























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( $0.24 \times 6 = 1.44$ )大约是 AppISO( $0.015 \times 2 + 0.06 \times 2 = 0.15$ )的 9.6 倍.

本文进一步使用 `lmbench` 测试用例集<sup>[12]</sup>,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.

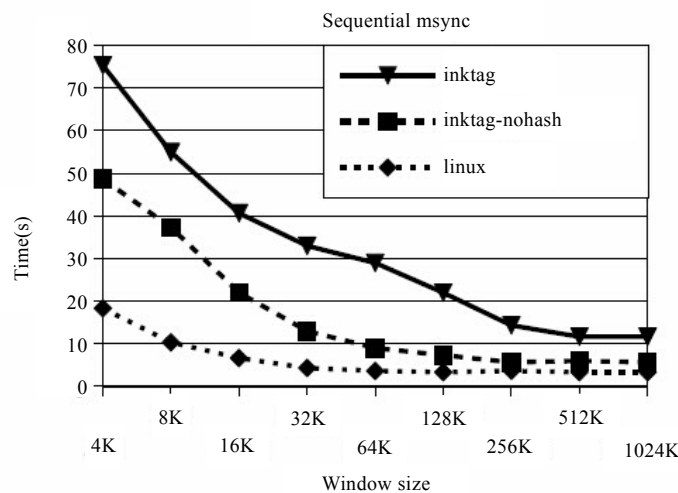


Fig.6 Results of msync in Inktag

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。

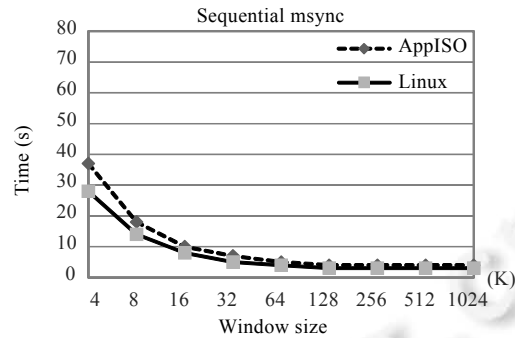


Fig.7 Results of msync in AppISO

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。

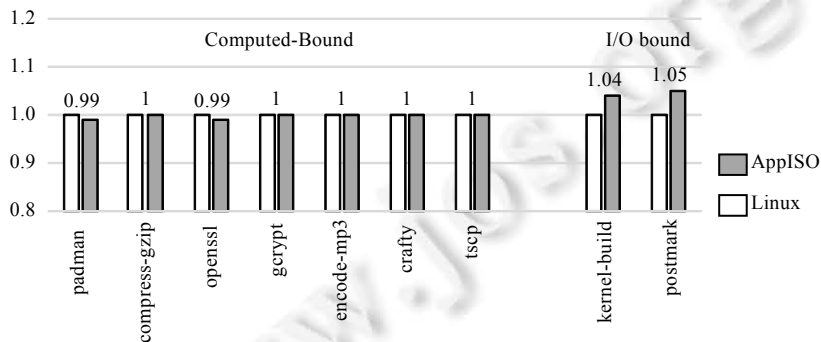


Fig.8 Results of application benchmarks

图 8 应用程序测试结果

## 10 相关工作

Overshadow<sup>[2]</sup>和 Inktag<sup>[1]</sup>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<sup>[13]</sup>,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<sup>[14]</sup>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<sup>[15]</sup>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<sup>[16-20]</sup>,实现应用程序隔离.Virtual Ghost<sup>[21]</sup>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<sup>[22]</sup>.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<sup>[23-25]</sup>,或者直接保护操作系统本身安全,比如 Hooksafe<sup>[26]</sup>保护操作系统中函数钩子,OSck<sup>[27]</sup>保护操作系统中的动态数据,Secvisor<sup>[28]</sup>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. Oversight: 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] Dvoskin 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, Dvoskin 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.