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.

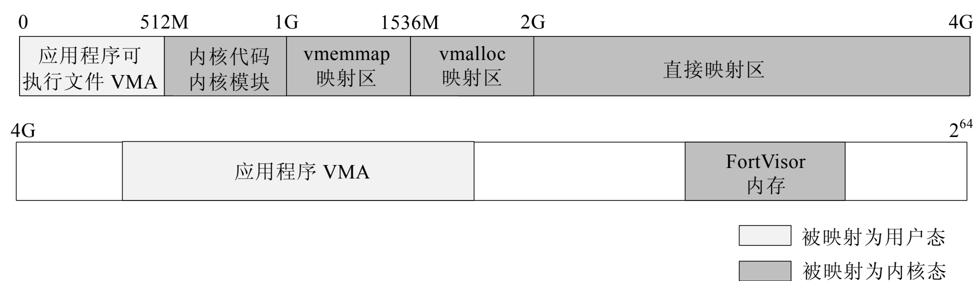

Fig.4 Address space layout after relocation

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).

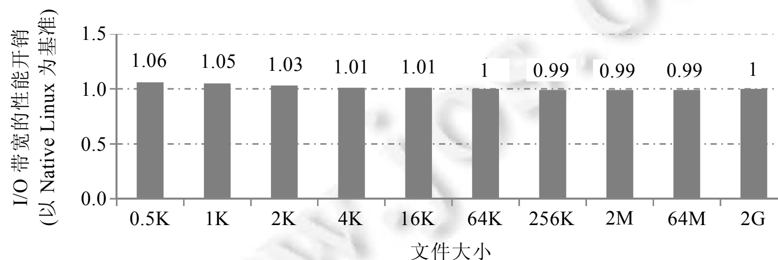

Fig.5 Experimental result of file system benchmark

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.

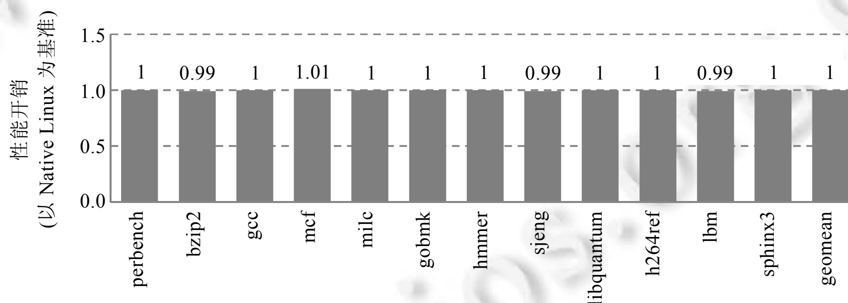

Fig. 6 Experimental result of SPEC CPU 2006

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

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