软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2016,27(5):1309−1324 [doi: 10.13328/j.cnki.jos.005017]
©中国科学院软件研究所版权所有.
E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

# 一种在不可信操作系统内核中高效保护应用程序的方法*

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)

通讯作者: 曾庆凯, E-mail: zqk@nju.edu.cn

**摘 要**: 在现代操作系统中,内核运行在最高特权层,管理底层硬件并向上层应用程序提供系统服务,因而安全敏感的应用程序很容易受到来自底层不可信内核的攻击.提出了一种在不可信操作系统内核中保护应用程序的方法AppFort.针对现有方法的高开销问题,AppFort 结合 x86 硬件机制(操作数地址长度)、内核代码完整性保护和内核控制流完整性保护,对不可信内核的硬件操作和软件行为进行截获和验证,从而高效地保证应用程序的内存、控制流和文件 I/O 安全.实验结果表明:AppFort 的开销极小,与现有工作相比明显提高了性能.

**关键词**: 不可信操作系统内核;应用程序保护;操作数地址长度

**中图法分类号**: TP316

中文引用格式: 邓良,曾庆凯.一种在不可信操作系统内核中高效保护应用程序的方法.软件学报,2016,27(5):1309−1324. http://www.jos.org.cn/1000-9825/5017.htm

英文引用格式: Deng L, Zeng QK. Method to efficiently protect applications from untrusted OS kernel. Ruan Jian Xue Bao/ Journal of Software, 2016,27(5):1309−1324 (in Chinese). http://www.jos.org.cn/1000-9825/5017.htm

## Method to Efficiently Protect Applications from Untrusted OS Kernel

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**: In commodity OS, the OS kernel runs in the highest privilege layer to manage hardware resources and provides system services. Thus, security-sensitive applications are vulnerable to compromises the underlying untrusted kernel. In this paper, an approach named AppFort is proposed to protect applications from an untrusted OS kernel. To address the high overheads of existing solutions, AppFort makes use of the unique combination of an x86 hardware feature (operand address size), kernel code integrity protection and kernel control flow integrity protection, to intercept and verify both hardware and software operations of the untrusted kernel. As a result, AppFort efficiently protects application's memory, control flows and file I/O, even if the kernel is fully compromised. Experimental results demonstrate that AppFort only incurs very small overhead, which is much better than previous work.

**Key words**: untrusted OS kernel; application protection; operand address size

现代操作系统内核的代码量庞大,结构复杂,存在大量攻击窗口,而且越来越多的漏洞报告和攻击案例表明,内核的安全问题十分严峻.然而,安全敏感的应用程序不可避免地将内核作为可信基,管理底层硬件和提供

* 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); Program B for Outstanding Ph.D. Candidate of Nanjing University of China (2015)

收稿时间: 2015-06-29; 修改时间: 2015-09-23; 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 13:50:57, http://www.cnki.net/kcms/detail/11.2560.TP.20160118.1350.003.html

系统服务.当内核被攻陷后,敏感应用程序通常很难抵御来自底层的不可信内核的攻击.近年来,在不可信内核中保护敏感应用程序的安全已成为操作系统安全领域的热点问题之一.其中,一系列现有工作关注于如何在不可信内核中实现全面的应用程序保护(whole-application protection),包括应用程序内存保护、控制流保护和文件 I/O 保护等.根据底层实现机制的不同,这些工作可分为两类:

- 虚拟化方法(overshadow[1],inktag[2])

该方法在不可信内核底层引入更高权限的 hypervisor,基于虚拟化技术(包括内存虚拟化、中断虚拟化和 I/O 虚拟化等)对底层硬件操作和内核行为进行截获和验证,从而为敏感应用程序提供安全的执行环境.然而,虚拟化方法需要频繁的特权层切换,对于那些内存和 I/O 操作频繁的应用程序会造成很大的性能开销.

- 软件插装方法(virtual ghost[3])

该方法不需要引入更高的特权层,而是在不可信内核的同一特权层引入可信基(SVA-VM).通过对内核代码进行插装,该方法一方面保证内核只能使用 SVA-VM 提供的接口访问底层硬件,另一方面也保证内核无法访问敏感应用程序的内存和可信基 SVA-VM 的内存.然而,软件插装方法需要对内核代码中的每一条访存指令进行插装和运行时检查,同样造成了很高的性能开销.

针对现有方法的高开销问题,本文结合 x86 硬件机制,提出了一种在不可信内核中高效保护敏感应用程序的新方法 AppFort.和 Virtual Ghost 一样,AppFort 在不可信内核的同一特权层引入可信基(称为 FortVisor),对内核中所有的底层硬件操作进行截获和验证.但是,AppFort 不需要对内核代码中的访存指令进行插装,而是结合 x86 硬件机制(指令地址长度)、内核控制流完整性保护和内核代码完整性保护,对内核能够访问的地址空间进行限制,从而保证应用程序和可信基 FortVisor 的安全.AppFort 为敏感应用程序提供了以下安全保护:

1) 内存保护:不可信内核无法读写应用程序地址空间中的数据和代码.
2) 控制流保护:不可信内核无法修改应用程序的控制流,也无法访问应用程序上下文中的敏感数据.
3) 文件 I/O 保护:不可信内核无法窃取或者篡改应用程序的文件数据,也无法访问交换(swap)到磁盘上的应用程序数据.

本文在 Linux 3.8.0 上实现了 AppFort 的原型系统,并使用一系列内核和应用程序测试用例测量 AppFort 的性能.实验结果表明,AppFort 在各种测试用例上的性能开销均十分微小,与现有工作相比,AppFort 的性能有明显的提升.

## 1 攻击模型

本文假设内核本身并不是恶意的,但是内核存在漏洞可能被攻陷或者加载了不可信内核模块(本文以后统称为不可信内核).在整个系统中,不可信内核运行在最高特权层,在内核之下不存在更高特权层的软件(比如 hypervisor).因而,不可信内核能够完全控制底层硬件资源、执行内存中任意代码、读写内存或磁盘中的任意数据以及实施恶意的 DMA(direct memory access)攻击.本文不考虑敏感应用程序中的漏洞.在现实中,敏感应用程序的代码量比内核小得多,通常经过了充分的验证和测试,因而它们之中存在漏洞的几率比内核小得多.

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 I/O 等.下面将详细讨论内核对应用程序可能的攻击方式,构建具体的攻击模型.

### 1.1 攻击内存数据

由于内核运行在最高特权层,它不仅能够直接读写应用程序数据,而且也拥有足够的权限操作底层硬件来实现恶意的内存访问.下面对这些攻击加以介绍.

#### 1.1.1 直接访问

在传统操作系统中,内核和应用程序运行在同一个地址空间.x86 硬件在页表中提供 supervisor 位,防止运行在低特权层的应用程序访问高权限内核的内存,但是并不会限制内核访问应用程序的内存.因此,不可信内核能够直接读写、执行应用程序的数据和代码,破坏它们的私密性和完整性.此外,在应用程序请求内核服务时(系统调用),内核和应用程序之间需要进行数据交互,这使得应用程序数据的保护更加复杂.

© 中国科学院软件研究所 http://www.jos.org.cn

#### 1.1.2 修改页表

除了直接访问以外,内核还具有足够的权限修改页表、对应用程序的页框进行恶意映射(修改现有映射或者重映射),实现对应用程序数据代码的读写、执行.事实上,出于对性能的考虑,传统操作系统中,内核原本就会对每个应用程序页框进行重映射(或者被称为 double-mapping).

#### 1.1.3 DMA 攻击

DMA 是现代 I/O 设备中普遍支持的一种高速数据传输操作,允许在 I/O 设备和内存之间直接传送数据.在我们的假设中,内核有足够的权限操控 I/O 设备,从而利用 DMA 操作恶意读写任意物理内存中的数据,包括应用程序的数据.

#### 1.1.4 Iago 攻击

Iago 攻击是近年来新提出的攻击方式.其主要思想是:由于应用程序仍然依赖于不可信内核来提供服务(系统调用),内核可返回一系列精心选择的系统调用返回值,从而间接实现修改应用程序数据的目的.例如,当应用程序调用 *mmap*()系统调用申请内存时,内核可以将该应用程序当前栈的地址作为返回值,造成 *mmap* 内存与栈重叠.因而,当应用程序自己修改 *mmap* 内存时,就会修改到栈上的数据(比如函数返回地址),造成应用程序控制流或者数据被破坏.

### 1.2 劫持控制流

在应用程序执行过程中,其控制流可能随时中断和被异常打断.应用程序自身也需要发起系统调用,请求内核服务.当这些事件发生时,内核能够获得应用程序的执行上下文,窃取其中的敏感信息或者修改应用程序的控制流.

### 1.3 攻击文件I/O

在传统操作系统中,应用程序依靠内核实现与磁盘外设之间的数据传送,包括从磁盘中获得可执行代码、在磁盘上存储文件等.因而,内核能够轻易破坏应用程序文件数据的私密性和完整性.此外,当系统内存不够时,内核需要将应用程序的数据交换(swap)到磁盘上,导致这些数据也十分容易受到内核的攻击.

## 2 概 述

### 2.1 背景知识:x86中的操作数地址长度

在 x86 中,可通过指令前缀修改指令操作数的寻址地址长度(address size).在 x86_64 中,每条指令默认的操作数地址长度是 64 位,能够寻址整个$(0,2^{64})$地址空间.如果在指令前添加前缀(0x67),则地址长度变为 32 位,该指令只能寻址(0,4G)的地址空间.x86 提供该硬件机制原本是为了在 64 位系统中兼容运行 32 位程序,然而 AppFort 利用该机制,在不可信内核中实现了轻量级的可信基和应用程序的隔离和保护技术.

### 2.2 AppFort概述

图 1 给出了 AppFort 中的地址空间布局.AppFort 仍然让内核和应用程序运行在同一个地址空间,内核运行在 CPU 的内核态,应用程序运行在 CPU 的用户态.然而与传统地址空间布局不同的是,AppFort 将内核运行在低地址空间(0,4G),将应用程序运行在高地址空间$(4G,2^{64})$.在该地址空间布局基础上,AppFort 对内核代码中所有的访存指令进行改写,在每条访存指令之前加入前缀(0x67),将操作数地址长度强制修改为 32 位.因此,内核只能访问(0,4G)内自己的数据和代码,无法访问位于地址空间$(4G,2^{64})$中应用程序的数据和代码.

Fig.1 Address space layout of AppFort

图 1 AppFort 的地址空间布局

然而,该方式只能防止直接访问攻击(第 1.1.1 节),最高权限的内核仍然能够操作底层硬件(包括修改页表、DMA)来实现对应用程序数据和代码的攻击.比如,通过修改页表,将应用程序页框重映射到内核能够访问的地址区间(0,4G),或者直接修改内核代码,去掉指令前缀.因此,AppFort 进一步引入可信基 FortVisor.FortVisor 的数据和代码也被映射到地址空间(4G,$2^{64}$)内,无法被内核直接修改.AppFort 结合内核控制流完整性保护和内核代码完整性保护,利用 FortVisor 截获并验证内核中所有的硬件操作(包括修改页表、配置 DMA、中断处理、I/O 访问等),从而禁止内核通过操作底层硬件来实现攻击.

和 Virtual Ghost[1]一样,AppFort 的可信基(FortVisor)与不可信内核运行在同一特权层和同一地址空间,从而有效避免了特权层切换和地址空间切换带来的开销.

## 3 FortVisor 的设计和保护

### 3.1 硬件操作的截获和验证

x86 中,软件要操作底层硬件必须同时满足两个基本条件:(1) 运行在内核态;(2) 需要执行硬件操作对应的特权指令.在 AppFort 中,内核虽然运行在内核态,但是无法执行特权指令.根据构成方式的不同,特权指令可分为 intended 特权指令和 unintended 特权指令.intended 特权指令位于合法的指令边界上,在内核代码中本身就作为特权指令执行;unintended 特权指令位于非法的指令边界上,可能由其他指令的一个或者多个字节偶然形成(可能被 ROP(return-oriented programming)攻击利用[4]).AppFort 对内核代码进行修改,保证内核代码中不包含任何 intended 特权指令;AppFort 基于控制流完整性保护(详见第 3.1.4 节),确保内核无法执行 unintended 特权指令;同时,AppFort 基于内核代码完整性保护(详见第 3.1.4 节),保证内核无法在其代码中重新引入特权指令,或执行其他地方的特权指令.

因此,当内核需要操作底层硬件时,只能使用 FortVisor 提供的硬件访问接口进行操作.从而,FortVisor 能够对所有来自内核的硬件操作请求进行截获和验证.只有当验证通过时,FortVisor 才完成最终的硬件操作.图 2 描述了 FortVisor 的整体架构.

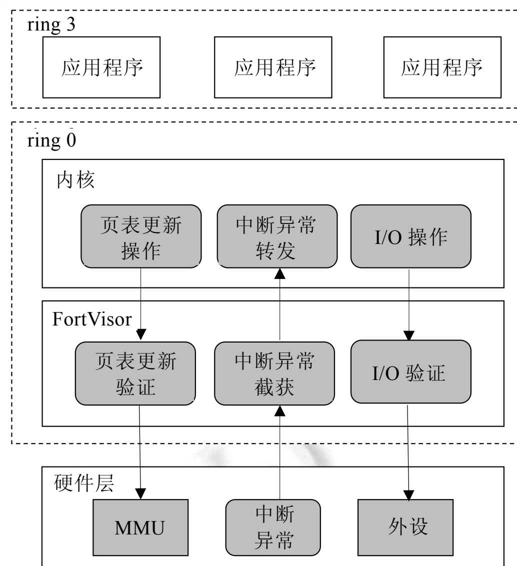


Fig.2 Overview of FortVisor

图 2 FortVisor 的整体架构

内核和 FortVisor 运行在同一特权层(ring 0 层)的同一地址空间.FortVisor 对内核的硬件操作(包括页表更新操作和 I/O 操作)进行截获和验证;只有验证通过时,这些操作才能最终到达硬件(MMU(memory management

© 中国科学院软件研究所 http://www.jos.org.cn

unit)和外设).同时,当硬件层发生中断异常时,FortVisor 首先对这些中断异常进行截获和处理;只有当这些中断异常事件在 FortVisor 中处理完成后,FortVisor 才将它们转发到内核.下面将具体论述 FortVisor 如何设计和实现这些截获和验证操作.

3.1.1 页表更新验证

在 AppFort 中,内核仍然分配和维护自己的页表,但是硬件 MMU(memory management unit)实际使用的是 FortVisor 分配和维护的页表(称为影子页表).具体来说,由于内核代码中不包含任何特权指令,内核只能使用 FortVisor 提供的接口来修改页表基地址寄存器(CR3 寄存器).从而,FortVisor 能够验证所有的 CR3 修改操作,保证 CR3 寄存器总是指向 FortVisor 自己分配的影子页表.其次,所有的影子页表均从 FortVisor 内存中分配.AppFort 将 FortVisor 内存映射到地址空间(4G,$2^{64}$)中,保证内核无法直接访问 FortVisor 内存(以及其中的影子页表).当内核想要把页表更新装配到 MMU 时,它只能使用 FortVisor 提供的接口来更新影子页表.从而,FortVisor 能够对所有的影子页表更新进行验证.在验证过程中,FortVisor 禁止在影子页表中修改 FortVisor 内存的现有映射,或者在影子页表中对 FortVisor 内存进行重映射(比如将 FortVisor 内存重映射到内核可以访问到的地址空间(0,4G)).在这种情况下,内核既无法直接访问 FortVisor 内存,也无法通过修改页表来访问 FortVisor 内存,因而,FortVisor 内存(包括其中的影子页表)始终是安全的.

3.1.2 中断、异常处理和系统调用的截获

在 x86 中,中断异常的处理程序由 IDT(interrupt descriptor table)指定,IDT 的基地址由 IDTR(interrupt descriptor table register)指定,IDTR 由特权指令 lidt 修改.在 AppFort 中,内核只能使用 FortVisor 提供的接口来修改 IDTR、指定 IDT.因而,FortVisor 能够确保 IDTR 始终指向 FortVisor 自己分配的 IDT.该 IDT 及其指向的处理程序均从 FortVisor 内存中分配,无法被内核修改.因此,当中断异常发生时,它们首先会进入 FortVisor 指定的处理程序中进行处理.从而,FortVisor 能够截获每一个中断异常事件.当 FortVisor 处理完这些事件后,再将它们转发到内核的中断异常处理程序.

除了软件异常(int80)以外,x86 也支持快速系统调用(sysenter 和 syscall).快速系统调用的入口地址由特定的 MSR 指定.在 AppFort 中,内核只能使用 FortVisor 提供的接口来修改 MSR、指定快速系统调用的入口点.因而,FortVisor 能够确保该 MSR 始终指向自己的入口点,从而截获所有的快速系统调用.

3.1.3 I/O 和 DMA 验证

在 x86 中,软件可通过两种方式访问和控制 I/O 设备:端口 I/O(PIO)和内存映射 I/O(MMIO).其中,PIO 使用 I/O 特权指令(in 和 out)来操作 I/O 设备;MMIO 将一块内存(I/O 内存)关联到 I/O 设备,通过读写 I/O 内存来实现对 I/O 设备的访问和控制.在 AppFort 中,由于内核代码中不包含 I/O 特权指令,内核只能使用 FortVisor 提供的接口来实现 PIO;AppFort 在 FortVisor 内存中分配所有 I/O 内存,因而内核也只能使用 FortVisor 提供的接口来实现 MMIO.从而,FortVisor 能够对所有的 I/O 操作进行验证.由于 I/O 操作中也包含了 DMA 的配置,FortVisor 也能够对 DMA 操作进行验证,从而防御 DMA 攻击.

3.1.4 内核控制流和代码完整性保护

如前所述,保证内核控制流和代码完整性是实现硬件操作截获和验证的基础,本节将具体论述.

首先,为了保证内核的代码完整性,AppFort 将所有的内核代码映射为 W⊕X.该机制不允许内核修改现有内核代码或者引入新的内核代码.FortVisor 通过页表更新验证(见第 3.1.1 节),确保内核无法破坏 W⊕X 机制.同时,AppFort 利用 x86 的 SMEP(supervisor mode execution protection)机制,确保内核无法执行位于低权限层的代码(比如应用程序代码).由于内核只能使用 FortVisor 提供的接口修改 cr4 寄存器,FortVisor 禁止内核修改 cr4 寄存器的 SMEP 位,从而确保内核无法使 SMEP 机制失效.内核代码的完整性,保证了内核无法在内核代码中重新引入特权指令,或执行其他地方的特权指令.

其次,参照 CCFIR[5],AppFort 对内核代码中的间接分支指令进行插装,实现了 2-ID 的控制流完整性保护策略:所有的间接函数调用只能跳转到某个内核函数的入口点;所有的函数返回只能跳转到某个内核函数调用点的下一条指令.内核控制流完整性确保了内核只能从正常的指令边界开始执行指令,防止内核通过 ROP 攻击,

© 中国科学院软件研究所 http://www.jos.org.cn

从非正常指令边界构建 unintended 特权指令.

内核代码的完整性和控制流的完整性保护,也确保了内核无法去掉或者绕过访存指令的前缀.首先,代码的完整性保证了内核无法通过修改内核代码去掉指令前缀,也无法引入新代码;其次,控制流的完整性保证了内核只能从合法的指令边界开始执行内核指令,从而禁止内核恶意跳转到访存指令的中间执行、确保访存指令的前缀无法被绕过.

### 3.2 FortVisor的保护

- FortVisor 内存保护

FortVisor 内存中包括影子页表、IDT、I/O 内存和 FortVisor 自身的数据和代码.如第 3.1.1 节所述,不可信内核既无法访问映射到$(4G,2^{64})$中的 FortVisor 内存,也无法通过修改页表改变 FortVisor 内存的现有映射或对 FortVisor 内存进行重映射.此外,值得指出的是,虽然 FortVisor 和应用程序都运行在地址空间$(4G,2^{64})$中,但是两者运行在不同的特权层,因而应用程序也无法访问 FortVisor 内存.

- 指定入口点

内核控制流的完整性保证了内核代码中的分支指令只能跳转到内核代码内部的函数入口点或者函数返回点,而无法直接跳转到 FortVisor.内核代码中有且仅有一条直接分支指令可用于从内核跳转到 FortVisor.在该直接分支指令中,FortVisor 的指定入口点作为立即数被写入指令的操作数中,无法被内核修改.因此,内核只能使用这条直接分支指令,从指定入口点进入 FortVisor.由此也可看出,内核与 FortVisor 之间的切换十分轻量,只需要一条直接分支指令.

- 控制流保护

在 FortVisor 的入口点,中断立即被禁止.内核无法中断 FortVisor 的控制流、破坏 FortVisor 的控制流完整性.当不可屏蔽中断(non-markable interrupt,简称 NMI)发生时,FortVisor 基于中断截获机制(见第 3.1.2 节)暂时阻塞该 NMI.当控制流最终返回内核时,再将该 NMI 发送到内核中进行处理.

## 4 应用程序保护

### 4.1 应用程序内存保护

应用程序内存保护的原理与 FortVisor 内存保护类似:AppFort 确保所有存储应用程序数据代码的页框(简称为应用程序页框)只能被映射到地址空间$(4G,2^{64})$中;基于 FortVisor 的页表更新验证,禁止内核修改应用程序页框的现有映射,或者对应用程序页框进行重映射.然而,我们在设计过程中仍然遇到了许多挑战.

1) 出于可信基最小化的考虑,应用程序的所有内存管理工作仍然需要由不可信内核完成,包括应用程序的地址空间映射管理和应用程序页框分配等.然而在这种情况下,如何实现以上安全保护?

2) 当应用程序调用系统调用时,内核需要访问系统调用参数指向的应用程序数据,以实现数据交互.然而,我们的保护机制又不允许内核访问任何应用程序数据.针对该问题,Inktag[2]提出将这些应用程序数据拷贝到内核能够访问的区域,然而,该方式对于有大量数据交互的情况(比如 I/O 系统调用)性能太差.Virtual Ghost[3]提出了一种折中的方法:它要求在应用程序设计过程中,将那些用于交互的应用程序数据预先识别出来,从而不对这些数据进行任何的保护.然而,该方法对于现有的应用程序并不适用.因此,我们需要设计一种更加高效和通用的方式.

下面我们将具体论述如何解决以上挑战,实现应用程序内存保护.

#### 4.1.1 地址空间管理

AppFort 仍然依赖内核来管理应用程序的地址空间.内核处理应用程序的地址空间映射请求(比如 *mmap*() 系统调用),在地址空间中为应用程序分配空闲的虚拟内存区(virtual memory area,简称 VMA).仅当内核将分配好的 VMA 地址(比如作为 *mmap*()的返回值)返回给应用程序时,FortVisor 基于系统调用截获(见第 3.1.2 节),对每一个返回给应用程序的 VMA 进行验证.验证工作主要包括:(1) 分配的 VMA 必须位于$(4G,2^{64})$中;(2) 各个

VMA 之间不能相互重叠(防范内核的 Iago 攻击).

4.1.2 页框分配和映射

当应用程序访问 VMA 中的某个虚拟地址时,由于物理页框还没有分配,该操作会产生缺页中断.AppFort 仍然依赖于内核在其缺页中断处理程序中完成应用程序页框的分配.当页框分配完成后,内核只能使用 FortVisor 提供的接口,在影子页表中映射该页框.从而,FortVisor 能够对内核提供的页框进行验证.在验证中,FortVisor 检查该页框当前映射状态,确保该页框没有映射到地址空间中的任何位置.只有验证通过后,FortVisor 才在影子页表中将该页框映射到应用程序的 VMA.当返回应用程序执行时,该新分配的页框即成为应用程序页框,用于存储应用程序的数据.

从以上设计可以看出,虽然 AppFort 仍然依赖于内核完成应用程序的内存管理,但是 FortVisor 通过截获和验证操作,确保应用程序页框只映射到应用程序的 VMA(所有 VMA 都位于地址空间(4G,$2^{64}$)中);同时,FortVisor 基于页表更新验证,禁止内核修改应用程序页框的现有映射,或者对应用程序页框进行重映射.

4.1.3 内核和应用程序的数据交互

当应用程序调用系统调用时,内核需要读写系统调用参数指向的应用程序数据,以实现内核与应用程序之间的数据交互.在 AppFort 中,当内核需要读写应用程序数据时(比如调用 *copy_to_user*()和 *copy_from_user*()时),由于内核无法访问 4G 以外的地址空间,只能从指定入口点跳转到 FortVisor,向 FortVisor 发出读写请求(包括想要读写的源地址、目的地址和长度).FortVisor 代替内核完成应用程序数据的读写,然后返回内核.同时,FortVisor 对内核的读写请求进行验证,保证内核无法读写系统调用参数定义之外的应用程序数据.值得指出的是,AppFort 之所以能够采用这样一种数据交互方式,是因为内核和 FortVisor 的切换十分轻量(仅需要一条直接分支指令).

### 4.2 应用程序控制流保护

应用程序执行过程中,其控制流可能随时被中断异常和系统调用打断.FortVisor 截获这些事件,在自己的内存中安全地保存应用程序的上下文,然后再将这些事件发送到内核中处理.当内核返回应用程序时,由于内核无法访问应用程序的上下文和数据代码,只能跳转到 FortVisor.然后,FortVisor 安全地恢复应用程序的上下文,并返回应用程序中执行.在整个过程中,内核无法窃取或篡改应用程序的上下文,也无法修改应用程序的控制流.

然而,在传统操作系统中,内核又需要修改应用程序的上下文,以实现信号处理.针对该问题,FortVisor 提供接口,允许内核以一种安全的方式修改应用程序上下文.具体来说,在信号处理过程中,内核向 FortVisor 发出修改应用程序上下文的请求;然后,FortVisor 对该请求进行验证(比如,确保内核只能将应用程序上下文中的指令指针(rip)修改为应用程序自己指定的信号处理程序).此外,当应用程序调用 *sigaction*()系统调用时,FortVisor 会截获该系统调用,记录应用程序的信号处理程序.

### 4.3 应用程序文件I/O保护

在传统操作系统中,应用程序使用内核服务来实现文件 I/O(比如调用 *read*()和 *write*()系统调用),因而,不可信内核能够轻易破坏应用程序文件数据的私密性和完整性.现有工作一般使用加密和哈希技术,防止内核篡改或窃取应用程序的文件数据.在 AppFort 中,我们采用了一种更加高效的方式,即,在不可信内核中构建可信文件数据流来保护应用程序文件 I/O 的安全.

4.3.1 可信文件数据流

图 3 描述了当应用程序调用 *write*()时,文件数据在内核中的流动情况.

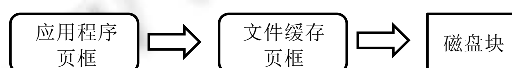


Fig.3 File data flow when applications call *write*()

图 3 应用程序调用 *write*()时的文件数据流

文件数据首先从应用程序页框被传输到操作系统的文件缓存页框,然后再从文件缓存页框被传输到磁盘块中.可信文件数据流保证:文件数据在以上传输过程中,始终无法被不可信内核篡改或窃取.

具体来说,当应用程序调用 *write*()时,AppFort 仍然让内核处理文件 I/O 操作,包括文件名解析、目录管理、文件缓存页框的分配和管理等.但是,内核需要将文件缓存页框向 FortVisor 进行注册.对于每一个注册的文件缓存页框,FortVisor 在影子页表中修改它们的页表映射,仅将它们映射到地址空间(4G,$2^{64}$)中.因此,内核仍然继续管理这些文件缓存页框,但是无法读写文件缓存页框中的数据.

当内核需要将文件数据从应用程序页框传输到文件缓存页框时,由于内核无法访问应用程序页框,它必须跳转到 FortVisor,请求 FortVisor 完成该数据传输操作.然后,FortVisor 对文件缓存页框进行验证,确保文件缓存页框已经注册.因此,当文件数据被传输到这些文件缓存页框后,它们仍然是安全的.

当内核需要将文件数据从文件缓存页框传输到磁盘块时,FortVisor 基于 I/O 验证,对内核的磁盘 I/O 操作进行验证.FortVisor 定义了两种磁盘块的状态:free 和 occupied,同时对所有发送到磁盘的 I/O 命令进行验证.如果该命令是写磁盘块命令,FortVisor 保证目标磁盘块只能是 free 状态.在数据传输完成后,FortVisor 将该磁盘块转变为 occupied 状态,并使用应用程序的标识符 SID(secure identifier,稍后论述)标记该磁盘块上数据的所有者.此后,FortVisor 基于 I/O 验证,确保所有 occupied 磁盘块只能被它们的所有者访问,保证磁盘块上文件数据的安全.

当应用程序调用 *read*(),试图读取磁盘块上的文件数据时,可信文件数据流的保护十分类似.FortVisor 只允许 occupied 磁盘块上的文件数据传输到已注册的文件缓存页框,并用 SID 标记该文件缓存页框.此后,FortVisor 只允许该文件缓存页框上的数据传输到具有相同 SID 的应用程序页框.

值得指出的是,在可信文件数据流的构建过程中,虽然内核需要多次调用 FortVisor,但是内核与 FortVisor 之间的切换十分轻量(仅需要一条直接分支指令),并不会明显增加系统开销.

#### 4.3.2 应用程序 SID 的获取

AppFort 使用 SID 来标示应用程序文件数据的所有者.和 Virtual Ghost 一样,FortVisor 使用公钥/私钥对,安全地从应用程序可执行文件的数据段获得应用程序的 SID.由于每个 SID 都被公钥加密,内核无法窃取.不同的应用程序之间也可以共享 SID,实现文件数据的共享.

## 5 重定位问题

在 AppFort 的设计中,必须将内核重定位到地址空间(0,4G);将应用程序重定位到地址空间(4G,$2^{64}$).只有这样,才能保证内核在地址空间(0,4G)中正常执行,同时,不能访问地址空间(4G,$2^{64}$)中应用程序的数据代码.

### 5.1 内核重定位

在传统 Linux 中,内核地址空间中包括 5 个虚拟内存区域:直接映射区(direct mapping area)、vmalloc 映射区、vmemmap 映射区、内核代码映射区和内核模块映射区.下面将具体论述如何将每一个区域重定位到(0,4G).

**直接映射区**:直接映射区用于线性映射整个物理内存,即,直接映射区的大小和物理内存一样.然而,实际系统中的物理内存往往大于 4G,因而地址空间(0,4G)无法容纳下整个直接映射区,造成重定位失败.

为了解决直接映射区的重定位问题,我们首先思考:为什么内核在执行过程中需要直接映射区?这是由于内核使用直接映射区映射整个物理内存,从而可以访问物理内存任意页框中的数据,包括内核自己的数据和应用程序页框中的数据.然而在 AppFort 中,内核并不需要直接访问应用程序的数据,如第 4.1.3 节所述,应用程序数据的访问由 FortVisor 代替内核完成.因此,直接映射区只需要映射物理内存中很小的一部分(即那些存储内核数据的页框).在这种情况下,地址空间(0,4G)足以容纳下直接映射区.具体来说,在内核初始化时,物理内存被划分为两块区域:内核区和应用程序区.内核的数据分配请求(比如 *kmalloc*(),*kmem_cache_alloc*()等)从内核区分配,应用程序页框从应用程序区分配.内核区的大小可基于实际物理内存的大小静态配置,或基于内核的内存消耗进行动态调整.我们通过大量测试发现:在实际系统中,内核在内核区中消耗的物理内存通常远远小于 1GB.因此,我们在地址空间(0,4G)中分出 2G 大小的空间,即(2G,4G),用于构建直接映射区,线性映射内核区.

此外,由于 FortVisor 需要同时访问内核区和应用程序区,FortVisor 在地址空间(4G,$2^{64}$)分出一块足够大的区

© 中国科学院软件研究所 http://www.jos.org.cn

域线性映射整个物理内存.

**vmalloc 映射区**:vmalloc 映射区用于映射内核中由 *vmalloc*()分配的物理内存,为非线性映射.AppFort 在地址空间(0,4G)中分出 512M 大小的空间,即(1536M,2G),用于构建 vmalloc 映射区.

**vmemmap 映射区**:vmemmap 映射区用于映射内核中的 *page* 对象数组,其中,每个 *page* 对象对应物理内存中的一个页框.对于一个物理内存小于 48G 的实际系统,*page* 对象数组小于 480M.在这种情况下,AppFort 在地址空间(0,4G)中分出 512M 大小的空间,即(1G,1536M),用于构建 vmemmap 映射区.

对于物理内存大于 48G 的系统,AppFort 仍然在地址空间(4G,$2^{64}$)中构建 vmemmap 映射区.在这种情况下,内核需要跳转到 FortVisor,使用 FortVisor 提供的接口来访问 *page* 对象数组.由于内核不需要经常访问 *page* 对象数组,这样的设计并不会增加太大的开销.

**内核代码和内核模块映射区**:内核代码和内核模块的大小通常小于 5M,AppFort 在地址空间(0,4G)中分出 512M 大小的空间,即(512M, 1G),用于映射内核代码和所有内核模块.该映射区的大小足够容纳 100 个以上的内核模块,完全能够满足实际系统的需求.

### 5.2 应用程序重定位

应用程序地址空间中包括 5 种类型的 VMA:可执行文件 VMA、堆 VMA、栈 VMA、匿名映射 VMA、文件映射 VMA.5 种 VMA 中,除可执行文件 VMA 外,其他 VMA 的重定位可直接由内核实现(通过少量内核修改).

在 Linux 中,编译后的可执行文件一般是不能被重定位的,它们只能被加载到地址空间中固定的虚拟地址(0x400000).因此,对于一个需要保护的敏感应用程序,AppFort 需要获得它的源代码,并进行重新编译,从而将可执行文件的加载地址重定位到(4G,$2^{64}$).但是,对于其他不需要保护的应用程序,AppFort 仍然允许它们将可执行文件加载到原来的虚拟地址.AppFort 在地址空间(0,4G)中保留 512M 大小的区域,即(0,512M),仍然作为应用程序地址空间,映射这些应用程序的可执行文件.因此,对于不需要保护的应用程序,AppFort 并不要求获得它们的源代码.

AppFort 与现有工作 Inktag[2],Virtual Ghost[3]一样,都需要获得被保护应用程序的源代码.然而,AppFort 并不需要对源代码作任何修改,只需要重编译.此外,值得指出的是,虽然其他 VMA 的重定位是由不可信内核实现的,但是当这些 VMA 返回给应用程序时,FortVisor 会对每个 VMA 的地址进行验证,以确保它们已经重定位到(4G, $2^{64}$),详见第 4.1.1 节.

图 4 描述了实现内核和应用程序重定位以后的地址空间布局.

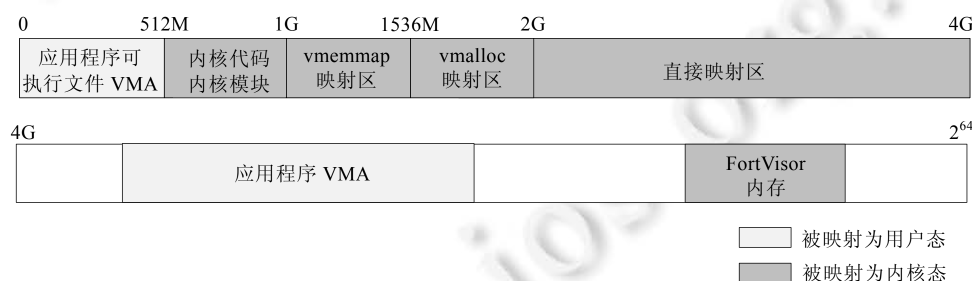


Fig.4 Address space layout after relocation

图 4 重定位后的地址空间布局

## 6 实 现

本文在 Linux 3.8.0 上实现了 AppFort 的原型系统.在我们目前的实现方案中,AppFort 采取了简单的、预设定的保护策略,即,对所有运行在不可信内核上的应用程序都进行保护,包括保护应用程序所有内存的安全、控制流的完整性以及保护所有通过系统调用 *read*()和 *write*()实现的文件 I/O 的安全.如何提供更加灵活的保护策略控制方式,我们将在第 10 节的未来工作中讨论.当前实现中,可信基 FortVisor 仅包含约 5.5k 行代码.

### 6.1 内核代码改写

在 AppFort 的实现中,我们提供了 rewriter 工具对内核代码进行改写.首先,编译器将 Linux 内核源代码编译成汇编代码;然后,rewriter 遍历内核汇编代码中每一条指令.对于访存指令,rewriter 添加指令前缀(0x67),将该访存指令的操作数地址长度修改为 32 位;对于间接分支指令,rewriter 进行插装,保证内核控制流的完整性.对于特权指令,rewriter 将它替换为一条指向 FortVisor 的直接分支指令.

然而,x86 中存在一类特殊的访存指令,即栈指令(包括 *push*,*pop*,*call* 和 *ret*).当该类指令访问栈上的内存时,其操作数地址长度总是为 64 位,无法被修改为 32 位.因而,不可信内核可能利用这类栈指令恶意访问地址空间(4G,$2^{64}$).针对该问题,我们采用了 MIP[6]中提出的方法,对栈指针(rsp 寄存器)进行限制.具体来说,rewriter 对内核代码中的每一条修改 rsp 的指令进行改写,比如,将 add rsp,0x10 改写为 add esp,0x10.由于修改 esp 将自动地把 rsp 的高 32 为清零,因而 rsp 寄存器的值将始终被限制在(0,4G)的范围内.从而,栈指令也只能访问(0,4G)内的地址空间.

### 6.2 FortVisor初始化

和传统的 hypervisor 一样,FortVisor 在不可信内核之前启动,并初始化整个系统的安全环境.首先,FortVisor 完成必要的硬件初始化和配置工作,包括初始化影子页表、IDT 和 I/O 内存等;其次,FortVisor 基于 hash 对内核代码进行验证,确保第 6.1 节中的内核代码改写已完全实现;当验证通过后,FortVisor 进一步保证内核的代码完整性;最后,FortVisor 利用 Linux 的半虚拟化接口,启动内核执行.

### 6.3 启动应用程序

在 Linux 中,进程通过调用 *exec*()系统调用来启动一个新的应用程序执行.AppFort 仍然让内核完成绝大部分的应用程序初始化工作,然而应用程序的执行上下文由 FortVisor 准备.同时,FortVisor 确保该执行上下文指向的应用程序 VMA 是合法的:上下文中的指令指针(rip)指向的已可执行文件 VMA 以及栈指针(rsp)指向的栈 VMA 必须已经重定位到(4G,$2^{64}$).

在进程切换时,AppFort 仍然依赖内核实现复杂的进程调度算法.仅当内核返回应用程序执行时,才需要跳转到 FortVisor.然后,FortVisor 在 CPU 寄存器中装载应用程序的上下文,并跳转到应用程序中执行.

### 6.4 内核模块加载

在内核执行过程中,可能需要动态地加载内核模块.针对该问题,FortVisor 向内核提供接口,允许内核动态提交新的内核代码.同时,FortVisor 对新代码进行验证(类似 CCFIR[5]中的 Verifier),确保所有特权指令已被消除、访存指令已被改写、间接分支指令已被保护.只要当验证通过时,内核才在地址空间中将新的内核模块代码映射为可执行.

## 7 安全性分析

在本文的第 1 节,我们把不可信内核对上层应用程序的攻击主要分为 3 类:1) 攻击内存;2) 劫持控制流;3) 攻击文件 I/O.我们假定的攻击模型主要考虑这 3 类攻击方式,不考虑诸如内核拒绝服务攻击等其他方式.该攻击模型假定与现有相关工作(如 Overshadow[1],Inktag[2]和 Virtual Ghost[3])是一致的.

本节结合攻击模型,系统性地分析总结 AppFort 针对每类攻击的防护措施,阐明 AppFort 的安全性.

1) 攻击内存的防御

攻击内存可进一步分为 4 种子攻击方式:直接访问攻击、修改页表攻击、DMA 攻击和 Iago 攻击(第 1.1 节论述).如论文 Virtual Ghost 所述,这 4 种子攻击方式基本涵盖了目前已知的内核攻击应用程序内存的所有方式.

为了避免直接访问攻击,AppFort 确保所有的应用程序页框只能被映射到地址空间(4G,$2^{64}$)中.由于内核代码中所有的访存指令都被加上了指令前缀,因而无法直接访问 4G 以外的应用程序的内存(第 4.1 节论述).对内核代码完整性和控制流完整性的保护,也确保了内核无法去掉或者绕过访存指令的前缀.

© 中国科学院软件研究所 http://www.jos.org.cn

为了防止修改页表攻击,内核中所有的页表操作都被 FortVisor 截获和验证(第 3.1.1 节论述).FortVisor 禁止内核恶意修改页表,包括禁止内核修改应用程序内存的现有映射,或者将应用程序内存重映射到 4G 以内(第 4.1 节论述).

为防护 DMA 攻击,FortVisor 对内核所有的 I/O 操作进行验证,禁止任何恶意的 DMA 行为(第 3.1.3 节论述).

为了防御 Iago 攻击,FortVisor 对所有从内核返回的应用程序 VMA 进行验证,确保每个 VMA 之间不能相互重叠,从而有效防范 Iago 攻击(第 4.1.1 节论述).

2) 劫持控制流的防御

劫持控制流的防御,即如何防范内核劫持应用程序控制流,我们已在第 4.2 节做了详细的说明.应用程序在执行过程中,其上下文始终被 FortVisor 保护,无法被内核修改.因而,内核无法劫持应用程序控制流.

3) 文件 I/O 攻击的防御

文件 I/O 攻击的防御,即如何防范内核攻击应用程序文件 I/O,我们已在第 4.2 节做了详细的说明.当应用程序调用文件相关系统调用进行文件存储时,FortVisor 在不可信内核中构建可信文件数据流来传输文件数据,保证内核始终无法破坏应用程序文件的私密性和完整性.

## 8 实　验

### 8.1 安全测试实验

在安全测试实验中,我们针对攻击模型中(第 1 节)的每种攻击行为构造了具体的攻击实例,测试 AppFort 能否防范这些攻击.

- 攻击实例 1(直接访问攻击)

内核加载了一个恶意内核模块,该模块试图在地址空间中直接读取应用程序的数据.该攻击在我们的测试中无法实现.如第 6.4 节所述,AppFort 在将内核模块代码映射为可执行之前,会对代码中的每条访存指令进行动态检查.如果模块代码中的访存指令没有预先添加前缀,AppFort 就直接拒绝将模块代码映射为可执行.

- 攻击实例 2(修改页表攻击)

内核加载了一个恶意内核模块,该模块试图修改页表,将应用程序的内存重映射到地址空间(0,4G).该攻击在我们的测试中无法实现,因为当内核向 FortVisor 发送相应页表修改请求时,该请求会被 FortVisor 验证.当 FortVisor 发现内核试图重映射应用程序的内存时,会直接拒绝该请求.

- 攻击实例 3(DMA 攻击)

内核加载了一个恶意内核模块,该模块向磁盘发送恶意的 I/O 指令,利用 DMA 操作访问应用程序的内存.该攻击在我们的测试中无法实现,因为 FortVisor 对内核所有的 I/O 操作进行验证.当 FortVisor 发现内核通过 DMA 操作恶意访问应用程序内存时,会直接拒绝该操作.

- 攻击实例 4(Iago 攻击)

内核加载了一个恶意内核模块,该模块对 *mmap*()系统调用进行 hook,并精心构造 *mmap*()的返回值,使得 *mmap*()分配的 VMA 与应用程序栈所在的 VMA 重叠.当应用程序修改 *mmap*()分配的地址空间时,会间接修改栈上的数据,破坏应用程序数据的完整性.该攻击在我们的测试中无法实现,因为 FortVisor 对所有从内核返回的应用程序 VMA 进行验证.当 FortVisor 检测到 *mmap*()分配的 VMA 与应用程序的其他 VMA 重叠时,直接向应用程序返回错误码.

- 攻击实例 5(劫持控制流)

内核加载了一个恶意内核模块,该模块将应用程序的信号处理程序指向恶意代码,并向应用程序发送一个信号,劫持应用程序控制流,触发恶意代码执行.该攻击在我们的测试中无法实现,因为应用程序的执行上下文始终被 FortVisor 保护.内核无法访问应用程序上下文,也无法修改信号处理中的控制流.

- 攻击实例 6(攻击文件 I/O)

内核加载了一个恶意内核模块,该模块向磁盘发送恶意的 I/O 指令,试图读取磁盘块上的应用程序文件数

© 中国科学院软件研究所 http://www.jos.org.cn

据.该攻击在我们的测试中无法实现,因为磁盘块上的文件数据被应用程序的 SID 标记;并且,FortVisor 基于 I/O 验证确保这些文件数据只能被传输到已注册的文件缓存页框和具有相同 SID 的应用程序页框.在这整个过程中,恶意模块始终无法访问这些文件数据.

### 8.2 性能测试实验

本文选择一系列的内核和应用程序测试用例,测试 AppFort 的性能开销.其中,Lmbench 测试集[7]用于测量各种内核操作的性能,Postmark[8]和 Dokuwiki[9]用于模拟现实中 I/O-intensive 的应用程序,SPEC CPU2006[10]用于模拟现实中 CPU-intensive 的应用程序.同时,本文将测试结果与现有工作进行了详细的对比.实验环境为 CPU Intel i7-3770(4 cores),内存 8GB,500G SATA 磁盘,操作系统 Linux-kernel-3.8.0,编译环境 gcc-4.6.

#### 8.2.1 Lmbench 测试结果

本文从 Lmbench 测试集中挑选出一系列测试用例,测试 AppFort 对内核中各种操作(包括系统调用、内存操作、信号处理、进程创建和进程切换)的性能影响.表 1 给出了 AppFort 的实验结果,并使用 Native Linux 的实验结果作为基准.AppFort 在每一个测试用例上,性能开销都十分小(小于 1.10x).AppFort 的性能开销主要来源于对内核代码中间接分支指令的插装.在内核执行过程中,虽然 FortVisor 需要多次截获并验证内核操作,但是 FortVisor 与内核之间的切换十分轻量,仅需要一条直接跳转指令.表 1 中也给出了现有工作 Virtual Ghost[3]和 Inktag[2]的开销.Virtual Ghost 的开销主要来源于对内核代码中每条访存指令的插装;Inktag 中的开销主要来源于频繁的特权层切换.AppFort 避免了这些明显影响性能的操作.与现有工作相比,AppFort 中每一个测试用例的开销都明显减小了.

**Table 1** Experimental result of Lmbench

**表 1** Lmbench 实验结果

| 测试用例 | Native Linux | AppFort | 性能开销 | Virtual Ghost | Inktag |
|---|---|---|---|---|---|
| null syscall | 0.036 3 | 0.039 6 | 1.09x | 3.90x | 55.8x |
| open/close | 0.790 1 | 0.842 1 | 1.07x | 4,83x | 7.95x |
| mmap | 4 440 | 4 475 | 1.00x | 4,70x | 9.94x |
| page fault | 0.178 8 | 0.182 1 | 1.02x | 1,15x | 7.50x |
| signal install | 0.091 3 | 0.099 8 | 1.09x | 3,24x | – |
| signal deliver | 0.565 5 | 0.603 2 | 1.07x | 1,61x | – |
| fork_exit | 159.10 | 168.65 | 1.06x | 4.40x | 5.74x |
| fork_exec | 507.88 | 530.43 | 1.05x | 4.20x | 3.04x |
| select | 3.210 2 | 3.423 9 | 1.07x | 3.40x | |
| ctxsw 2p/ok | 1.62 | 1.62 | 1.00x | – | 1.41x |

其次,为了测量文件 I/O 性能,本文在 Lmbench 测试集中选取文件系统测试用例,测量调用 *read*()系统调用从文件系统中读取不同大小文件的 I/O 带宽.图 5 给出了 AppFort 的性能开销(以 Native Linux 作为基准).

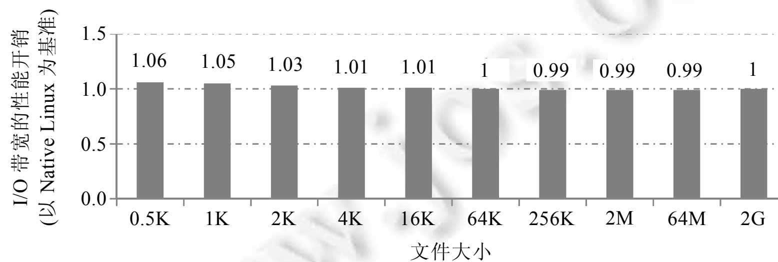


Fig.5 Experimental result of file system benchmark

图 5 文件系统测试用例实验结果

对于每一种大小的文件,AppFort 中 I/O 带宽的性能开销都十分小(小于 1.06x).

© 中国科学院软件研究所 http://www.jos.org.cn

8.2.2 Postmark 和 Dokuwiki 测试结果

Postmark 通过模拟实际邮件服务器的行为来测量文件 I/O 的性能.本实验中,Postmark 的配置环境如下:

- base files: 500;
- file size: 0.5KB~9.77KB;
- block size: 512;
- biases: 5;
- transactions: 500 000.

该配置环境与 Virtual Ghost 的配置环境一致.本文重复进行了 20 次测试,实验结果见表 2.从表 2 中可以看出,AppFort 几乎没有性能开销,与 Virtual Ghost(4.72x)相比明显提高了性能.该测试结果与 Lmbench 中测试用例(文件 I/O 带宽和 open/close)的测试结果是一致的.

**Table 2** Experimental result of Postmark

**表 2** Postmark 实验结果

| Native Linux (s) | Std.Dev | AppFort (s) | Std.Dev | 性能开销 | Virtual Ghost |
|---|---|---|---|---|---|
| 10.2 | 0.45 | 10.2 | 0.49 | 1.00x | 4.72x |

DokuWiki 在运行过程中映射大量的文件和匿名内存(anonymous memory),因而能够反映内核内存操作和文件 I/O 的开销.本实验中,DokuWiki 的配置环境与 Inktag 一致,并重复进行了 20 次测试,实验结果如表 3 所示.AppFort 同样几乎没有性能开销,与 Inktag(1.54x)相比明显提高了性能.

**Table 3** Experimental result of DokuWiki

**表 3** DokuWiki 实验结果

| Native Linux (req/s) | Std.Dev | AppFort (s) | Std.Dev | 性能开销 | Inktag |
|---|---|---|---|---|---|
| 14.7 | 0.54 | 14.8 | 0.68 | 1.01x | 1.54x |

8.2.3 SPEC CPU2006 测试结果

最后,图 6 给出了 SPEC CPU 2006 的测试结果,AppFort 几乎没有性能开销.事实上,SPEC CPU 2006 在 Inktag 和 Virtual Ghost 中的性能开销也比较小,因为 CPU-intensive 的应用程序很少进入内核执行.

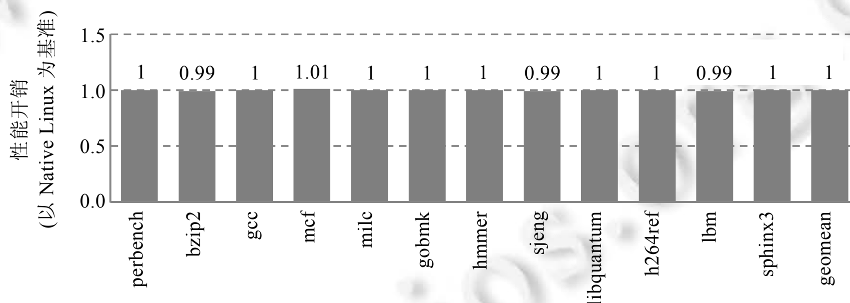


Fig. 6 Experimental result of SPEC CPU 2006

图 6 SPEC CPU 2006 实验结果

## 9 相关工作

在硬件方面,x86 支持(或将要支持)SMEP 和 SMAP 机制[11],用于禁止运行在内核态的软件执行或者访问用户态的数据代码.然而,这些机制只能防止直接访问攻击,不能防范第 1.1 节中提到的其他攻击.ARM 的 TrustZone 机制[12]在同一个物理 CPU 上创建两个虚拟执行环境(secure world 和 normal world),保证运行在 Secure World 中的应用程序与运行在 Secure World 中的内核之间完全隔离.此外,现有工作[13−17]通过修改 CPU

© 中国科学院软件研究所 http://www.jos.org.cn

体系构建,提出特殊的硬件机制来保证应用程序的安全.但这些新硬件机制很难应用到实际系统的保护中.

Flicker[18],TrustVisor[19]和 Memoir[20]基于 TPM 硬件或者虚拟化技术,保护应用程序中安全敏感的代码片段(codeblock).在此基础上,Fides[21]研究如何保护多个 codeblock 之间的安全交互.TP[22]和 DriverGuard[23]进一步研究如何保证这些 codeblock 与外部设备之间的安全 I/O.然而,这些工作并不能提供完全的应用程序保护,而且需要预先将应用程序分为安全敏感和安全不敏感两个部分.

Overshadow[1],Inktag[2]和 Virtual Ghost[3]提供完全的应用程序保护,包括所有数据代码的保护、控制流的保护和 I/O 保护(AppFort 也属于这一类别).然而,Overshadow 和 Inktag 依赖于虚拟化技术,其中频繁的特权层切换造成了较大的性能开销.Virtual Ghost 需要对内核代码中所有的访存指令进行插装,性能开销也比较大.

此外,许多现有工作关注内核本身的安全.虚拟机自省技术[24−26]通过底层的 hypervisor 对不可信内核的行为进行监控和验证,包括完整性验证、rootkit 检测等.其他工作保护内核的代码完整性[27]、控制流完整性[28]、动态数据完整性[29]和函数钩子的安全[30].

## 10 未来工作

如第 6 节所述,在我们目前的实现方案中,AppFort 只遵循简单的、预设定的保护策略.在我们的未来工作中,我们将提供更加灵活的、以用户为中心的保护策略制定方式.比如,我们将会像 Inktag 一样提供一个安全 shell,用户可以选择在安全 shell 下启动应用程序.只有在安全 shell 下启动的应用程序才会被 AppFort 保护,在其他情况下启动的应用程序将不会被保护;其次,用户可以在不同应用程序之间共享 SID,使得不同应用程序之间可以共享文件,实现更加灵活的文件访问控制策略.

此外,内核和应用程序的交互是复杂的,应用程序一方面要防范内核的攻击,另一方面又必须依赖内核提供系统服务.内核可能进行拒绝服务攻击,甚至提供一些错误的服务来实现攻击.然而,AppFort 的安全模型主要考虑如何保护应用程序本身的安全,包括内存安全、控制流完整性和 I/O 文件安全.即使内核实施拒绝服务攻击、或者提供不可信的服务,内核也无法破坏应用程序内存的私密性和完整性、劫持应用程序控制流或者攻击应用程序的 I/O 文件安全.因而,AppFort 的安全模型并没有将保护内核服务考虑在内.在本文研究过程中,我们对内核服务的保护问题已有了一些初步研究,我们将该问题留待今后工作中解决.

## 11 结 论

本文提出了一种在不可信内核中高效保护应用程序的新方法 AppFort.针对现有方法的高开销问题,AppFort 结合 x86 硬件机制、内核代码完整性保护和内核控制流完整性保护,在不可信内核同一特权层引入可信基 FortVisor,截获并验证内核的硬件操作和软件行为,从而保护应用程序的安全.实验结果表明,AppFort 与现有工作相比,在性能方面有了明显的提升.

**References**:

[1] 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]

[2] 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]

[3] 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]

[4] Shacham H. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In: Proc. of the ACM Conf. on Computer and Communications Security (CCS). 2007. 552−561. [doi: 10.1145/1315245.1315313]

© 中国科学院软件研究所 http://www.jos.org.cn

[5] Zhang C, Wei T, Chen Z, Duan L, Szekeres L, McCamant S, Zou W. Practical control flow integrity and randomization for binary executables. In: Proc. of the IEEE Symp. on Security and Privacy (S&P). 2013. 559−573. [doi: 10.1109/SP.2013.44]

[6] Niu B, Tan G. Monitor integrity protection with space efficiency and separate compilation. In: Proc. of the ACM Conf. on Computer and Communications Security (CCS). 2013. 199−210. [doi: 10.1145/2508859.2516649]

[7] Mcvoy LW, Staelin C. Lmbench: Portable tools for performance analysis. In: Proc. of the USENIX Annual Technical Conf. 1996. 23−23.

[8] Postmark. Email Delivery for Web Apps. 2013.

[9] Dokuwiki. 2015. http://www.dokuwiki.org

[10] Henning JL. SPEC CPU2006 benchmark descriptions. ACM SIGARCH Computer Architecture News, 2006,34(4):1−17. [doi: 10.1145/1186736.1186737]

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

[12] ARM Limited. ARM Security Technology: Building a Secure System Using Trustzone Technology. 2009.

[13] 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]

[14] 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]

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

[16] 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]

[17] 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]

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

[19] 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 (S&P). 2010. 143−158. [doi: 10.1109/SP.2010.17]

[20] Parno B, Lorch JR, Douceur JR, Mickens J, McCune JM. Memoir: Practical state continuity for protected modules. In: Proc. of the IEEE Symp. on Security and Privacy (S&P). 2011. 379−394. [doi: 10.1109/SP.2011.38]

[21] Strackx R, Piessens F. 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]

[22] Zhou Z, Gligor VD, Newsome J, McCune JM. Building verifiable trusted path on commodity x86 computers. In: Proc. of the IEEE Symp. on Security and Privacy (S&P). 2012. 616−630. [doi: 10.1109/SP.2012.42]

[23] Cheng Y, Ding X, Deng RH. DriverGuard: Virtualization-Based fine-grained protection on I/O flows. ACM Trans. on Information and System Security, 2013,16(2):Article 6. [doi: 10.1145/2505123]

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

[25] 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]

[26] 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]

[27] 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]


© 中国科学院软件研究所 http://www.jos.org.cn

[28] Criswell J, Dautenhahn N, Adve V. KCoFI: Complete ccontrol-flow integrity for commodity operating system kernels. In: Proc. of the IEEE Symp. on Security and Privacy. Oakland, 2014. 292−307. [doi: 10.1109/SP.2014.26]

[29] 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]

[30] 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]

**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.

www.jos.org.cn
www.jos.org.cn