软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2016,27(4):1042−1058 [doi: 10.13328/j.cnki.jos.005016]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

引入内可信基的应用程序保护方法*

邓 良[1,2], 曾庆凯[1,2]

[1](计算机软件新技术国家重点实验室(南京大学),江苏 南京 210023)

[2](南京大学 计算机科学与技术系,江苏 南京 210023)

通信作者: 曾庆凯, E-mail: zqk@nju.edu.cn

摘 要: 提出一种在不可信操作系统中保护应用程序的新方法 AppISO.针对传统的虚拟化方法的高开销问题,AppISO 在不可信操作系统的同一特权层引入内可信基,代替虚拟机监控器实施应用程序保护,避免了昂贵的特权层切换.同时利用硬件虚拟化,以及页表锁定、影子 IDT、切换页面等软件技术保证内可信基的安全性.证明了所提内可信基方法与虚拟化方法具有同样的高安全性.实验和分析结果表明,内可信基方法可显著地提高系统性能.

关键词: 内可信基;应用程序保护;不可信操作系统;虚拟化

中图法分类号: TP316

中文引用格式: 邓良,曾庆凯.引入内可信基的应用程序保护方法.软件学报,2016,27(4):1042−1058. http://www.jos.org.cn/1000-9825/5016.htm

英文引用格式: Deng L, Zeng QK. Inner TCB based application protection. Ruan Jian Xue Bao/Journal of Software, 2016,27(4):1042−1058 (in Chinese). http://www.jos.org.cn/1000-9825/5016.htm

Inner TCB Based Application Protection

DENG Liang[1,2], ZENG Qing-Kai[1,2]

[1](State Key Laboratory for Novel Software Technology (Nanjing University), Nanjing 210023, China)

[2](Department of Computer Science and Technology, Nanjing University, Nanjing 210023, China)

Abstract: This paper presents AppISO, a novel approach to provide whole-application protection in an untrusted operating system (OS). Unlike previous virtualization-based approach, AppISO does not directly use any higher privilege hypervisor for application protection, which is known to cause high overhead due to frequent privilege transitions. Instead, AppISO introduces a software component named Inner TCB running in the same privilege layer with the untrusted OS, and uses Inner TCB to realize application protection. Meanwhile AppISO leverages hardware virtualization and software techniques such as page table lockdown, shadow IDT, and transition page to guarantee the security and isolation of Inner TCB. This paper proves that Inner TCB can achieve the same level of security as hypervisor, and experimental results show that the presented approach has significant improvement in performance.

Key words: inner TCB; application protection; untrusted OS; virtualization

在现代操作系统中,安全敏感的应用程序很容易受到攻击.即使对应用程序本身进行安全设计和验证,底层的操作系统仍然存在广泛的攻击窗口,如大量的系统调用接口,硬件服务商提供的设备驱动等.操作系统结构复杂,通常采用不安全的程序语言编写.越来越多的安全漏洞报告表明,操作系统安全问题依然不容忽视.然而,在传统的系统中,由于操作系统运行在最高特权层,攻击者只要攻陷了操作系统,就能够控制并攻击应用程序,包

* 基金项目: 国家自然科学基金(61170070, 61572248, 61431008, 61321491); 国家科技支撑计划(2012BAK26B01); 南京大学优秀博士研究生创新能力提升计划 B(2015)

Foundation item: National Natural Science Foundation of China (61170070, 61572248, 61431008, 61321491); National Key Technology R&D Program of China (2012BAK26B01); the Program B for Outstanding PhD Candidate of Nanjing University (2015)

收稿时间: 2014-12-14; 修改时间: 2015-11-06; 采用时间: 2015-12-16; jos 在线出版时间: 2016-01-16

CNKI 网络优先出版: 2016-01-18 13:50:54, http://www.cnki.net/kcms/detail/11.2560.TP.20160118.1350.002.html

括窃取和篡改敏感信息、破坏控制流完整性等等.

针对操作系统不可信的问题,目前应用程序保护的主流方法是虚拟化方法.Inktag[1]和 Overshadow[2]在操作系统底层引入更高特权层的虚拟机监控器,对应用程序实现了全面保护,包括保护应用程序的内存隔离、控制流完整性和文件数据安全等.Trustvisor[3]基于虚拟机监控器,构建软件 TPM,保证应用程序中的代码片段(codeblock)的执行是安全隔离和可验证的.Fides[4]进一步保护了多个代码片段,保证它们之间的安全交互.Trusted Path[5]关注不可信操作系统中应用程序的I/O安全问题,基于虚拟化技术构建一条从应用程序到外设的可信I/O路径.此外,针对操作系统服务的不可信问题,Proxos[6]将安全敏感的系统调用导入到一个可信的VM中加以处理,以保证这些系统服务的可信.

虚拟化方法基于如下可信链:硬件>虚拟机监控器>安全策略.其中,所有的安全策略都基于虚拟机监控器实现;虚拟机监控器运行在更高的特权层,与不可信操作系统完全隔离,也有着更小的代码量和攻击窗口.引入更高特权层的虚拟机监控器保证了高安全性.然而,频繁的特权层切换也导致了较高的运行开销.

针对这一矛盾,本文提出了基于内可信基的应用程序保护方法AppISO.它在不可信操作系统的同一特权层引入软件模块,即内可信基.在应用程序保护中,内可信基接管了虚拟化方法中虚拟机监控器的所有工作,但是这些工作的执行无需特权层的切换.由于内可信基与不可信操作系统运行在同一特权层,本文提出了一系列技术保证内可信基的可靠执行和安全隔离.此外,本文提出了基于内可信基的文件保护和文件访问控制,避免了传统虚拟化方法中低效的加密解密计算.AppISO 具有以下特点:(1) 高安全性.应用程序保护基于可信链:硬件>内可信基>安全策略.所有的安全策略都基于内可信基实现.与虚拟机监控器一样,内可信基与不可信操作系统完全隔离.(2) 高性能.与传统虚拟化方法相比,内可信基的引入避免了频繁的特权层切换和低效的加密解密.本文进一步提出了一系列针对内可信基的性能优化技术.(3) 全面保护.与 Inktag 和 Oveshadow 一样,AppISO 提供全面的应用程序保护,包括内存保护、控制流完整性保护和文件访问控制等.

1 安全假设

本文假设操作系统内核是不可信的.不可信内核运行在最高特权层,能够任意访问底层硬件、执行内存中任意代码、读写内存中的任意数据以及利用外设进行恶意的 DMA 操作.在该假设下,不可信内核能够攻击系统中的任意一个应用程序,包括攻击它们的内存数据、控制流和文件 I/O 等.本文不考虑应用程序本身的漏洞,因为在现实中,需要保护的敏感应用程序通常代码量小且经过了充分的验证和测试,存在漏洞的几率比较小.此外,本文假设硬件(包括 CPU、内存控制器、外设)是可信的,不考虑物理攻击.以上安全假设与传统虚拟化方法 Inktag 和 Oveshadow 是一致的.

2 基本模型和安全目标

2.1 Inktag的基本模型和安全目标

为了与虚拟化方法对比,我们首先以 Inktag[1]为例,阐述虚拟化方法实现全面应用程序保护的基本模型和安全目标.如图1所示,该模型在操作系统底层引入了更高权限(root模式)的虚拟机监控器.应用程序运行在被虚拟机监控器保护的安全执行环境中,与不可信操作系统(与其他应用程序)隔离.基于该虚拟化模型,Inktag 实现了以下安全目标:

(1) 应用程序地址空间安全.Inktag 基于虚拟机监控器的内存保护机制,保护地址空间中数据和代码的安全隔离;并进一步在虚拟化模型的基础上提出 Paraverification 机制,保护应用程序的地址空间完整性.

(2) 应用程序控制流完整性.应用程序在运行过程中可能随时被操作系统中断.Inktag 基于虚拟机监控器,截获应用程序中的系统调用和中断异常,安全保存应用程序的执行上下文,并在返回应用程序之前恢复执行上下文,确保操作系统无法恶意修改应用程序的控制流和寄存器.

(3) 应用程序文件 I/O 安全和访问控制.由于应用程序仍然需要依赖操作系统提供的文件服务实现文件存

储,Inktag 基于加密和哈希,保证文件数据在操作系统中的私密性和完整性;并进一步提出基于虚拟机监控器的文件访问控制,保证文件访问的安全性.

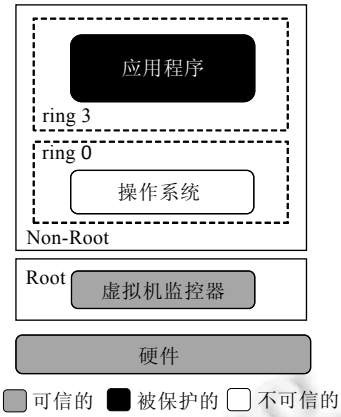


Fig.1 Overview of Inktag

图 1 Inktag 的基本模型

然而,在该模型中,如下两个因素造成了较高的性能开销.

(1) 频繁的 non-root/root 切换.虚拟机监控器必须截获应用程序中所有的页表更新操作,实现地址空间完整性保护;虚拟机监控器必须截获应用程序中所有的系统调用和中断异常.例如,一次系统调用需要 4 次昂贵的 non-root/root 切换.

(2) 低效的加密解密.虚拟机监控器对文件数据的加密解密、密钥的管理也带来了较大的开销.

2.2 AppISO的基本模型、安全目标和主要挑战

本文中,AppISO 的安全目标与 Inktag 一致,包括保护应用程序的地址空间安全、控制流完整性、文件 I/O 安全和实现访问控制.然而,针对虚拟化模型的高开销问题,AppISO 提出了内可信基模型,以实现这些安全目标.

如图 2 所示,在该模型中,整个系统始终运行在 non-root 模式中.该 non-root 模式的执行环境在系统启动时被设置,此后,在系统运行过程中不再陷入到 root 模式中.在 non-root 模式中,操作系统与应用程序运行在两个不同的地址空间,分别称为内核地址空间 KAS(kernel address space)和安全地址空间 SAS(secure address space).操作系统运行在 KAS 的 ring 0 层中;应用程序运行在 SAS 的 ring 3 层中.与 Inktag 一样,本文中的操作系统指的是操作系统内核.KAS 中使用的页表称为 KPT(kernel page table),SAS 中使用的页表称为 SPT(secure page table).

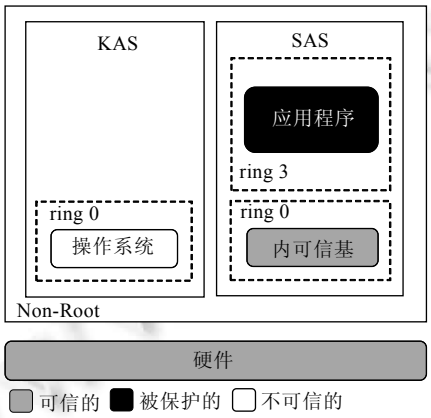


Fig.2 Overview of AppISO

图 2 AppISO 的基本模型

该模型在 SAS 的 ring 0 层引入内可信基,应用程序运行在内可信基保护的安全上下文中.当 SAS ring 3 层

的应用程序与操作系统交互时,应用程序首先陷入到 SAS ring 0 层的内可信基,内可信基保存应用程序上下文,然后切换到KAS,进入操作系统执行.当操作系统返回应用程序时,必须首先返回SAS ring 0层的内可信基,内可信基恢复应用程序上下文,再返回 SAS ring 3 层中的应用程序继续执行.

在该模型中,应用程序的保护由 non-root 模式中的内可信基实现,无需陷入到 root 模式中.

(1) 基于内可信基的内存保护机制,保证应用程序地址空间与不可信操作系统的安全隔离和地址空间的完整性.

(2) 基于内可信基的影子 IDT 机制,截获系统调用和中断异常,保护应用程序的控制流完整性和寄存器安全.

(3) 基于内可信基的 I/O 验证机制,在不可信操作系统内构建可信文件数据流,保证应用程序文件数据的私密性和完整性,同时避免了低效的加密解密;并进一步提出基于内可信基的文件访问控制.

该模型能够有效地解决虚拟化方法影响性能的两个主要问题.然而,该模型的挑战在于:内可信基和不可信操作系统运行在同一特权层(non-root 模式的 ring 0 层),那么,如何在同一特权层实现内存保护、控制流完整性保护和文件保护呢?更为重要的是,如何保证内可信基本身的安全?

本文第 3 节介绍技术背景.第 4 节介绍内可信基的内存保护机制、影子 IDT 机制.第 5 节介绍应用程序地址空间保护和控制流保护.第 6 节介绍内可信基的 I/O 验证机制和文件访问控制.第 7 节构建内可信基的安全模型,分析和证明内可信基安全隔离和可信执行必须满足的安全属性.第 8 节介绍 AppISO 在 Linux 上的原型系统实现.第 9 节给出实验结果并进行性能分析.第 10 节介绍相关工作.第 11 节对本文进行总结.

3 技术背景

为了更加方便地理解 AppISO 的设计和实现原理,我们首先对 AppISO 用到的硬件虚拟化技术作一些简单的介绍.在 x86 中,硬件虚拟化(Intel VMX 和 AMD SVM)提供了两种 CPU 运行模式:root 模式和 non-root 模式.运行在 root 模式中的软件可通过使用一个位于内存中的数据结构 VMCS(virtual machine control structure)来控制 non-root 模式中软件的行为,包括限制 non-root 模式中软件能够执行的特权操作以及能够直接处理的系统事件.当 non-root 模式软件执行某个不允许的特权操作时,CPU 会产生 VM exit 并陷入到 root 模式中.从而,root 模式中软件能够对这些操作进行截获和处理.

在 AppISO 中,我们将整个系统(包括内核、内可信基和应用程序)始终运行在 non-root 模式中.通过系统初始化时 VMCS 的设置,AppISO 允许 non-root 模式中软件执行大部分的特权操作和直接处理系统事件,保证整个系统在 non-root 模式中的顺利运行.同时,AppISO 基于硬件虚拟化,对 non-root 模式中几个特殊的特权操作(比如修改 cr3 寄存器)进行限制,在此基础上,实现了内可信基的隔离和应用程序的保护.

4 内可信基的内存保护机制

在虚拟化方法中,虚拟机监控器实施保护的关键是,截获并验证系统中所有的 CR3 寄存器修改和页表更新操作,以完全控制页表,实现内存保护.

在 AppISO 中,内可信基利用硬件虚拟化直接对 CR3 寄存器的目标值进行限制.同时,内可信基截获并验证系统中所有的页表更新操作,实现内存保护.

4.1 CR3限制

在系统启动、设置 non-root 执行环境时,AppISO 使用硬件虚拟化的 CR3-Target Controls[7]机制保证整个 non-root 模式系统(操作系统、应用程序和内可信基)只能运行在 KAS 和 SAS 中.CR3-Target Controls 机制允许在 VMCS 的 CR3_TARGET_LIST 中预设定最多 4 个 CR3 寄存器的目标值.当运行在 non-root 模式中的软件修改 CR3 寄存器时,若目标值是 CR3_TARGET_LIST 中的一个,则该 CR3 修改可直接在 non-root 模式中完成.AppISO 仅使用两个目标值,将 KAS 的页表 KPT 和 SAS 的页表 SPT 的基地址写入 CR3_TARGET_LIST,因此,

整个系统在运行过程中只能将CR3寄存器修改为这两个值.如果试图修改为其他目标值,则均会使CPU陷入到root模式,AppISO中任何导致陷入到root模式的操作都被认为是恶意的,并会导致整个系统重启.因此,基于硬件虚拟化的设置,整个系统只能在KAS和SAS中运行.

4.2 页表验证

在KAS中,AppISO基于"页表锁定"技术,保证运行在KAS中的任何软件(即使运行在ring 0层)都不能修改页表.具体来说,在KAS中,整个页表被映射为只读.任何运行在KAS中的软件若想要修改某个页表项,则必须首先将该页表项映射为可写.然而,整个页表都是只读的,不允许将任何只读映射修改为可写.因此,在KAS中,整个页表都被只读锁定了.KAS中的软件试图解除页表锁定的方式只有两个:(1) 修改CR3寄存器、切换地址空间.但是该方式已被硬件虚拟化限制.(2) 修改CR0寄存器的WP位,使页表的只读保护失效.针对这种方式,AppISO使用硬件虚拟化禁止non-root模式的所有软件修改CR0寄存器的WP位,任何修改WP位的操作都会陷入到root模式,进而导致系统重启.

在SAS中,AppISO基于传统特权层隔离(ring 3/ring 0)的方式,保证只有ring 0层的软件能够修改页表.页表在其他层(ring 3层)被映射为不可见.

因此,整个系统中只有SAS ring 0层的软件能够修改页表.AppISO将内可信基运行在SAS ring 0层,将操作系统运行在KAS ring 0层,应用程序运行在SAS ring 3层.因而所有的页表更新操作只能由内可信基完成.当操作系统需要更新页表时,只能向内可信基发出请求,内可信基能够截获并验证所有的页表更新操作,实现内存保护.这种页表验证方式与半虚拟化方法类似.

需要强调的是,虽然KAS ring 0层和SAS ring 0层都是系统中最高特权层,但AppISO基于CR3限制和页表锁定技术,消除了KAS ring 0层的操作系统修改页表的权限.

4.3 内可信基的隔离与保护

由于只有SAS ring 0层的软件能够修改页表,因此,实现内存保护的关键是:保证SAS ring 0层中只有内可信基运行,外部组件(操作系统和应用程序)不能破坏SAS ring 0层中内可信基的数据代码和执行流的完整性.

我们分析一下虚拟化方法的可信基(虚拟机监控器)的保护原理:虚拟机监控器控制了所有进入root模式的入口,保证CPU一旦切换到root模式,虚拟机监控器将获得系统的控制权.虚拟机监控器在root模式中完成自己的执行流程,在返回外部组件时,将CPU切换回non-root模式,保证root模式中只有虚拟机监控器运行.因此,外部组件始终只能运行在non-root模式,不能破坏root模式中虚拟机监控器的数据代码和执行流完整性.

参照虚拟化方法,内可信基也必须控制所有进入SAS ring 0层的入口点,保证一旦CPU进入SAS ring 0层,内可信基便获得系统控制权.内可信基在SAS ring 0层完成自己的执行流程,在返回外部组件时,将CPU切换回KAS或者SAS ring 3,保证SAS ring 0层中只有内可信基运行.因此,外部组件始终只能运行在KAS或者SAS ring 3层中,不能破坏SAS ring 0层中内可信基的数据代码和执行流的完整性.

4.3.1 入口点控制

硬件虚拟化规定了只允许系统在KAS和SAS中运行.因此,SAS ring 0层的入口点包括两类:(1) 从SAS ring 3层通过中断异常陷入,进入SAS ring 0层(从应用程序进入内可信基);(2) 从KAS切换到SAS,进入SAS ring 0层(从操作系统进入内可信基).

第1类入口点控制.在x86中,中断异常的入口点在IDT(interrupt descriptor table)中设置,IDT的基地址由IDTR(interrupt descriptor table register)寄存器指定.当中断异常发生时,CPU根据IDTR寄存器中的地址找到IDT,并根据中断向量(interrupt vector)索引IDT,跳转到相应IDT表项指定的中断处理程序.

在AppISO,操作系统仍然维护自己的IDT,但是CPU实际使用的是内可信基维护的IDT(称为影子IDT),IDTR寄存器指向影子IDT.同时,AppISO使用以下技术保证影子IDT的安全:(1) 利用硬件虚拟化禁止non-root模式的所有软件执行LIDT指令、修改IDTR寄存器,因此IDTR寄存器只能指向影子IDT的基地址.(2) 将影子IDT所在的页框、对应的中断处理程序的代码均映射为只读,保证它们无法被操作系统修改.(3) 由

于 IDTR 和 IDT 表项均使用虚拟地址,因此,它们对应的页表映射也被保护.(4) 内可信基在影子 IDT 的表项中使用中断门(interrupt gate[7]),保证在执行影子 IDT 的中断处理程序时,中断被禁止,控制流无法被操作系统劫持.因此,当 SAS ring 3 层发生中断异常时,内可信基通过影子 IDT 在 SAS ring 0 层可靠地获得系统控制权.

此外,x86 中可调用 sysenter 和 syscall 两条指令(快速系统调用)陷入 SAS ring 0 层.sysenter 和 syscall 使用 MSR 寄存器(比如 msr_sysenter_eip)来指定入口点.AppISO 将这些 MSR 寄存器指向内可信基的入口点,并使用硬件虚拟化的 MSR Controls 机制[7]禁止在系统初始化后修改该 MSR 寄存器.

第 2 类入口点控制.这类入口点的控制更加困难,KAS 中的不可信操作系统可能执行自己代码中任意的 MOV-TO-CR3 指令(x86 中修改 CR3 寄存器的指令)、修改 CR3 寄存器值,试图切换到 SAS.AppISO 必须保证,一旦操作系统试图从 KAS 切换到 SAS,内可信基将可靠地获得系统的控制权.

AppISO 在 KAS 和 SAS 中引入切换页面,并保证该页面是 KAS 和 SAS 中被同时映射为可执行的唯一页面.KAS 中的操作系统只能使用切换页面中的 MOV-TO-CR3 指令切换到 SAS,而执行其他页面中的 MOV-TO-CR3 指令试图切换地址空间均会导致缺页异常,进而被内可信基的影子 IDT 截获并禁止,因为其他页面被禁止在 KAS 和 SAS 中同时执行.因此,切换页面是从 KAS(操作系统)切换到 SAS(内可信基)的唯一方式.

图 3 描述了作为操作系统进入内可信基唯一入口的切换页面中的指令序列.首先,KAS 中的操作系统跳转到切换页面中的 P1,P1 将 SPT 的页表基地址赋值给 RAX 寄存器,P2 再将之赋予 CR3 寄存器,切换到 SAS.即使不可信操作系统试图绕过 P1,直接执行 P2 中的 MOV-TO-CR3 指令,它也只能将 CR3 寄存器的值修改为 SPT 的页表基地址,因为其他的目标值都已被硬件虚拟化设置禁止.进入 SAS 后,P3 执行 CLI 指令,立即禁止中断,保证从此以后的指令执行都是原子的,操作系统无法再获得控制流.然而,不可信操作系统仍然可能在 P2 和 P3 触发恶意中断.针对这一问题,内可信基通过影子 IDT 检测并禁止该恶意中断.最后,P4 和 P5 跳转到内可信基的入口点,进入内可信基执行.

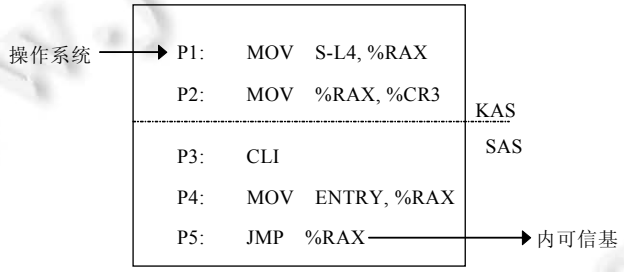


Fig.3 Transition page

图 3 切换页面

切换页面的入口机制保证了,KAS 中操作系统只能执行切换页面中的 MOV-TO-CR3 切换到 SAS、进入内可信基;在切换页面中,执行流一旦进入到 SAS ring 0 层,将原子性地进入内可信基中指令执行,保证内可信基获得控制权.

4.3.2 内可信基完整性保护

从入口点进入 SAS ring 0 后,内可信基在整个执行过程中,中断被禁止,执行流无法被外部组件劫持:在第 1 类入口,中断被影子 IDT 中的中断门禁止;在第 2 类入口,中断被切换页面中的 CLI 指令禁止.只有当返回外部组件时,内可信基才恢复中断.同时,内可信基将 CPU 切换回 KAS 或者 SAS ring 3,保证 SAS ring 0 中只有内可信基运行.当内可信基中发生不可屏蔽中断(NMI)时,内可信基通过影子 IDT 暂时阻塞该 NMI,防止外部组件使用 NMI 劫持内可信基执行流.当返回到操作系统时,内可信基再将该 NMI 转发给操作系统进行处理.因此,内可信基的执行流完整性得到保护.

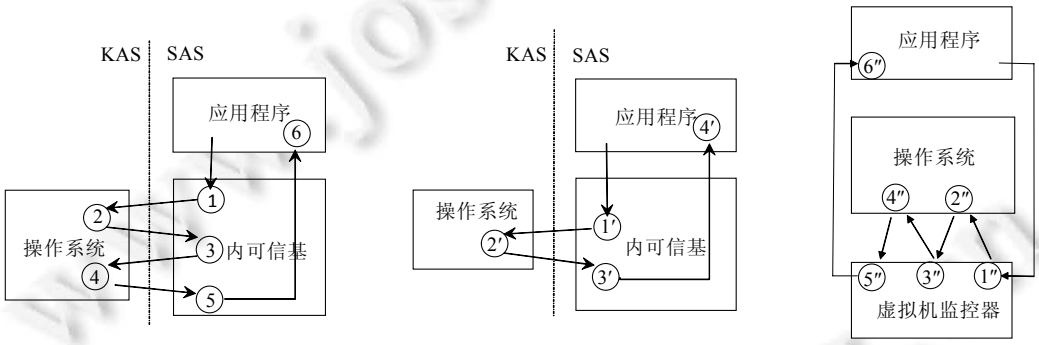
运行在 KAS 的操作系统或者 SAS ring 3 层的应用程序无法破坏内可信基数据代码的完整性(它们在 KAS 和 SAS ring 3 层被映射为不可见),也无法修改页表、对内可信基的数据代码进行恶意映射.

4.4 应用程序页表切换

每个应用程序都有自己的页表,进程切换时应用程序页表也需要作相应切换.然而,硬件虚拟化设置不允许修改CR3寄存器切换页表.AppISO提出“软切换”技术,实现SAS中不同应用程序页表之间的切换.在x86-64中,页表由4级结构组成(L1,L2,L3和L4表示),CR3寄存器指向L4页表的物理基地址.在传统操作系统中,页表的切换是通过修改CR3寄存器来实现的,本文称其为“硬切换”.而在软切换中,CR3寄存器保持不变,它指向SPT的L4页表(称为S-L4),页表切换是通过将目标应用程序的L4页表复制到S-L4来实现的.由于x86中L4页表的大小只有一个页面的大小(4KB),而且只需要拷贝应用程序部分,软切换并不会带来太大的性能开销.而且,与其他页表一样,S-L4只能被SAS ring 0层中的内可信基修改.因此,进程切换时,操作系统只能向内可信基发出请求,由内可信基切换应用程序页表.内可信基完成软切换并对目标应用程序页表进行验证,本质上与内可信基对页表更新的验证没有区别.

4.5 页表更新验证和性能优化

图4(a)阐述了内可信基的页表更新验证过程.当应用程序发生缺页中断、陷入内可信基时,在步骤1,内可信基保存应用程序安全上下文,并将该缺页中断转发给KAS中的操作系统.在步骤2,操作系统完成缺页中断处理,分配相应的物理页框.由于KAS中的操作系统无法直接更新页表映射新页框(页表被锁定),它只能切换到SAS,由切换页面进入内可信基,请求内可信基完成页表更新.在步骤3,内可信基根据请求完成页表更新和验证操作,然后返回操作系统.在步骤4,操作系统完成余下的缺页中断处理工作,最终返回内可信基.在步骤5,内可信基恢复应用程序安全上下文,返回应用程序.



(a) 基于内可信基(优化前) (b) 基于内可信基(优化后) (c) 基于虚拟化

Fig.4 The process of page table update verification

图4 页表更新验证过程

然而,整个过程额外增加了两次KAS/SAS的切换,造成了额外的性能开销.针对这一问题,我们提出了称为“延迟更新”的优化.事实上,该缺页中断处理的是SPT的更新(应用程序页表的更新),而应用程序页表的更新一直要到应用程序运行时才会发生作用.也就是说,即使在步骤3中,内可信基马上更新了SPT,该更新也要到步骤6才会用于映射内存.因此,AppISO推迟了该页表的更新操作.优化后的处理过程如图4(b)所示,在步骤1′,内可信基向操作系统发送缺页中断.在步骤2′,操作系统完成物理页面分配后,并不马上请求内可信基更新页表,而是将页表更新的具体内容置于一块共享内存中.一直到操作系统完成所有处理,返回内可信基时(步骤3′),内可信基才从共享内存中读取更新请求,完成页表更新,然后直接返回应用程序(步骤4′).由此可见,优化后的缺页中断处理无需额外的地址空间切换.如果缺页中断发生在KAS中(即KPT缺页中断),页表更新请求无法延迟.然而,KPT缺页中断很少发生,特别是在64位操作系统中.此外,AppISO进一步利用x86的硬件特性(PCID[7])给KAS和SAS中的页表项做上不同的硬件标记(tag).因此,在KAS和SAS的切换过程中,无需刷新TLB缓存,极大地提高了性能.

图4(c)描述了基于虚拟化系统(Inktag)的页表验证过程.其中,离开和返回应用程序必须进入虚拟机监控器,更新页表也必须进入虚拟机监控器,导致了6次non-root/root的切换.而在内可信基的页表更新验证过程,无需

non-root/root 的切换.

5 应用程序地址空间和控制流完整性保护

SAS ring 0 层中的内可信基基于页表验证,完全控制了页表,因而可以通过页表映射对系统中任意一块内存进行保护.下面具体讨论内可信基如何保护应用程序地址空间中的内存,实现地址空间隔离和完整性保护.

5.1 地址空间隔离

为了实现应用程序地址空间与操作系统的隔离,内可信基使用一个数组来跟踪系统中每个物理页框的映射情况.如表 1 所示,每个页框可被定义为 3 种映射状态:normal,isolated 和 owned.Normal 状态的页框被 KPT 和 SPT-内(本文将 SPT 中映射内可信基的部分称为 SPT-内;映射应用程序的部分称为 SPT-应)两者映射,因而操作系统和内可信基都能访问它们.Isolated 状态的页框仅被 SPT-内映射,只能被内可信基访问.内可信基和页表所在的页框都被映射为 isolated 状态.Owned 状态的页框已被分配给某个应用程序,仅被 SPT-内及其应用程序拥有者的 SPT-应映射,因而只有内可信基及其拥有者能访问它们.内可信基使用唯一的安全标识符(SID)来标识 owned 页框的拥有者.SID 在进程创建时由内可信基赋予每个进程,每个进程都有自己单独的 SID.

Table 1 Mapping states of memory frames (M means mapping, N means not mapping)

表 1 页表映射状态(M 表示映射,N 标识不映射)

	SPT-内	KPT	SPT-应
Normal	M	M	N
Isolated	M	N	N
Owned	M	N	仅被拥有者映射

以应用程序 A 的页框分配为例,阐述页框映射状态的转化过程和地址空间隔离.当一个页框被操作系统分配给应用程序 A 时,内可信基要求操作系统只能使用 isolated 状态的页框.当内可信基将该页框映射到 A 的 SPT-应时,会对该页框的映射状态进行验证,拒绝所有非 isolated 页框.然后,该页框被安全地分配给 A,标记为 owned 状态、并被 A 的 SID 标记.内可信基通过验证 SID,禁止 owned 状态的页框被 KPT 或者其他应用程序的 SPT-应映射,保证 A 的内存隔离.同时,为了防范 Iago 攻击[8],内可信基也禁止应用程序页框在应用程序地址空间中被重映射.

5.2 地址空间完整性保护

除开地址空间隔离,内可信基必须保证应用程序地址空间中映射的内存是应用程序想要的(地址空间完整性).例如,对于文件映射,内可信基需要保证:(1) 应用程序访问的文件是它想要的文件;(2) 访问的文件位置是它想要的位置.

AppISO 采用了 Inktag 提出的 Paraverification 机制解决该问题.内可信基要求应用程序自己维护一个链表,描述自己的地址空间映射状态.该链表与应用程序地址空间的其他数据一样,无法被操作系统修改.当内可信基更新应用程序页表时,将会查看该链表.如果操作系统的页表更新请求与该数组描述的地址空间映射状态不一致,则内可信基将拒绝该请求.下面以应用程序调用 mmap 系统调用映射文件为例,阐述这一机制.

在磁盘上,不同文件被不同的 ID 标记,同一文件的不同数据块被文件偏移标记,表示该数据块在文件中的位置.当应用程序调用 mmap 系统调用时,它同时更新自己维护的地址空间状态链表,在该数组中描述自己的意图,包括想要读取的文件对应的 ID 和偏移.当内可信基进行页表更新验证时,会查看该链表.如果该更新对应的地址空间映射到文件,则内可信基将该链表中的 ID 和偏移与文件本身的 ID 和偏移进行比对.如果不是应用程序想要的文件或者偏移位置,内可信基将会拒绝将文件映射到地址空间、并告知应用程序.

在应用程序地址空间保护中,本文的主要贡献是提出了内可信基的内存保护机制.而在内存保护机制上如何具体实现地址空间完整性保护与 Inktag 类似(比如 Paraverification 机制),具体细节本文不再重复.

5.3 控制流完整性保护

如第 4.3.1 节所述,内可信基通过影子 IDT 技术和硬件虚拟化截获系统中所有的中断异常和系统调用.当 SAS ring 3 层中应用程序的控制流被中断异常或者系统调用中断时,SAS ring 0 层的内可信基获得系统的控制权,保存应用程序的安全上下文,然后进入操作系统执行.当操作系统返回应用程序时,操作系统只能使用切换页面(唯一方式)进入 SAS.同时,SAS ring 0 层的内可信基获得系统的控制权,并恢复应用程序的安全上下文,最后进入应用程序执行.因此,即使操作系统能够任意中断应用程序的执行,也无法破坏应用程序的控制流完整性或窃取上下文中的敏感数据.

6 内可信基的 I/O 验证机制和应用程序文件保护

6.1 内可信基的I/O验证机制

内可信基截获并验证系统中所有发送到磁盘外设的 I/O 命令,整个 I/O 验证过程在 non-root 模式中实现.

在 x86 中,软件通过两种方式向外设发送 I/O 命令:端口 I/O(port I/O)和内存映射 I/O(memory mapped I/O).端口 I/O 使用特殊的 I/O 指令(比如 in 和 out),而内存映射 I/O 将一段系统内存(称为 I/O 内存)分配给外设,使用一般的访存指令来控制外设.软件可访问 PCI/PCIe 配置空间,修改分配给设备的 I/O 端口和 I/O 内存的基地址.

现代磁盘管理器(比如 SATA[9])均使用内存映射 I/O,因而 AppISO 仅需要截获内存映射 I/O,就可以截获发送到磁盘外设的I/O命令.具体来说,基于内可信基的内存保护机制,内可信基将分配给磁盘管理器的I/O内存映射为只读.当操作系统需要向磁盘发送 I/O 命令时,它只能将该命令转发给内可信基,由内可信基访问 I/O 内存,完成 I/O 操作.因此,内可信基能够截获这些 I/O 命令,并进行必要的 I/O 验证.然而,不可信操作系统仍然能够通过访问 PCI/PCIe 配置空间,恶意修改磁盘管理器的 I/O 内存基地址,从而绕过内可信基的只读保护.针对该问题,AppISO 在系统启动时,对 BIOS 设置的 PCI/PCIe 配置空间进行验证,在系统运行过程中禁止操作系统访问整个 PCI/PCIe 配置空间.在 x86 中,PCI/PCIe 配置空间可以通过两种方式访问:(1) 通过特殊的 I/O 端口访问;(2) 通过保留的系统内存访问.对于第 1 种方式,AppISO 在系统启动时,利用硬件虚拟化的 I/O bitmap 机制[7],禁止 non-root 模式中的软件(包括操作系统)访问该段 I/O 端口;对于第 2 种方式,内可信基将这段保留的系统内存在地址空间中映射为不可见,禁止操作系统访问.

6.2 文件数据保护和访问控制

传统虚拟化方法(Inktag 和 overshadow)均使用加密和哈希技术保护应用程序文件数据的安全.在文件数据进入到操作系统之前,虚拟机监控器对文件数据进行加密,防止数据被窃取;当应用程序从操作系统获取文件数据时,虚拟机监控器比对文件数据的哈希值,以防止数据被篡改.

在 AppISO 中,应用程序的文件数据以明文的形式进入操作系统,内可信基通过在不可信操作系统中构建一条可信文件数据流来传输文件数据,防止文件数据被窃取或篡改.同时,类似于 Inktag 中基于虚拟机监控器的访问控制模型,AppISO 实现了基于内可信基的访问控制模型.在该模型中,文件数据被 SUID(secure user identifier)标识(类似于 Inktag 的 OID).每个用户都拥有自己的 SUID,用户在启动自己的应用程序时,将 SUID 赋予应用程序.内可信基通过验证 SUID 保证文件数据只能被其所有者访问.AppISO 也向用户提供了灵活的文件访问控制,比如 SUID 可以标记一个用户组(group),用于用户之间的文件共享;SUID 可以指定不同的读写和执行权限等.

6.2.1 可信文件数据流和文件访问控制

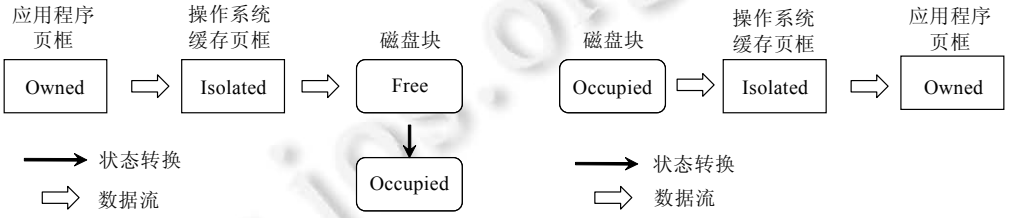
图 5(a)描述了应用程序调用 write 系统调用存储文件时,文件数据在操作系统中的流动情况.其中,文件数据从应用程序地址空间的页框被拷贝到操作系统的缓存页框,然后从缓存页框传输到磁盘块(disk block).可信文件数据流能够保证:文件数据在操作系统中传输时,文件数据与操作系统隔离,无法被操作系统访问.

(1) 从应用程序页框到操作系统缓存页框的数据流隔离.首先,应用程序页框被映射为 owned 状态,其中的文件数据无法被操作系统访问.当操作系统需要将文件数据从应用程序页框复制到缓存页框时,只能向内可信

基发出请求,由内可信基完成数据拷贝.内可信基要求操作系统只能提供 isolated 状态的缓存页框,对于非 isolated 状态的缓存页框,内可信基将拒绝拷贝文件数据.因此,在拷贝后,操作系统仍然无法访问缓存页框上的文件数据.同时,该缓存页框被打上应用程序的 SUID,标识缓存页框上文件数据的所有者.

(2) 从缓存页框到磁盘块的数据流隔离.内可信基定义了两种磁盘块的状态:free 和 occupied,并使用一个数组(称为磁盘块数组)进行状态跟踪.在磁盘块数组中,内存的一个位对应一个磁盘块的状态.内可信基基于 I/O 验证机制,对所有发送到磁盘的 I/O 命令进行验证.如果该命令是将文件数据写入磁盘块,内可信基确保被写入的磁盘块只能是 free 状态.在数据传输完成后,该磁盘块变为 occupied 状态,SUID 和文件数据一起存储在磁盘块上,用于标识该磁盘块的所有者.此后,基于 I/O 验证机制,内可信基只允许具有相同 SUID 的内存页框与该 occupied 磁盘块进行数据传输,因而磁盘块上文件数据隔离和访问控制得到保证.

当应用程序调用 read 系统调用读取文件时,可信文件数据流的实现方法类似.如图 5(b)所示,基于 I/O 验证机制,内可信基只允许 occupied 磁盘块上的文件数据传输到 isolated 状态的缓存页框,同时,该缓存页面被赋予 occupied 磁盘块的 SUID.此后,内可信基只允许该缓存页框的数据拷贝到具有相同 SUID 的应用程序页框,被其应用程序所有者访问.此外,第 5.2 节中验证地址空间完整性相关的元数据(文件 ID 和偏移)也随文件数据一起,在可信文件数据流中传输,与不可信操作系统隔离.



(a) 调用 write 时的可信文件数据流 (b) 调用 read 时的可信文件数据流

Fig.5 Trusted file data flow

图 5 可信文件数据流

6.2.2 性能优化

在构建可信文件数据流中,操作系统需要向内可信基发出请求,由内可信基完成应用程序页框和操作系统缓存页框之间的数据拷贝,造成了额外的 KAS/SAS 的切换.针对这一问题,AppISO 基于现代操作系统的磁盘缓存特性提出了以下性能优化.事实上,在 read 系统调用的文件数据流中(如图 5(b)所示),即使内可信基立即将文件数据从操作系统缓存页框复制到应用程序页框,该文件数据也要等到返回应用程序之后才会被使用.因此,AppISO 将该数据复制延迟到操作系统最终返回内可信基时,以避免额外的地址空间切换.该思想与第 4.5 节中针对页表验证的优化类似.同样,该延迟思想对于 write 系统调用中的文件数据流也适用,因为在现代操作系统中磁盘 I/O 的传输总是被延迟的.在图 5(a)中,即使内可信基将文件数据立即从应用程序页框拷贝到操作系统缓存页框,该缓存页框中的文件数据也不会马上被传输到磁盘块中.因此,AppISO 将该数据拷贝延迟.操作系统只需将缓存页框的地址记录下来,发送到一块共享内存中;当操作系统最终返回内可信基时,内可信基读取该记录,完成数据复制,并代替操作系统将该缓存页框标记为脏页(dirty page).

需要强调的是,虽然不可信操作系统参与到性能优化过程中,但可信文件数据流中文件数据的安全隔离仅依赖于内可信基.

6.2.3 SUID 身份验证问题

内可信基的用户验证与 Linux 中用户验证过程类似,是基于验证用户密码来实现的.内可信基进一步使用公钥/私钥对保证不可信操作系统无法窃取用户密码.

AppISO 假设 CPU 的 TPM[10,11]硬件模块可用,并使用 TPM 中的存储密钥(storage key)来加密和解密内可信基的私钥.当用户启动自己的应用程序时,将自己的用户密码放置在应用程序的可执行文件里,并使用内可信基的公钥进行加密.内可信基从应用程序可执行文件中获得用户密码,并使用自己的私钥进行解密.然后,内可信

基通过验证用户密码赋予该应用程序相应的 SUID,标识应用程序的身份.在整个身份验证过程中,AppISO 保证了可信链:TPM 储存密钥>内可信基私钥>用户密码>SUID.

AppISO 虽然在 SUID 身份验证时使用了加密解密,但是此过程只在应用程序启动时发生,并不会影响应用程序的文件性能.

7 内可信基的安全性

为了实施应用程序保护,内可信基必须具有与虚拟机监控器一样的安全性.本节进一步归纳内可信基必须满足的安全属性,并从攻击的角度对这些安全属性进行证明.

定理 1. 在内核不可信的安全假设下,内可信基满足如下安全属性:

(1) 数据代码完整性.内可信基的数据代码不能被外部不可信组件(操作系统和应用程序)读写、执行.

(2) 入口点完整性.外部组件(操作系统和应用程序)只能从指定入口点进入内可信基.

(3) 执行流完整性.进入内可信基后,按照内可信基预先设定的方式运行,执行流不会被外部组件修改.

证明:在系统启动后,系统将处于操作系统初始化、执行应用程序、应用程序请求操作系统服务、操作系统服务以及进程切换等几种工作情况.以下逐一证明各种情况下,内可信基这些安全属性都能得到满足.

(1) 系统启动:系统启动时,首先进入内可信基执行.内可信基初始化整个系统的安全环境,以保证系统在此后运行过程中内可信基的所有安全属性得到满足.具体来说,内可信基一开始运行在 root 模式,它利用硬件虚拟化设置 non-root 模式执行环境,并初始化 non-root 模式中 KAS 和 SAS,然而将整个系统陷入到 non-root 模式中执行.在 non-root 模式中,内可信基运行在 SAS ring 0 层,并通过切换页面和影子 IDT 技术控制了所有进入 SAS ring 0 层的入口点.由于此时不可信组件(操作系统和应用程序)还没有运行,并且 AppISO 通过可信启动硬件保证内可信基映像的完整性,因此内可信基设置安全环境的整个过程是可信的.最后,内可信基启动操作系统在 KAS ring 0 层中执行.

(2) 操作系统初始化:操作系统在 KAS ring 0 层完成自己的初始化工作.操作系统无法读写、执行内可信基的数据代码(它们在 KAS 中映射为不可见);操作系统也无法修改页表映射,因为 KAS 中页表被只读锁定.硬件虚拟化只允许操作系统从 KAS 切换到 SAS.然而,一旦操作系统试图切换到 SAS,内可信基将通过切换页面(唯一方式)获得系统的控制权.对于正常的切换(比如,操作系统请求内可信基更新页表),内可信基完成自己的服务(页表更新和验证),并在返回操作系统时,将 CPU 切换回 KAS,保证操作系统始终只能在 KAS 中运行;对于异常切换,内可信基将直接重启系统.由于操作系统始终只能运行在 KAS 中,它无法修改页表映射、破坏内可信基对 SAS ring 0 层入口点的控制,也无法修改只读的切换页面,破坏内可信基的入口点完整性(入口点由切换页面中的 JMP 指令指定).此外,由于操作系统进入内可信基时,中断立即被切换页面中的 CLI 指令禁止,操作系统也无法利用中断破坏内可信基的执行流完整性.因此,进入内可信基后,内可信基在自己的数据代码上(数据代码完整性已保证),按照其预先设定的方式运行.

(3) 执行应用程序:当操作系统离开 KAS、执行自己的应用程序时,只能从切换页面进入内可信基.内可信基将应用程序运行在 SAS 的 ring 3 层,通过传统特权层隔离的方法(ring 3/ring 0)保证应用程序无法读写、执行内可信基的数据代码,或者破坏内可信基的入口点完整性和执行流的完整性.

(4) 应用程序请求操作系统服务:内可信基通过影子 IDT 技术控制了所有从 SAS ring 3 层到 SAS ring 0 层的入口点,保证 CPU 一旦从 SAS ring 3 层陷入到 SAS ring 0 层(应用程序发出系统调用或者发生中断异常),内可信基将获得系统的控制权.内可信基将相关系统事件转发给操作系统之前,将地址空间切换到 KAS,保证操作系统只能在 KAS 中执行.

(5) 操作系统服务:由于内可信基在进入操作系统时,将 CPU 切换到 KAS,如上面(2)所述,因此,在操作系统服务过程中,也无法破坏内可信基的安全.

(6) 进程切换:当操作系统进行进程切换时,由于它无法切换 SAS 中的应用程序页表(硬切换和软切换均不能使用),只能向内可信基发出请求.内可信基在 SAS ring 0 层对应用程序页表实现软切换,并对目标应用程序页

表进行验证.此后,内可信基将在 SAS ring 3 层执行新切换的应用程序.如上面的 3)所述,新切换的应用程序也无法破坏内可信基的安全. □

8 原型系统实现

我们在 Linux 操作系统上实现了 AppISO 的系统原型.

8.1 系统启动

系统启动时,BIOS 首先进入内可信基执行.AppISO 依赖于 x86 的可信启动硬件(TPM)保证内可信基的可信启动.内可信基如表 2 所示设置 VMCS(具体过程在前文中已论述),配置 non-root 模式的执行环境,然后将整个系统陷入到 non-root 模式,并启动操作系统在 KAS ring 0 层执行,该启动过程使用了 Linux 的半虚拟化接口.此后,内可信基和操作系统始终运行在 non-root 模式中,不再陷入 root 模式.

Table 2 VMCS configurations in system startup

表 2 系统启动过程中 VMCS 设置

VMCS 设置	作用
Descriptor-Table exiting=1	禁止 non-root 模式修改 IDTR 寄存器;将 non-root 模式的 IDTR 设置为影子 IDT 的基地址
MSR bitmaps	禁止 non-root 模式修改 syscall 和 sysenter 相关的 MSR 寄存器;将入口点指向内可信基
I/O bitmaps	禁止 non-root 模式访问 PCI/PCIe 配置空间
CR3-Target controls	允许 non-root 模式在 SAS 和 KAS 之间切换
CR3-Load exiting=1	禁止 non-root 模式切换到其他地址空间

由于 x86 硬件的限制,non-root 模式中的软件执行 CPUID 指令会无条件地陷入 root 模式.AppISO 的处理方式是:内可信基在启动时(仍运行在 root 模式时)执行 CPUID 指令,并将结果保存.此后,应用程序和操作系统以调用的形式直接从内可信基中获得 CPUID 信息.由于 CPUID 只在极少的库函数(libc)中执行,我们只需将 libc 中的 CPUID 指令替换掉,而无需修改应用程序.更简单的方法是,在 root 模式中仅保留一小段代码,专门处理 CPUID 指令.

8.2 进程创建和进程切换

在 Linux 中,应用程序调用 fork()系统调用创建新的子进程,然后子进程调用 exec()系统调用加载自己的可执行文件.对于 fork(),内可信基通过页表验证,保证子进程映射的内存只能是父进程的克隆.对于 exec(),内可信基通过在地址空间中验证文件的 ID 和偏移(第 5.2 节),保证被加载的可执行文件的完整性.

进程切换时,内可信基在软切换过程中对目标应用程序页表进行验证.具体的验证过程与半虚拟化类似,内可信基对系统中所有合法的、已验证的应用程序页表进行跟踪和标记,在切换时只需拒绝所有未标记的页表.操作系统在创建进程时,需要将新应用程序页表向内可信基注册,完成验证.

8.3 系统调用参数传输

与传统的虚拟化方法一样,AppISO 使用复制的方式在操作系统与应用程序之间传递系统调用参数.当发生系统调用时,内可信基将系统调用参数复制到一块共享内存中,将该内存的地址作为参数发送给操作系统.操作系统在处理系统调用过程中对该共享内存进行操作.当操作系统返回时,内可信基再将共享内存中的数据复制回应用程序.

由于操作系统的请求分页机制,当内可信基试图将共享内存中的数据拷贝回应用程序时,应用程序的页框可能还未分配,因而发生缺页中断现象.针对该问题,最简单的方法就是重新进入操作系统分配应用程序页框,但这会增加额外的地址空间切换.AppISO 实现了一种更高效的方式.当应用程序触发系统调用进入内可信基时,内可信基会探测所有的系统调用参数的页框分配情况.如果有页框没有分配,内可信基暂时记录下这次缺页,并在进入操作系统时,在操作系统中模拟该次缺页中断,完成页框分配.这样便有效地避免了额外的 KAS/SAS 切换.

8.4 应用程序的透明性

AppISO 的整个保护机制对上层应用程序是透明的,AppISO 中可以运行任意未修改的应用程序.

在实现应用程序地址空间保护的过程中,AppISO 要求应用程序在调用 mmap()系统调用时,额外维护地址空间状态链表(见第 5.2 节).为了保证应用程序的透明性,我们采取的方案是在标准 c 库中对应用程序的 mmap()系统调用进行截获,通过修改标准 c 库实现应用程序地址空间状态链表的维护.因而,不需要对应用程序本身进行任何修改.

在实现应用程序控制流和 I/O 保护的过程中,AppISO 通过内可信基对中断异常和应用程序的相关系统调用(read()和 write())进行截获,透明地实现应用程序上下文保护和可信文件数据流的构建.

在应用程序启动、退出的过程中,AppISO 通过内可信基对相关系统调用(fork(),exec()和 exit()等)进行截获,透明地实现与安全保护相关的初始化以及程序退出后的清扫工作.此外,考虑到用户需要在应用程序可执行文件中加入自己的密码信息用于 SUID 验证(见第 6.2.3 节),AppISO 提供了一个简单的工具,可直接在未修改的应用程序可执行文件上完成该操作.

9 实验设计和性能分析

本文选择一系列的开源测试用例,测试 AppISO 的性能开销.同时将测试结果与虚拟化方法 Inktag 进行对比.AppISO 和 Inktag 均实现了全面的应用程序保护,包括地址空间保护、控制流完整性保护、文件保护和访问控制等,因此该对比能够比较公平地反映出内可信基方法的优势.实验环境为:CPU Intel i7-3770(4 cores),内存 8GB,操作系统 Linux-kernel-3.4.1,编译环境 gcc-4.3.1.

9.1 内存保护的性能对比

AppISO 和 Inktag 均实现了应用程序地址空间的隔离和完整性保护.然而,Inktag 依赖于虚拟机监控器验证页表,在页表验证过程中需要 6 次昂贵的 non-root/root 切换(如图 4(c)所示);而 AppISO 基于内可信基实现页表验证,仅需要轻量的 2 次 ring 3/ring 0 切换和 2 次 KAS/SAS 切换(如图 4(b)所示).表 3 给出了这 3 类模式切换在本文实验系统中消耗的具体时间.理论上,Inktag 在整个页表验证过程中消耗的切换时间(0.24×6=1.44)大约是 AppISO(0.015×2+0.06×2=0.15)的 9.6 倍.

本文进一步使用 lmbench 测试用例集[12],测量操作系统中实际内存操作的执行时间,并与 Inktag 的实验结果进行对比(见表 4).Page fault 测量缺页中断处理和页表更新验证的执行时间.Mmap lat 测量应用程序调用 mmap 系统调用映射一块内存所消耗的时间.Fork 和 fork+exec 测量进程创建的时间.由于进程创建时需要频繁的内存映射操作,这两个测试用例也能从侧面反映出内存保护的性能.从实验结果可见,在内存保护上,AppISO 相对于 Inktag 有了明显的性能提升,已接近未修改的 Linux 的性能.

Table 3 Execution time of different mode switches

表 3 不同模式切换消耗的时间

Non-Root/Root 切换	ring 3/ring 0 切换	KAS/SAS 切换
0.24us	0.015us	0.06us

Table 4 Results of memory performance benchmarks in lmbench

表 4 lmbench 中内存性能测试用例的实验结果

	Linux	AppISO	AppISO 的开销	Inktag 的开销
page fault	0.17	0.22	1.29x	7.50x
mmap lat	3975	4758	1.20x	9.94x
fork	49	58	1.18x	5.74x
fork+exec	167	194	1.16x	3.04x
2p/0k	0.48	0.67	1.42x	1.41x

此外,2p/0k 测量进程切换消耗的时间.在 AppISO 中,内可信基使用软切换完成进程切换时不同应用程序页表的切换.从测试结果来看,软切换的开销与虚拟化方法的切换开销差不多.需要指出的是,进程切换开销并不是影响应用程序性能的主要因素.

9.2 控制流完整性保护的性能对比

在 Inktag 中,为保护应用程序的控制流完整性,虚拟机监控器截获系统调用和中断异常,在进入和返回操作系统时均需要陷入 root 模式,导致了 4 次 non-root/root 切换.而在 AppISO 中只需要 2 次 ring 3/ring 0 切换和 2 次 KAS/SAS 切换.3 类模式切换的性能已在表 3 中给出.

本文进一步使用 lmbench,测量实际系统调用的执行时间,并与 Inktag 的测试结果进行对比(见表 5).null call 测量应用程序调用一次简单的系统调用(getppid)消耗的时间.Open/Close,file create 和 file delete 对应文件系统相关的系统调用.

Table 5 Results of system call benchmarks in lmbench

表 5 lmbench 中系统调用测试用例的实验结果

	Linux	AppISO	AppISO 的开销	Inktag 的开销
Null call	0.035	0.14	4.00x	55.80x
Open/Close	0.61	0.90	1.48x	7.95x
File create	4.19	4.89	1.17x	2.36x
File delete	3.29	3.91	1.19x	2.23x

9.3 文件I/O的性能对比

AppISO 和 Inktag 均实现了灵活的文件访问控制,但两者在具体方法上有着本质的不同.Inktag 的文件访问控制基于虚拟机监控器来实现;通过加密和哈希技术,保证文件数据的私密性和完整性.而 AppISO 的文件访问控制基于内可信基来实现;通过构建可信文件数据流以保证文件数据的安全,并提出基于内可信基的性能优化.

本文参照 Inktag 的实验环境,在 AppISO 中测量应用程序以不同窗口大小(window size)调用 msync 的执行时间,该 msync 顺序地将一个 256M 的文件从内存刷新到磁盘.图 6 给出了 Inktag 的 msync 实验结果(直接从 Inktag 的图 7 复制而来).图中的 inktag 线给出了 Inktag 完全实现文件保护和访问控制时,msync 的执行时间;inktag-nohash 线给出了不使用加密和哈希、仅实现访问控制时,msync 的执行时间;linux 线作为基准,给出了 Linux 中 msync 执行的时间.Inktag 的开销主要源于:(1) 文件数据的加密和解密(即图中 inktag 线相对于 inktag-nohash 线的增长).(2) 访问控制过程中,为保护元数据(OID、文件偏移、hash 值等)的安全,它们被单独存放在由虚拟机监控器控制的磁盘块中.然而,在元数据和文件数据同步时,这导致了频繁的磁盘调度,造成了较高的开销.(3) 虚拟化本身(比如 non-root/root 切换)也会对文件性能产生影响.图 6 中,inktag-nohash 线相对于 linux 线的增长体现了后两类开销.

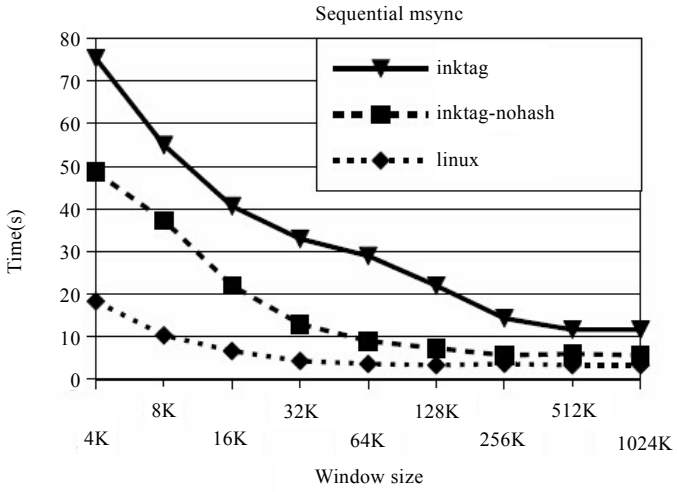


Fig.6 Results of msync in Inktag

图 6 Inktag 中 msync 的实验结果

图 7 给出了 AppISO 的实验结果,msync 的性能接近 Linux.原因是:(1) AppISO 避免了低效的加密解密;

(2) 由于可信文件数据流的构建,访问控制相关的元数据(SUID,文件偏移)不必再单独存放、单独保护,它们与文件数据存放在一起,在可信文件数据流中安全传输,因而避免了频繁的磁盘调度;(3) AppISO 避免了虚拟化造成的 non-root/root 切换.

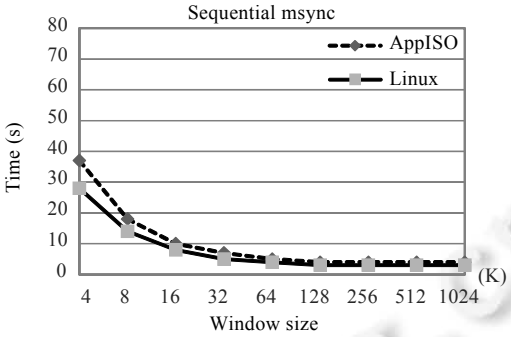


Fig.7 Results of msync in AppISO

图 7 AppISO 中 msync 的实验结果

9.4 应用程序性能对比

与 Inktag 一样,本文使用 DokuWiki 测试应用程序的性能.DokuWiki 在运行过程中映射大量的文件和匿名内存(anonymous memory),因而可以反映出内存保护和文件保护性能的最坏情况.本文参照 Inktag 实验中 DokuWiki 的配置情况和实验方法(具体细节不再重复),将 DokuWiki 运行在 AppISO 中,并将 AppISO 的所有保护机制(内存保护、控制流完整性保护和文件访问控制等)应用到 DokuWiki 上,最后测量 DokuWiki 的吞吐量(throughput).表 6 给出了实验结果,AppISO 的开销仅 1.08x,比 Inktag(1.54x)有了明显的性能优势.

Table 6 Results of DokuWiki

表 6 DokuWiki 的实验结果

	Linux	AppISO	AppISO 的开销	Inktag 的开销
DokuWiki throughput	14.8 req/s	13.7 req/s	1.08x	1.54x

此外,本文选择 Phoronix Test Suite 测试集中一系列应用程序测试用例,来测试其他应用程序在 AppISO 中的开销.实验结果如图 8 所示,测试用例包括 computed-bound 和 I/O-bound 两类.I/O-bound 类应用程序 kernel build 和 postmark 中大量的内存和文件操作带来了一定的开销.

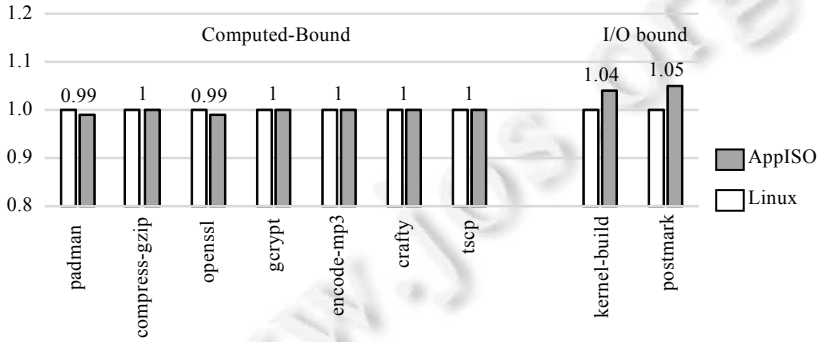


Fig.8 Results of application benchmarks

图 8 应用程序测试结果

10 相关工作

Overshadow[2]和 Inktag[1]基于传统虚拟化技术,依赖于更高特权层的 hypervisor 实现应用程序保护.Inktag 进一步提出 Paraverification 机制,使得应用程序和不可信内核参与到安全验证过程中,降低了 hypervisor 安全验

证的复杂度,从而有效地减小了可信基规模和系统开销.与 Inktag 相比,AppISO 提出了一种完全不同的保护机制,在不可信内核同一层构建内可信基,实现应用程序保护.该方案有效地避免了传统虚拟化方法由于特权层切换造成的高开销问题.其次,AppISO 提出在不可信内核中构建可信文件数据流,保证应用程序 I/O 的安全,有效地避免了 Inktag 中低效的加密和哈希计算,明显地提高了 I/O 性能.另一方面,AppISO 也借鉴了 Inktag 的 Paraverification 机制,将该机制应用到内可信基的设计实现中,从而有效地减小了内可信基的复杂度和系统开销.此外,需要指出的是,虽然 Inktag 提供了更加精细的文件访问控制和一致性保护,AppISO 基于内可信基同样也能够实现这些保护技术,只不过额外的代价是会增加可信基的规模.

Intel 提供了 SMEP 和 SMAP 机制[13],保证运行在 ring 0 层的操作系统无法执行或访问在页表中映射为 ring 3 权限的代码或数据,但是不可信操作系统仍然可以通过修改页表来绕过该机制.Flicker[14]基于 TPM 硬件保护应用程序中的安全敏感代码片段.然而,由于 TPM 本身的限制,它们无法提供全面的应用程序保护.SICE[15]利用 x86 硬件的系统管理模式(SMM)在不可信的虚拟机监控器中创建隔离的虚拟机.同样的方法也能应用到保护不可信操作系统中的应用程序,但使用 SMM 模式进行隔离会导致很大的性能开销,且无法提供全面的应用程序保护和页粒度级的内存保护.其他工作通过改变硬件体系结构[16−20],实现应用程序隔离.Virtual Ghost[21]提出基于编译器插装和传统的 SVM(secure virtual machine)技术,构建可信硬件层,实现应用程序保护.然而,其实现依赖于编译器的可信,但现有安全报告表明,编译器(如 gcc)仍然存在许多安全漏洞[22].而且对操作系统代码进行插装,也带来了较大的开销.

此外,针对操作系统的不可信问题,其他工作使用虚拟机自省技术对不可信操作系统进行监控和验证[23−25],或者直接保护操作系统本身安全,比如 Hooksafe[26]保护操作系统中函数钩子,OSck[27]保护操作系统中的动态数据,Secvisor[28]保护操作系统中的代码完整性等等.

11 总 结

本文提出了在不可信操作系统的同一层构建安全隔离的内可信基,并详细介绍了内可信基的内存保护机制、影子 IDT 机制和 I/O 验证机制,以及在此之上实现的全面应用程序保护.本文的安全分析表明,内可信基方法具有与传统虚拟化方法一样的高安全性,实验结果和分析也表明,它在性能方面具有明显的提高.

References:

[1] Hofmann OS, Kim S, Dunn AM. Inktag: Secure applications on an untrusted operating system. In: Proc. of the Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS). 2013. 265−278. [doi: 10.1145/2451116.2451146]

[2] Chen X, Garfinkel T, Lewis EC, Subrahmanyam P, Waldspurger CA, Boneh D, Dwoskin J, Ports DR. Overshadow: A virtualization-based approach to retrofitting protection in commodity operating systems. In: Proc. of the Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS). 2008. 2−13. [doi: 10.1145/1346281.1346284]

[3] McCune JM, Li Y, Qu N, Zhou Z, Datta A, Gligor V, Perrig A. Trustvisor: Efficient TCB reduction and attestation. In: Proc. of the IEEE Symp. on Security and Privacy (Oakland). 2010. 143−158. [doi: 10.1109/SP.2010.17]

[4] Raoul A, Frank P. Fides: Selectively hardening software application components against kernel-level or process-level malware. In: Proc. of the ACM Conf. on Computer and Communications Security (CCS). 2012. 2−13. [doi: 10.1145/2382196.2382200]

[5] Zongwei Z, Virgil DG, James N, Jonathan M. Building verifiable trusted path on commodity x86 computers. In: Proc. of the IEEE Symp. on Security and Privacy (Oakland). 2012. 616−630. [doi: 10.1109/SP.2012.42]

[6] Richard TM, Lionel L, David L. Splitting interfaces: Making trust between applications and operating systems configurable. In: Proc. of the USENIX Symp. on Operating System Design and Implementation (OSDI). 2006. 279−292. https://www.usenix.org/legacy/event/osdi06/

[7] Intel Corporation. Intel 64 and IA-32 Architectures Software Developer's Manual. 2013.

[8] Stephen C, Hovav S. Iago attacks: Why the system call API is a bad untrusted RPC interface. In: Proc. of the Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS). 2013. 253−264. [doi: 10.1145/2451116.2451145]

[9] Intel Corporation. Serial ATA Advanced Host Controller Interface (AHCI) 1.3. 2008.

[10] AMD. AMD 64 Architecture Programmer's Manual: Volume 2: System Programming. 2011.

[11] Intel. Intel Trusted Execution Technology Preliminary Architecture Specification. 2006.

[12] McVoy L, Staelin C. Lmbench: Portable tools for performance analysis. In: Proc. of the USENIX Annual Technical Conf. 1996. 23. https://www.usenix.org/legacy/publications/library/proceedings/sd96/

[13] Intel Corporation. Intel Architecture Instruction Set Extensions Programming Reference. 2012.

[14] McCune JM, Parno B, Perrig A, Reiter MK, Isozaki H. Flicker: An execution infrastructure for tcb minimization. In: Proc. of the ACM European Conf. in Computer Systems (EuroSys). 2008. 315−328. [doi: 10.1145/1352592.1352625]

[15] Azab A, Ning P, Zhang X. Sice: A hardware-level strongly isolated computing environment for x86 multi-core platforms. In: Proc. of the ACM Conf. on Computer and Communications Security (CCS). 2011. 375−388. [doi: 10.1145/2046707.2046752]

[16] Dwoskin JS, Lee RB. Hardware-Rooted trust for secure key management and transient trust. In: Proc. of the ACM Conf. on Computer and Communications Security (CCS). 2007. 389−400. [doi: 10.1145/1315245.1315294]

[17] Lee RB, Kwan PCS, McGregor JP, Dwoskin J, Wang Z. Architecture for protecting critical secrets in microprocessors. In: Proc. of the Int'l Symp. on Computer Architecture (ISCA). 2005. 2−13. [doi: 10.1109/ISCA.2005.14]

[18] Lie D, Thekkath CA, Horowitz M. Implementing an untrusted operating system on trusted hardware. In: Proc. of the ACM Symp. on Operating Systems Principles (SOSP). 2003. 178−192. [doi: 10.1145/945445.945463]

[19] Lie D, Thekkath CA, Mitchell M, Lincoln P. Architectural support for copy and tamper resistant software. In: Proc. of the Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS). 2000. 168−177. [doi: 10.1145/378993.379237]

[20] Shi W, Fryman JB, Gu G, Lee HHS, Zhang Y, Yang J. Infoshield: A security architecture for protecting information usage in memory. In: Proc. of the Int'l Symp. on High Performance Computer Architecture (HPCA). 2006. 222−231. [doi: 10.1109/HPCA.2006.1598131]

[21] Criswell J, Dautenhahn N, Adve V. Virtual ghost: Protecting applications from hostile operating systems. In: Proc. of the Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS). 2014. 81−96. [doi: 10.1145/2541940.2541986]

[22] Yang X, Chen Y, Eide E, Regehr J. Finding and understanding bugs in C compilers. In: Proc. of the 32nd ACM SIGPLAN Conf. on Programming Language Design and Implementation. 2011. 283−294. [doi: 10.1145/1993498.1993532]

[23] Dolan B, Leek T, Zhivich M, Giffin J, Lee W. Virtuoso: Narrowing the semantic gap in virtual machine introspection. In: Proc. of the IEEE Symp. on Security and Privacy (Oakland). 2011. 297−312. [doi: 10.1109/SP.2011.11]

[24] Fu Y, Lin Z. Space traveling across VM: Automatically bridging the semantic gap in virtual machine introspection via online kernel data redirection. In: Proc. of the IEEE Symp. on Security and Privacy (Oakland). 2012. 586−600. [doi: 10.1109/SP.2012.40]

[25] Srinivasan D, Wang Z, Jiang X, Xu D. Process out-grafting: An efficient out-of-VM approach for fine-grained process execution monitoring. In: Proc. of the ACM Conf. on Computer and Communications Security (CCS). 2011. 363−374. [doi: 10.1145/2046707.2046751]

[26] Wang Z, Jiang X, Cui W, Ning P. Countering kernel rootkits with lightweight hook protection. In: Proc. of the ACM Conf. on Computer and Communications Security (CCS). 2009. 545−554. [doi: 10.1145/1653662.1653728]

[27] Hofmann OS, Dunn AM, Kim S, Roy I, Witchel E. Ensuring operating system kernel integrity with OSCK. In: Proc. of the Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS). 2011. 279−290. [doi: 10.1145/1950365.1950398]

[28] Seshadri A, Luk M, Qu N, Perrig A. Secvisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity oses. In: Proc. of the ACM Symp. on Operating Systems Principles (SOSP). 2007. 335−350. [doi: 10.1145/1294261.1294294]

邓良(1987－),男,湖南长沙人,博士生,主要研究领域为操作系统安全,虚拟化安全.

曾庆凯(1963－),男,博士,教授,博士生导师,CCF 高级会员,主要研究领域为信息安全,分布计算.