页表相关控制寄存器的操作;HCR.TTLB 置 1 可以拦截 TLB 的操作;HCR.TSC 置 1 可以拦截 SMC 指令的执行等等.不过 HCR 寄存器往往针对的是拦截一类事件,一但某个拦截功能被打开,Android 内核会频繁地陷进 HYP 模式,其中可能会引入大

量的噪音,因此 Hypervisor 或者 TrustZone 需要从中过滤出真正需要拦截的事件.

设置 Stage-2 页表项: Stage-2 地址翻译的页表项包含读、写、执行 3 个物理内存访问属性位.设置第三级页表项的这三个属性位可以实现页粒度的物理内存读、写、执行的访问监控.页表项的第 6 和 7 位为读写位,第 54 位为执行位.由于我们设计的 Stage-2 页表映射是一一映射,因此物理页的地址与该页所对应的第三级页表项的地址是线性关系.当 TrustZone 需要监控或者保护某个物理页的时候,根据物理页的地址即可方便得定位到对应页表项所在的地址,并根据监控或者保护的需求设置 3 个属性位.一旦 Android 内核违反物理页访问规则,系统将产生 Stage-2 的 page fault 而陷进 Hypervisor.此时,如果是 Android 内核的正常操作触发了 page fault,Hypervisor 或者 TrustZone 不能在物理页访问属性未修改的情况下返回到触发 page fault 的指令,因为这样的操作会再一次触发 page fault 而出现死循环.Hypervisor 或者 TrustZone 的处理方式有两类:1)修改物理页访问属性并返回(适合一次监控);2)不修改物理页访问属性模拟执行或者跳过触发 page fault 的指令返回到下一条内核指令(适合永久监控).这样 HTrustZone 可以实现细粒度的物理内存监控和保护.

设置 debug 寄存器: 在敏感指令或者数据上设置断点的方法可以提供指令级的物理内存访问监控.通常情况下,设置 DGBBCR(Breakpoint Control Register)和 DBGBVR(Breakpoint Value Register)这两个寄存器可以在 Android 内核指令上设置断点.当 CPU 读取该指令的时候,断点将触发并产生一个异常陷入到内核的异常处理函数中.但是当 HDCR(HYP Debug Configuration Register)的 TDE 位被置 1 的时候,该异常会直接陷入 HYP 模式.同时 HDCR.TDA 置 1 可以防止 Android 内核修改 DGBBCR 和 DBGBVR 的配置以保证断点的有效性和断点地址不被修改.当 TrustZone 需要监控内核的某个指令的时候,可以通过配置 debug 寄存器,借助 Hypervisor 的拦截功能来实现.

12 实验结果

在 Raspberry Pi2 开发板上, 我们对 HTrustZone 系统运行的性能进行了测试.Raspberry Pi2 是 ARMv7 架构下的开发板,它搭载了四个 Cortex-A7 CPU 和 1GB 的内存,其中 CPU 的主频是 900MHz.运行在 Raspberry Pi2 开发板非安全世界的操作系统是 Android 5.1 Lollipop,内核版本为 4.1.3.HTrustZone 框架系统的可信计算基非常小,一共仅包含 143 行汇编码和 248 行 C 代码,其中 Hypervisor 包含 63 行汇编码,TrustZone 包含 79 行汇编码和 248 行 C 代码.Hypervisor 和 TrustZone 自身代码的安全性可以通过形式化验证的方法来保证.HTrustZone 框架系统只包含动态启动 Hypervisor 和 Hypervisor 与 TrustZone 之间相互切换这两部分功能,不包含 TrustZone 对监控事件检查和处理的功能.而 HTrustZone 框架系统是可扩展的,可以根据需求在 HTrustZone 中设置监控点并添加相应的处理功能.我们测试了 HTrustZone 系统启动所需要的时间、上下文切换的时间和虚拟化环境下 Android 系统的整体性能并与传统的 TrustZone 监控方法做了比较.

12.1 HTrustZone启动

通常,TrustZone 的初始化是在系统启动阶段、Android 内核初始化之前完成的[17-19],在我们的实验环境中也是如此.而 HTrustZone 是在运行时(run-time)启动的,引入了额外的运行时启动的性能开销.HTrustZone 系统的启动主要包含 4 个步骤:1)Android 内核发起 SMC 请求,切换上下文;2)TrustZone 准备 Stage-2 页表数据;3)TrustZone 计算 Hypervisor 代码的 HMAC 值,检查代码完整性;4)TrustZone 初始化 Hypervisor 完成并返回 Android 内核.其中主要的时间消耗是在第二步和第三步.从表 1 中,我们可以看到 HTrustZone 的一次启动时间(从发起 SMC 请求到返回 SMC 的下一条指令)是 18.064 毫秒,时间很短,对用户来说几乎感觉不到. TrustZone 往内存中填写超过 8M 的三级 Stage-2 页表数据需要 14.498 毫秒,计算 Hypervisor 代码 HMAC 值需要的时间是 3.451 毫秒,而这两者的时间之和与 HTrustZone 启动的总时间已经十分接近.

Table 1 HTrustZone Lauch Lantency(unit: ms)

表 1 HTrustZone 启动时间(单位: 毫秒)

准备 Stage-2 页表数据	计算 HMAC 值	HTrustZone 启动时间
14.498	3.451	18.064

12.2 上下文切换

在传统的仅仅依赖 TrustZone 监控非安全世界操作系统运行的模式中,只有一种上下文切换的模式,即非安全世界内核模式与安全世界监控模式之间的切换.通常,从非安全世界切换到 TrustZone 是通过 SMC 调用实现的[6,7,8,20],因此监控的开销只是一次 SMC 调用,即 247.4ns(表 2).

相比之下,HTrustZone 系统涉及到多个上下文切换,包括简单的切换和复杂的切换.其中简单的切换是指非安全世界的内核模式和 HYP 模式的切换(HVC 指令)、非安全世界和安全世界的切换(SMC 指令),我们测试了一次空的 HVC 调用和 SMC 调用所需要的时间.复杂的切换是指 Android 内核模式调用 SMC 指令切换到安全世界监控模式,监控模式返回非安全世界 HYP 模式,最后由 HYP 模式回到 Android 内核模式和 Android 内核模式调用 HYP 指令切换到 HYP 模式,HYP 模式再调用 SMC 指令进入安全世界监控模式,最后由监控模式返回到 Android 内核.表 2 中,"HVC 调用"测试的是从 Android 内核发起一次 HVC 调用开始到 HVC 调用返回到 Android 内核所需要的时间,这里包含两次上下文切换;"SMC 调用"测试的是从 Android 内核发起一次 SMC 调用开始到 SMC 调用返回到 Android 内核所需要的时间,它也包含两次上下文切换(world switch)."内核_监控_HYP_内核"测试的是从 Android 内核模式发起一次 SMC 调用进入监控模式,监控模式修改相关寄存器返回 HYP 模式,最后由 HYP 模式通过修改相关寄存器返回内核模式所需要的时间,这里存在三次上下文切换."内核_HYP_监控_内核"测试的是从 Android 内核模式发起一次 HYP 调用进入 HYP 模式,HYP 模式再发起 SMC 调用进入监控模式,最后由监控模式通过修改相关寄存器返回内核模式所需要的时间,它也包含三次上下文切换.

从表 2 中可以看到,在 Raspberry Pi2 开发板上一次 HVC 调用所需要的时间是 39.2ns,而一次 SMC 调用所需要的时间则是 247.4ns.一次 SMC 调用所需要的时间是一次 HVC 调用所需要时间的近 10 倍.其中的原因是 HVC 调用只是在非安全世界下两个不同的 CPU 模式之间的切换,而 SMC 调用不仅是在 CPU 不同模式之间的切换,而且是在两个不同世界的切换(world switch),会涉及更多硬件处理.

在 HTrustZone 中,由于 TrustZone 需要借助 Hypervisor 的功能增强监控能力,原本仅仅依赖 SMC 调用进行的上下文切换将变成"内核_监控_HYP_内核"或"内核_HYP_监控_内核"这两种模式的切换.在监控功能增强的同时也引入了额外的开销,但相比于 SMC 调用,"内核_监控_HYP_内核"的额外开销是 58.7ns;而"内核_监控_HYP_内核"的额外开销是 60.0ns.额外开销都很小,这主要得益于 HVC 调用比 SMC 调用快得多.

Table 2 Context Switch Latencies(unit: ns)

表 2 上下文切换时间(单位: 纳秒)

		TrustZone	HTrustZone	
SMC 调用	HVC 调用	SMC 调用	内核_监控_HYP_内核	内核_HYP_监控_内核
247.4	39.2	247.4	306.1	307.4

12.3 Android系统的整体性能

当 ARM 平台上只有非安全世界的 Android 系统在工作的时,内存地址只需要进行一层翻译(虚拟地址到物理地址);而当 Android 系统和安全世界的 TrustZone 同时在 ARM 平台工作的时候,由于这两者运行在不同的世界,彼此之间完全隔离,在 Android 系统不发起 SMC 调用和 TrustZone 不主动打断 Android 系统运行的情况下,Android 系统仍可以保持原有的系统性能,即 TrustZone 的运行不会给 Android 系统带来额外的性能开销.但是当 HTrustZone 工作时,Hypervisor 被激活,为保护它自身的空间, Stage-2 地址翻译被打开.这对 Android 系统来说,原本虚拟地址到物理地址的一层内存地址翻译就变成了虚拟地址到中间地址(IPA)再到物理地址的两层内存地址翻译,额外的性能开销(Stage-2 地址翻译)就被引入了.我们在 Hypervisor 未被激活和 Hypervisor 激活两种状态下,在 Raspberry Pi2 开发板运行了 Vellamo 和 CF-bench 两个 benchmark 来测试 Android 系统的整体性能.

表 3 中记录了在 Hypervisor 未激活(TrustZone)和激活(HTrustZone)状态下,Android 系统运行两个 benchmark 的得分,得分越高性能越好.其中,Vellamo 的 Multicore 测试项测试的是浮点运算、内存读写、系统

调用和并行计算的系统性能;Metal 项测试的是 CPU 和网络的性能.CF-bench 测试的是内存读写和磁盘读写分别在本地和 Dalvik 虚拟机中的性能,并在最后给出了总体的性能.从表中的数据我们可以看到,Hypervisor 激活状态下,即 Stage-2 地址翻译的额外开销为(0.0%-3.6%)几乎可以忽略不计.如果只从地址翻译的角度考,理论上二层地址翻译的开销应该是一层地址翻译开销的两倍.但是由于 TLB(Translation Looksides Buffer)的存在,使得 Stage-2 地址翻译的开销大大降低,因为并不是每一次地址翻译都要去走两层 MMU,绝大多数情况都可以直接从 TLB 缓存中命中.

Table 3 Benchmark Scores

表 3 Benchmark 结果

	TrustZone	HTrustZone
Vellamo:		
Multicore	552.9	548.7(0.8%)
Metal	285.2	274.9(3.6%)
CF-bench:		
Native	12252.2	12212.1(0.3%)
Java	3828.1	3831.6(0.0%)
Overall	7273.3	7183.3(0.7%)

12.4 HTrustZone与TrustZone监控能力的比较

现有的 TrustZone 的监控系统[6,7,8,20],都只工作在安全世界,无法抵御 world gap 的攻击;对于指令的拦截,通常需要用 SMC 指令替换所要拦截的指令;而且只能实现物理内存的隔离,无法实现物理内存的访问监控.而 HTrustZone 结合了 TrustZone 和 Hypervisor 的硬件特性,可以有效地抵御 world gap 的攻击;借助 Hypervisor 的拦截功能可以实现对控制指令的拦截;激活 Stage-2 地址翻译为需要监控的物理内存设置访问属性,HTrustZone 能够监控 Android 系统对物理内存的访问.表 4 对比了 HTrustZone 和 TrustZone 的监控能力,HTrustZone 在对 Android 系统的监控能力上较 TrustZone 具有明显的优势.

Table 4 Introspection Capability Comparisons Between HTrustZone And TrustZone

表 4 HTrustZone 与 TrustZone 监控能力的比较

	TrustZone	HTrustZone
抵御 world gap 攻击	×	√
指令拦截	×	√
物理内存访问监控	×	√

13 相关工作

13.1 TrustZone监控系统

在 ARM 平台上,已有多种基于 TrustZone 的监控系统被提出来解决数据安全性审查、内核代码完整性加固、运行时内核代码保护等问题. Jang.等人提出了利用 TrustZone 来检查非安全世界操作系统的数据安全性[20].Ge.等人提出 SPROBES[7],利用 TrustZone 维护 5 条内核安全属性,以此来保护操作系统内核代码的完整性.Azab.等人提出的 Hypervision[6]在运行时对内核代码进行保护,它是三星 KNOX 技术的核心[8,17].这些监控系统利用 TrustZone 安全的执行环境和高权限,可以有效地检查数据的安全性和代码的完整性.但是,它们都需要修改操作系统内核,并在监控点添加 SMC 指令或者向内核添加一个内核模块以确保 TrustZone 可以在监控点处对操作系统进行安全性检查.这样的解决方案又引入了如何保证 SMC 指令不被跳过和如何保护内核模块的安全问题.而 HTrustZone 监控系统不需要修改操作系统的内核就可以实现监控的功能,它可以通过指令拦截、在 Stage-2 页表项上设置物理内存访问属性、设置 debug 寄存器等多种方式自由地设置监控点,并且这些监控点对于操作系统是透明的,因此操作系统无法修改或绕过这些监控点.利用 Stage-2 地址翻译,HTrustZone 可以有效地保护非安全世界的 Hypervisor 空间的安全性.

13.2 动态启动Hypervisor

在[18]中,Cho.等人提出了一种动态启动 Hypervisor 的方法:他们在内存高地址中预留了 128M 空间存放 Hyperviosr的代码和数据,当Hypervisor没有被激活的时候,这段内存被设置为安全内存以访问不可信内核的恶意修改;当 Hypervisor 被激活时,这段内存就被设置为非安全世界的非安全内存并受到 Stage-2 地址翻译的保护.Android 内核可以通过执行 SMC 指令进入 TrustZone,并由初始化 Hypervisor 的相关寄存器,打开 Staeg-2 地址翻译,动态修改这 128M 物理地址的属性为非安全世界的物理内存以保证 Hypervisor 的代码能顺利地在非安全世界执行.这种动态启动 Hypervisor 方法存在两点不足:一是在 Hypervisor 激活状态和非激活状态下,Android 内核都不能使用预留的内存空间,降低了内存的利用率;二是 TrustZone 不仅需要初始化 Hypervisor,还需要把 Hypervisor 的代码和数据加载到预留的内存中,这无疑增加了 TrustZone 的负担.而本文和[21]中所采用的动态启动 Hypervisor 的方法借助 Android 内核动态内存管理的功能并由 Android 内核加载 Hypervisor 的代码和数据.由于 Android 内核是不可信的,它所加载的 Hypervisor 的代码和数据必须经过 TrustZone 的验证以保证动态启动的安全性.相比于 Cho.等人的方法,本文的方法提高了内存的利用率,简化了 TrustZone 的工作.

13.3 Intel SGX技术

TrustZone 技术为运行在 ARM 平台上的操作系统提供了可信的隔离环境,可以抵御来自隔离环境外部的攻击.在 Intel 平台上,SGX(Software Guard Extensions)技术为开发者提供了一个可信的执行环境(Enclave),通过特殊的指令,开发者可以把指定的代码放到 Enclave 中来运行.SGX 技术保证在 Enclave 外部的程序无法访问 Enclave 内部的代码和数据,即便是该程序拥有高权限(内核权限或者 Hypervisor 权限)[22].TrustZone 技术和 SGX 技术的共同点是都提供了可抵御内核或者更高级别攻击的可信执行环境,不同点是 TrustZone 技术引入了新的 CPU 运行模式,通过控制寄存器的设置把硬件资源划分为两部分 secure 和 non-secure;TrustZone 和 Rich OS 分别运行在两个世界,相互独立.而 SGX 技术并没有引入新的 CPU 运行模式,而是借助内核动态地创建 Enclave 的执行环境,当用户态的程序执行特殊指令的时,就会进入 Enclave 执行 Enclave 内部的代码.开发者可以非常自由地把敏感的代码放到 Enclave 中去执行,这是 TrustZone 无法做到的.Enclave 和 Rich OS 更像是捆绑在一起的关系,而不是像 TrustZone 那样相互独立,因此利用 SGX 技术可以更加方便和有效地介入和监控 Rich OS 的运行.与此同时,和 Rich OS 密切的联系也为 Enclave 埋下了容易受到侧信道(side-channel)攻击的隐患,如基于 page fault[23][24]和基于 cache 的侧信道攻击.

13.4 Hypervisor、TrustZone的安全性

ARM 的虚拟化扩展和安全扩展从硬件上为 Hypervisor 和 TrustZone 提供了安全支持,大大增加了从用户态或者内核态对 Hypervisor 或 TrustZone 进行攻击的难度.目前主流的从外部攻击 Hypervisor 和 TrustZone 的方法是通过侧信道[25][26].然而 Hypervisor 或者 TrustZone 侧信道信息的获取并没有像获取 Enclave 内部侧信道信息那么容易,因此在 Hypervisor 或者 TrustZone 进行侧信道攻击虽然是可行的,但难度非常之大.攻击者更多的是选择从 Hypervisor 和 TrustZone 内部来进行攻击.随着 Hypervisor 和 TrustZone 的功能的复杂化,相应的代码量也快速增加,代码中的漏洞就不可避免地出现了[27-29].攻击者正是利用这些漏洞,获取系统的高权限并完成相应的攻击.这类攻击的防护是非常困难的,比较有效的防护措施是对 Hypervisor 和 TrustZone 的代码做形式化验证,但是目前形式化验证技术只适用于代码量较小的系统或软件,而商用 Hypervisor 和 TrustZone 的代码量非常庞大.

由于 ARM 平台上的 Hypervisor 并没有得到广泛得运用,而主流的 ARM 设备都是支持虚拟化(支持 Hypervisor)的,这些支持 Hypervisor 却没有激活 Hypervisor 的设备可能会受到一类新型的攻击,那就是攻击者可以利用内核漏洞动态地启动一个恶意的 Hypervisor 并发起类似 VMI(Virtual Machine Introspection)形式的攻击.

14 结束语

本文结合 ARM 硬件虚拟化特性和 TrustZone 技术特性,首次提出了一种可扩展框架系统 HTrustZone,能借

助 Hypervisor 提高 TrustZone 监控能力,解决了 HTrustZone 安全启动、Hypervisor 与 TrustZone 之间安全切换的问题.相比于传统的 TrustZone 监控系统,HTrustZone 系统拥有更为丰富的监控方式和更强大的监控能力,可以为被监控的操作系统提供更好的安全性支持.实验的结果也表明,HTrustZone 框架系统在系统启动、上下文切换和操作系统整体性能上的额外开销是可接受的.

References:

[30] W. Xu, J. Li, J. Shu, W. Yang, T. Xie, Y. Zhang, and D. Gu. From Collision To Exploitation: Unleashing Use-After-Free Vulnerabilities in Linux Kernel. In Proceedings of the 22nd ACM Conference on Computer and Communications Security, CCS 2015.

[31] Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution. In Proceedings of the 2012 IEEE Symposium on Security and Privacy, S&P 2012.

[32] Nitay Artenstein and Idan Revivo. Man in the Binder: He Who Controls IPC, Controls the Droid. Black Hat, 2014.

[33] W. Li, H. Li, H. Chen, and Y. Xia. Adattester: Secure online mobile advertisement attestation using trustzone. In Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services, MobiSys 2015.

[34] H. Sun, K. Sun, Y. Wang, J. Jing, and S. Jajodia. Trustdump: Reliable memory acquisition on smartphones. In Computer Security-ESORICS, 2014.

[35] A. M. Azab, P. Ning, J. Shah, Q. Chen, R. Bhutkar, G. Ganesh, J. Ma, and W. Shen. Hypervision across worlds: Real-time kernel protection from the arm trustzone secure world. In Proceedings of the 21st ACM Conference on Computer and Communications Security, CCS 2014.

[36] X. Ge, H. Vijayakumar, and T. Jaeger. SPROBES: Enforcing kernel code integrity on the trustzone architecture. In Proceedings of the 2014 Mobile Security Technologies (MoST) workshop, 2014.

[37] Samsung. White paper: An overview of Samsung KNOX, 2013.

[38] J. Jang, S. Kong, M. Kim, D. Kim, and B. B. Kang. Ssecret: Secure channel between rich execution environment and trusted execution environment. In 22nd Annual Network and Distributed System Security Symposium, NDSS 2015.

[39] H. Sun, K. Sun, Y. Wang, and J. Jing. Trustotp: Transforming smartphones into secure one-time password tokens. In Proceedings of the 22nd ACM Conference on Computer and Communications Security, CCS 2015.

[40] L. Guan, P. Liu, X. Xing, X. Ge, S. Zhang, M. Yu and T. Jaeger. TrustShadow: Secure Execution of Unmodified Applications with ARM TrustZone. In Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services, MobiSys 2017.

[41] ARM. ARMv6-m architecture reference manual. Technical report, 2007.

[42] ARM. Architecture reference manual ARMv7-a and ARMv7-r edition. Technical report, 2014.

[43] N. Zhang, H. Sun, K. Sun, W. Lou and Y. Hou. Cachekit: evading memory introspection using cache incoherence. In: 2016 IEEE European Symposium on Security and Privacy. Euro S&P 2016.

[44] K. Tam, S. J. Khan, A. Fattori, and L. Cavallaro. CopperDroid: Automatic Reconstruction of Android Malware Behaviors. In Proceedings of the 22nd Annual Network and Distributed System Security Symposium, NDSS 2015.

[45] Raspberry Pi2 development board. https://www.raspberrypi.org/products/raspberry-pi-2-model-b/

[46] U. Kanonov and A. Wool. Secure Containers in Android: the Samsung KNOX Case Study. The Workshop on Security and Privacy in Smartphones and Mobile Devices. SPSM 2016.

[47] Y. Cho, J. Shin, D. Kwon, M. J. Ham, Y. Kim, and Y. Paek. Hardware-assisted on-demand hypervisor activation for efficient security critical code execution on mobile devices. In USENIX ATC, 2016.

[48] H. Sun, K. Sun, Y. Wang, J. Jing, and H. Wang. TrustICE: Hardware-assisted Isolated Computing Environments on Mobile Devices. In Proceedings of The 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks. DSN 2015.

[49] J. Williams. Inspecting data from safety of your trusted execution environment. Black Hat 2015.

[50] Z. Zhang, X. Ding, G. Tsudik, J. Cui, and Z. Li. Presence Attestation: The Missing Link in Dynamic Trust Bootstrapping. In Proceedings of the 24nd ACM Conference on Computer and Communications Security, CCS 2017.

[51] Intel. Intel software guard extensions programming reference (rev2). Technical report, 2014.

[52] M. Shih, S. Lee, T. Kim and M. Peinado. T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs. In Proceedings of the 24nd Annual Network and Distributed System Security Symposium, NDSS 2017.

[53] W. Wang, G. Chen, X. Pan, Y. Zhang, X. Wang, V. Bindschaedler, H. Tang and C. Gunter. Leaky Cauldron on the Dark Land: Understanding Memory Side-Channel Hazards in SGX. In Proceedings of the 24nd ACM Conference on Computer and Communications Security, CCS 2017.

[54] A. Shahzad and A. Litchfield. Virtualization Technology: Cross-VM Cache Side Channel Attacks make it Vulnerable. Australasian Conference on Information Systems 2015.

[55] N. Zhang, K. Sun, D. Shands, W. Lou and Y. Hou. TruSpy: Cache Side-Channel Information Leakage from the Secure World on ARM Devices. Cryptology ePrint Archive 2016.

[56] A. Baumann, M. Peinado and G. Hunt. Shielding applications from an untrusted cloud with haven. ACM Transactions on Computer Systems, TOCS 2015.

[57] F. Zhang, J. Chen, H. Chen and B. Zhang. Cloudvisor: retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization. In Proceedings of Twenty-Third ACM Symposium on Operating System Principles 2011.

[58] A. Machiry, E. Gustafson, C. Spensky, C. Salls, N. Stephens, R. Wang, A. Bianchi,Y. Choe, C. Kruegel and G. Vigna. BOOMERANG: Exploiting the Semantic Gap in Trusted Execution Environments. In Pro- ceedings of the 24nd Annual Network and Distributed System Security Symposium, NDSS 2017.