

基于本地虚拟化技术的隔离执行模型研究

温 研 王怀民

(国防科学技术大学计算机学院 长沙 410073)

摘 要 程序隔离执行是一种将被隔离代码的执行效果与其它应用隔离的安全机制. 但是目前的相关研究无法在 PC 平台下兼顾操作系统隔离与被隔离代码的可用性. 针对这个问题, 文中提出并实现了一种新的名为 SVEE(Safe Virtual Execution Environment)的隔离执行模型. SVEE 具有两个关键特性: 一是借助基于本地虚拟化技术的系统级虚拟机(SVEE VM)有效实现了非可信代码与宿主操作系统的隔离; 二是利用本地虚拟化技术实现了宿主机计算环境在 SVEE VM 内的重现, 保证了被隔离程序在 SVEE VM 中与在宿主操作系统内的执行效果的一致性. 因此, SVEE 在保护了宿主操作系统安全的同时, 兼顾了隔离执行代码的可用性. 测试证明, 对于计算密集型应用 SVEE 虚拟机的性能达到了本地性能的 91.23%~97.88%, 具有很好的可用性.

关键词 入侵隔离; 隔离执行; 虚拟执行环境; 安全; 虚拟机

中图法分类号 TP393

A Isolated Execution Model Based on Local Virtualization Technology

WEN Yan WANG Huai-Min

(School of Computer, National University of Defense Technology, Changsha 410073)

Abstract Isolation is a mechanism that has been applied to allow the isolated code running while shields the rest of the system from their effects. However, under the PC platforms, existing isolated execution approaches cannot achieve both the OS isolation and the functionality benefits of the isolated untrusted applications. To address this problem, this paper proposes a novel isolated execution model called Secure Virtual Execution Environment (SVEE). There are two key features in SVEE. Firstly, it fulfills the OS isolation by implementing a hosted virtual machine as the container of untrusted programs. Secondly, it can reuse the preinstalled applications of the host OS and faithfully reproduce the behavior of the isolated applications, as if they were running on the underlying host OS natively. As a result, SVEE guarantees security against potential malicious code without negating the functionality benefits provided by benign programs. Functional evaluation illustrates the effectiveness of the approach, while the performance evaluation shows that compute-intensive benchmarks run essentially at native speed on SVEE virtual machine, reaching 91.23%~97.88%.

Keywords intrusion isolation; isolated execution; virtual execution environment; security; virtual machine

1 引言

随着以网格为典型代表的大型分布式计算技术的不断发展,越来越多的 PC 用户通过共享其计算资源参与大型分布式计算任务^[1]. 对这些资源提供者而言,下载并执行来自资源使用者的软件已是一种常见的资源共享模式. 由于在计算资源提供者和使用者之间建立信任关系是非常困难的,因此这种常见的计算资源共享模式面临着安全与功能难以兼顾的挑战. 从资源提供者的角度来看,这些程序可能包含易受攻击或者不稳定的代码,为了保证自身宿主系统的安全性,资源提供者往往限制这些程序使用的资源,从而可能导致程序功能无法正确再现;而资源使用者往往更加关注这些程序的功能,希望能够直接使用提供者的宿主计算机系统.

为了增强宿主系统的安全性,资源提供者往往引入包括访问控制、病毒检测、沙箱等在内的各种主机型安全防护机制. 然而,访问控制机制与病毒检测技术均无法解决已授权程序的恶意行为或误操作对系统造成的破坏;沙箱技术通过限制运行在其中的代码所能够访问的系统资源来维护系统的安全性,但由于安全性与被隔离代码的功能往往是难以兼顾的,因此沙箱机制最突出的不足之处就在于很难选择合适的程序资源控制策略.

为了抵御程序对计算机系统造成的危害,同时尽可能地重现这些代码的功能,研究者引入了程序隔离执行机制. 依据计算机系统中隔离机制所处的层次,目前主要有基于单操作系统(Mono-OS)和基于虚拟机技术的两类隔离技术. 基于 Mono-OS 的隔离机制在操作系统内实现,因此每个隔离的运行容器能够共享相同的软件运行环境,但是,这种隔离机制能够被特权级(内核态)的恶意代码破坏. 此外,在 Mono-OS 内实现的隔离机制为了限制内核态代码,往往需要禁止许多特权操作的执行. 基于硬件抽象层虚拟机的隔离机制可以有效实现操作系统的隔离,但虚拟机中的软件环境需要重新安装部署,所以不能在 PC 平台下重现已有的软件运行环境.

综上所述,目前的程序隔离执行技术无法兼顾操作系统隔离与计算环境的重现. 不采用操作系统隔离机制就很难抵御来自内核态恶意代码的攻击,而不能重现用户计算机系统的计算环境则无法重用已安装部署的软件资源,甚至无法正确再现被隔离程序的功能.

为了解决同时满足操作系统隔离与计算环境重现这一难题,本文提出一种新的基于本地虚拟化技术的隔离模型——安全虚拟执行环境(Safe Virtual Execution Environment, SVEE),并在 Windows 平台下实现了 SVEE 的原型系统. SVEE 中隔离非可信代码的执行环境为基于本地虚拟化技术的系统级虚拟机(SVEE VM),有效实现了操作系统隔离. 与现有基于虚拟机的隔离机制不同,SVEE VM 中加载的操作系统不是重新安装部署的新的操作系统,而是在 SVEE VM 启动时创建的宿主操作系统的副本. 因此,SVEE VM 既重现了宿主操作系统计算环境,又无需重新安装部署新的操作系统,且无需限制特权操作的执行,从而完整再现了被隔离程序的功能.

本文第 2 节回顾与程序隔离执行相关的安全防护技术;第 3 节讨论 SVEE 的体系结构;第 4 节对 SVEE 进行安全性评估;第 5 节分析本地化虚拟机技术的实现;第 6 节描述 SVEE 的功能测试与性能测试结果;最后一节总结全文.

2 相关工作

沙箱技术:沙箱是一种按照安全策略限制进程行为的执行环境. 基于沙箱的安全机制监视程序行为并限制其执行违反安全策略的操作. Janus^[2]与 Chakravyuha^[3]基于内核调用拦截实现了沙箱机制. MAPbox^[4]通过提供基于不同程序行为的安全策略模版提高沙箱的可配制性与易用性. Prevelakis^[5]等提出了通过跟踪程序的文件访问操作创建安全策略的沙箱机制. SVE(Safe Virtual Execution)^[6]基于一种名为软件动态转换(software dynamic translation)的运行时程序代码修改技术实现沙箱机制. Systrace^[7]提出通过监视程序行为并通知用户,然后按照用户响应结果控制程序的方式实现沙箱机制.

沙箱机制最显著的不足就是无法找到一种便捷有效的方式为其生成合适的安全控制策略,即很难准确判断哪些程序行为不会危害系统安全. 因此,基于沙箱的安全机制往往由于采用过于严格的安全策略导致程序无法正常执行,系统的安全性与代码功能的完整性与兼容性通常在沙箱系统中难以兼顾.

现实情况是,绝大多数用户(尤其是 PC 用户)往往为了系统的可用性而放弃使用沙箱系统.

基于单操作系统(Mono-OS)的隔离机制:程序

隔离行技术最先应用于控制 Java applet^[8-9]的行. 与一般的应用程序相比,Java applet 访问的系统资源相对有限,因此针对此类的隔离技术的实现思路是在单独的主机上执行下载的 applet. 但是多数桌面应用程序需要访问类型更多的系统资源,尤其是文件系统的内容. 所以要在隔离的计算环境中正常执行此类应用,就需要重现该应用所需的所有系统资源,尤其是文件系统. 所以,如利用 Java applet 的隔离技术实现一般程序的隔离执行将带来非常大的系统部署开销. Liu^[10]等提出了第一个比较系统的基于单向隔离(one-way isolation)概念的程序隔离执行的解决方案,并讨论了在数据库与文件系统实现单向隔离的相关理论. 但是,他们并没有给出具体的实现方案. Alcatraz^[11]及其改进版本 SEE(Security Execution Environment)^[12]的关键特性是按照单向隔离的概念实现了计算环境的重现. Entropia Virtual Machine^[13]原理与 SEE 类似,也是利用 OS 的系统调用截获技术限制代码对系统资源的访问,从而达到维护系统安全性的目的. 所有这些未实现操作系统隔离的安全防护机制均不能有效防范内核态的恶意代码,并且需要借助访问控制策略才能限制 SEE 内的代码对敏感数据的访问,而且需要限制许多特权操作,如文件系统加载、进程控制、内核模块加载与卸载等,因此依赖于此类操作的应用程序往往无法在此类隔离环境中正常执行.

以上这些未实现操作系统隔离的程序隔离执行机制主要有 3 个方面的不足:(1)均不能有效防范内核态的恶意代码,如果恶意代码进入操作系统内核则可以绕过此类隔离机制直接访问系统资源;(2)需要借助访问控制策略才能限制代码对敏感数据的访问;(3)由于不能有效隔离内核态的非可信操作,所以这类隔离机制需要限制许多特权操作的执行,如文件系统加载、进程控制、内核模块加载与卸载等,因此依赖于这些操作的应用程序往往无法在此类隔离环境中正常执行.

基于虚拟机的隔离机制:研究者也提出了一些基于虚拟机技术的隔离机制^[14-18]. Covirt^[18]提出大部分应用程序应该在虚拟机中执行而不是只直接利用宿主环境执行. Figueiredo^[1], Santhanam^[19], Krsul^[20]等也分别提出了基于虚拟机技术实现的 Grid 环境下的隔离执行机制,. 但所有这些解决方案均没有在 PC 平台下实现计算环境的重现,无法实现对资源提供者的宿主系统资源的直接利用. Denali^[15-16]则是将分布式服务置于虚拟机中运行的

隔离机制,但它需要修改操作系统的代码. VMWare 的 ESX server^[21]能够在同一计算机系统中运行同一操作系统的多个副本,但是 ESX server 必须直接运行在硬件系统之上,这一部署要求使得它不能适用于已经预先安装了操作系统的 PC 平台;此外,运行在 ESX server 虚拟机上的所有操作系统都会有一定程度的性能下降,这就在一定程度上影响宿主计算环境的可用性. SVGrid^[22]基于 XEN^[23]实现了一个虚拟执行环境,但是由于 XEN 与 ESX server 一样都需要直接运行在硬件系统之上,而且需要修改操作系统,所以同样不适合 PC 平台. 开源虚拟机 QEMU^[24]虽然能够屏蔽 Guest OS 对宿主文件系统的修改,但是 Host OS 产生的修改仍将会导致 Guest OS 文件系统数据与磁盘数据的冲突,所以也未能实现计算环境的重现,而且 QEMU 中 Guest OS 的性能下降了近 10 倍. 基于 QEMU 的 KVM^[25](Kernel-based Virtual Machine)虽然性能有了大幅提升,但是除了不能实现计算环境的重现外,它还需要修改宿主操作系统,而且需要处理器支持 Intel VT^[26]或者 AMD-V^[27]技术.

总之,现有针对基于虚拟机的隔离机制的研究均集中于实现隔离系统的方式,没有涉及如何在 PC 平台下实现计算环境的重现. 因此,现有的基于虚拟机的隔离执行环境无法实现对宿主系统资源的直接重用.

3 SVEE 隔离模型研究

SVEE 的目标是在 PC 平台下同时实现操作系统的隔离与计算环境的重现,即通过隔离执行资源使用者提供的程序以维护宿主系统安全性的同时,尽可能不影响被隔离程序的功能. 因此,SVEE 的隔离机制需要满足以下约束.

约束 1. 操作系统隔离. 非可信代码必须运行在一个与宿主操作系统隔离的虚拟计算机系统中,这是抵御内核态的恶意代码攻击与防范针对硬件破坏的必要条件.

约束 2. 应用程序与操作系统透明. 现有操作系统与应用程序及其将被隔离的代码不需做任何修改即可直接部署该隔离机制,这一点在 PC 平台下尤其重要. 此约束包含 4 个子约束:

约束 2A. 无需修改现有操作系统与应用程序及其将被隔离的代码的源代码,因为通常 PC 上流行的应用程序与操作系统(例如 Windows)都没有

开放源代码。

约束 2B. 不能限制非可信代码在其执行环境内访问的资源与执行的特权操作,这是保证被隔离程序可用性的必要条件。

约束 2C. 尽可能将隔离机制对宿主执行环境造成的性能影响最小化,即在提高安全性的同时兼顾系统的可用性。

约束 2D. 无需重新安装现有操作系统. PC 用户中绝大部分不是计算机专业技术人员,所以 PC 上往往都预装有操作系统,所以在应用隔离执行技术时必须保证能够继续使用原有操作系统。

约束 3. 可配置的计算环境重现. 由于程序的正常执行通常依赖于计算环境,尤其是文件系统内容与操作系统配置等. 所以在 SVEE 内重现宿主操作系统的计算环境也是保证被隔离程序可用性的必要条件. 本约束可细化为

约束 3A. 计算环境的重现不应通过复制整个计算机的软硬件系统来实现,这样的部署开销通常不能被 PC 用户接受。

约束 3B. 被导出到隔离环境中的文件系统内

容是可配置的,被隔离程序只能访问这些资源,为了提高系统机密性,涉及敏感信息的数据不应在隔离环境中再现。

为了满足约束 1,SVEE 必须利用虚拟机监视器(Virtual Machine Monitor,VMM)作为隔离可信代码运行环境与非可信代码运行环境的中间层. 按照 Goldberg 的定义^[28],VMM 是能够为计算机系统创建高效、隔离的副本的软件. 这些副本即为虚拟机(Virtual Machine,VM),在虚拟机内虚拟处理器的指令集的一个子集能够直接在物理处理器上执行. Goldberg 定义了两种 VMM: Type I VMM 和 Type II VMM. Type I VMM 直接运行在计算机硬件系统上,负责调度和分配系统硬件资源,可以将其理解为一个实现了虚拟化机制的操作系统. 而 Type II VMM 则以一个应用程序的形式运行在已有的传统操作系统之上,而这个实际控制系统资源的操作系统被称为“Host OS”,运行在 Type II 虚拟机中的操作系统则被称为“Guest OS”. 基于这两种 VMM,SVEE 就有了相应的两种体系结构,如图 1 所示。

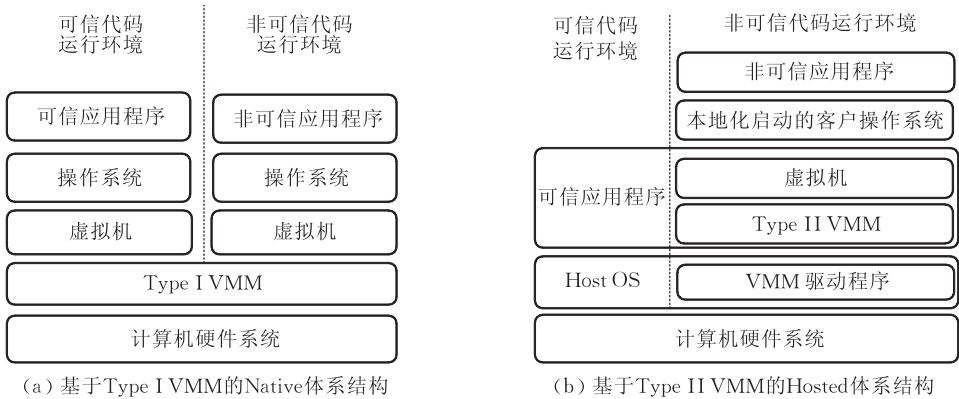


图 1 基于不同 VMM 的两种 SVEE 体系结构

除了约束 2C 和约束 2D,这两种体系结构都能够比较好地满足其它约束. 约束 2C 强调的是保证宿主代码运行环境的性能. 如图 1(a)所示,Native 体系结构中,所有操作系统都运行于虚拟机之上^[28],所以不可避免地导致可信运行环境的性能下降. 而基于 Type II VMM 的 Hosted 体系结构中的可信运行环境即为传统的操作系统,高效地直接运行于硬件系统之上. 在这一点上,基于 Type II VMM 的 Hosted 体系结构优于 Native 体系结构. 而对于约束 2D,在 Native 体系结构中,需要用 VMM 替换原有的操作系统,这往往对于 PC 用户来说是无法接受的. 与此相反,Hosted 体系结构则可以与已有操作系统共存。

此外,从软件开发角度,Native 体系结构将传统操作系统的硬件资源管理功能下移到 VMM 中,然而 PC 环境下硬件设备的多样化将会极大地增加 Type I VMM 开发的复杂性. PC 开放的体系结构导致 PC 平台下有类型繁多的硬件设备,而直接运行在硬件系统之上 Type I VMM 则需要管理这些设备,因此为这些设备编写相应的驱动程序将是工程浩大的开放工作. 与此不同,Type II VMM 可以直接利用操作系统提供的设备抽象接口,极大简化了 VMM 的开发,从而可以有效提高 VMM 的稳定性. 当然,与 Type I VMM 相比,Type II VMM 也有自身的劣势. 最明显的不足就是 Type II 虚拟机中虚拟 I/O 设备性能不及 Type I VMM. 但是随着

硬件虚拟化技术的普及、应用与提高^[29-30]以及各种虚拟 I/O 设备优化技术^[31]研究的不断发展,这种性能差距将逐渐缩小。

综上所述,由于 SVEE 主要针对的是 PC 环境,且 Hosed 体系结构在 PC 平台下具有显著优势,因此 SVEE 采用了基于 Type II VMM 的 Hosed 体系结构.在这种体系结构下,SVEE VMM 以 Type II VMM 的形式运行在宿主运行环境(即 Host OS)之上,并负责创建本地化启动的 SVEE 虚拟机作为执行非可信代码的运行环境.运行在本地化启动的 SVEE 虚拟机之上的 Guest OS 是 Host OS 的一个副本,因此非可信代码在 Host OS 上的行为得以精确再现,同时将其执行效果同宿主运行环境彻底隔离.第 5 节将描述基于 Hosted 体系结构的 SVEE 在 Windows 下的实现。

4 SVEE 的安全性分析

第 3 节已描述了 SVEE 的基于 Type II VMM 的 Hosted 体系结构,并讨论了其在 PC 平台下的优势.本节将进一步给出 SVEE 安全性的定量评估.为此,特做如下定义:

$$\begin{aligned} S &= \{p \mid p \text{ 为任一程序}\}, \\ S^U &= \{p \mid p \text{ 为任一非可信程序}\}, \\ S^T &= \{p \mid p \text{ 为任一可信程序}\} = S - S^U, \\ S^M &= \{p \mid p \in S^U \text{ 且包含恶意代码}\}, \\ S^I &= S^U - S^M, \\ S^V &= \{p \mid p \in S^T \text{ 且含有可被攻击的脆弱性代码}\}, \\ S^S &= S^T - S^V. \end{aligned}$$

VMM 和操作系统(OS)是两类特殊的程序,因为它们既是程序又是程序执行依赖的运行环境。

$$\begin{aligned} S_{env} &= \{p \mid p \text{ 为在 } env \text{ 中运行的任一程序}\}, \\ env &\in ENV = \end{aligned}$$

$$\{\text{OS, Local-Booted OS, Host OS, vOS}\},$$

其中,OS 表示传统的操作系统;如图 1(b)所示,Local-Booted OS 为在 SVEE 虚拟机中本地化启动的操作系统,而 Host OS 则表示 SVEE VMM 所依赖的宿主操作系统;vOS 代表运行在 Type I 虚拟机之上的操作系统,如图 1(a)所示。

$P(p)$, $p \in S_{env}$: 程序 p 破坏所在运行环境 env 安全性的概率。

$P_M(p)$, $p \in S_{env}$: 运行环境 env 中的程序 p 包含恶意代码的概率。

$P_V(p)$, $p \in S_{env}$: 程序包含有可导致破坏所在

运行环境 env 安全性的脆弱性代码的概率。

$Size(p)$: 程序 p 的源代码的行数,这是程序 p 规模的度量。

基于以上定义,可得如下公式:

$$S = S^T \cup S^U = (S^V \cup S^S) \cup (S^M \cup S^I) \quad (1)$$

$$P(p) = P_M(p) + P_V(p) \quad (2)$$

显然,随着运行环境 env 中运行的程序的增多, env 的安全性逐步降低,这个结论可用如下公式表示:

$$P(S'_{env}) = \sum_{p \in S'_{env}} P(p) < P(S''_{env}) = \sum_{p \in S''_{env}} P(p), S'_{env} \subset S''_{env} \quad (3)$$

此外,Venema 在“Secure Coder”中指出:在软件设计质量、代码复杂性与代码实现质量基本相当的前提下,平均每千行代码就隐藏有一个安全漏洞,即软件 p 的脆弱性大致与 $Size(p)$ 成正比,因此, $P_V(p)$ 可简化为下面的公式:

$$P_V(p) = \alpha \times \frac{Size(p)}{\sum_{p_i \in S_{env}} Size(p_i)}, p \in S_{env} \quad (4)$$

其中, α 为一经验常数。

对于目前的传统多任务操作系统 OS, $P(S_{OS})$ 可表示为下列公式:

$$\begin{aligned} P(S_{OS}) &= P(S_{OS}^T \cup S_{OS}^U) = P(S_{OS}^T) + P(S_{OS}^U) \\ &= P(S_{OS}^V) + P(S_{OS}^S) + P(S_{OS}^M) + P(S_{OS}^I) \\ &= P(S_{OS}^V) + P(S_{OS}^M) + P(S_{OS}^I) \end{aligned} \quad (5)$$

式(5)中 $P(S_{OS}^I)$ 最后未被计算是因为按照定义 S_{OS}^I 中的程序均可信且不包含脆弱性代码.基于这些基本结论,我们可以分别给出 SVEE 的 Hosted 体系结构的安全性分析。

对于 SVEE 中本地化启动的 Guest OS(Local-Booted OS)与宿主操作系统(Host OS)有

$$\begin{aligned} P(S_{\text{Local-Booted OS}}) &= P(S_{\text{Local-Booted OS}}^U) \\ &= P(S_{\text{Local-Booted OS}}^M) + P(S_{\text{Local-Booted OS}}^I) \end{aligned} \quad (6)$$

$$\begin{aligned} P(S_{\text{Host OS}}) &= P(S_{\text{Host OS}}^T) = P(S_{\text{Host OS}}^V) + P(S_{\text{Host OS}}^S) \\ &= P(S_{\text{Host OS}}^V) \end{aligned} \quad (7)$$

由于在作为宿主运行环境的 Host OS 中仅有 SVEE VMM、网卡设备驱动与操作系统的网络协议栈相关组件会与其它运行环境交换数据,因此可得如下公式:

$$S_{\text{Host OS}}^V \cong \{\text{SVEE VMM, Network Components}\},$$

$$S_{\text{Host OS}}^T \subset S_{OS} \text{ 且 } |S_{\text{Host OS}}^T| \ll |S_{OS}|,$$

$$Size(SVEE\ VMM) + Size(Network\ Components) \ll Size(OS). \quad (8)$$

基于式(3)、式(4)、式(7)和式(8),可得如下不等式:

$$P(S_{Host\ OS}) \cong P(SVEE\ VMM) + P(Network\ Components) \ll P(S_{OS}) \quad (9)$$

由于 SVEE VMM 以 Type II VMM 的形式实现,软件规模远远小于传统操作系统,所以即便是 $|S_{SVEE\ VMM}| = |S_{OS}|$, SVEE VMM 也相对来说更可靠.

SVEE 关注的是 Host OS 的安全问题, SVEE 虚拟机中的被隔离的代码破坏 Host OS 的安全性的概率可定义如下:

$$P(S_{Local-Booted\ OS} | VMM | Host\ OS) + P(S_{Host\ OS}) = P(S_{Local-Booted\ OS}) \times P(VMM) \times P(Host\ OS) + P(S_{Host\ OS}) \quad (10)$$

式中, $P(S_{Local-Booted\ OS} | VMM | Host\ OS)$ 表示 $S_{Local-Booted\ OS}$ 中的程序同时攻破 Local-Booted OS, SVEE VMM 与 Host OS 的安全防护的概率. 基于式(3)、式(9)与 $|\{VMM\}| = |\{Host\ OS\}| = 1 \ll |S_{OS}|$, 可得

$$P(S_{Local-Booted\ OS} | VMM | Host\ OS) + P(S_{Host\ OS}) = P(S_{Local-Booted\ OS}) \times P(VMM) \times P(Host\ OS) + P(S_{Host\ OS}) \ll P(S_{OS}) \quad (11)$$

综上所述,不等式(11)说明 SVEE 显著提高了 Host OS 的安全性.

5 SVEE 的系统实现

由于目前 PC 平台下最普及的配置是微软的 Windows 操作系统与 Intel 的 x86 系列处理器. 因此, SVEE 首先选择在面向 x86 处理器的 Windows 平台下实现原型系统.

SVEE 针对 Windows 平台的 Hosted 体系结构如图 2 所示. SVEE 的核心组件是 SVEE 虚拟机 (SVEE VM), 即采用本地化启动技术的系统级虚拟机程序; 被隔离的程序就在由 SVEE VM 启动的 Local-Booted Windows 中运行; 所有可信程序则直接在 Host Windows 上运行. 判断程序是否可信的操作必须在启动 SVEE 前完成, 因为不可信的程序必须被隔离在 SVEE 虚拟机中执行. 而判断的方法则采用了基于程序签名技术的白名单机制, 虽然现在已经有了多种攻击签名技术的方法, 但是由于运

行 SVEE 的宿主计算环境是可信的, 因此利用程序签名技术判断程序是否可信在 SVEE 隔离模型中还是行之有效的.

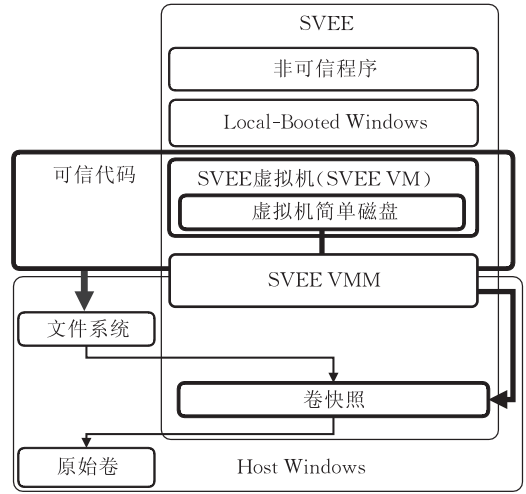


图 2 SVEE 在 Windows 下的体系结构

借助于 SVEE VMM 的系统级虚拟化能力, SVEE 有效实现了操作系统隔离与应用程序透明的特征. 同时, 通过本地化虚拟机启动技术, SVEE 实现了可配置的计算环境重现能力. 而被隔离的代码只能访问被导出到 SVEE VM 内的卷和文件, 而不是整个宿主机的文件系统.

针对不直接支持虚拟化的 x86 处理器, SVEE 采用了 Scan-Translation 技术实现高性能的 SVEE VMM. SVEE VMM 实现内存虚拟化和设备虚拟化的具体技术不在本文的论述范围内, 具体内容请参见文献[32].

对于本地化虚拟机启动技术而言, 关键问题是如何与宿主操作系统共享已安装在系统卷上的操作系统映像. 因为 SVEE VM 需要访问系统卷以启动 Local-Booted Windows, 而此时宿主操作系统也在修改系统卷, 而前者对数据的修改并没有使用宿主操作系统提供的访问接口, 后者的修改信息也无法及时被 SVEE VM 内的文件系统获取, 这就造成了文件系统数据与磁盘数据的冲突, 操作系统关键文件的不一致将会导致系统的崩溃.

为了解决这个问题, 本文引入了基于卷快照的虚拟简单磁盘 (Virtual Simple Disk) 技术. 卷快照是在创建时刻其对应的原始卷的一致性副本, 它提供了与文件系统标准卷相同的访问接口. 而虚拟简单磁盘则是卷快照 (必须包括系统卷的快照) 与虚拟分区表的集合, 它是将卷快照以存储设备的形式导出到 SVEE VM 的实现方式. 此外, 在将卷快照导出到 SVEE VM 前, 用户可以安全删除不允许在

Local-Booted Windows 中访问的目录、文件等敏感数据.

如上所述,SVEE 借助本地虚拟化同时实现了操作系统隔离与计算环境重现,下面将简要描述 SVEE VMM 的指令虚拟化技术,并详细介绍卷快照与虚拟简单磁盘的实现机制.

5.1 SVEE 虚拟机监视器

在传统的可虚拟化体系结构中,所有不能被虚拟机在处理器上直接执行的敏感指令(sensitive instruction)都需要产生自陷(trap),VMM 捕获自陷并依据虚拟机状态模拟相应敏感指令的执行.很容易验证这种技术能够满足 Popek 与 Goldberg 提出的 VMM 三约束^[28,33].但很多处理器(包括 Intel x86 处理器)的指令集都包括了不产生自陷的敏感指令(x86 处理器有 17 个这样的指令^[34]),所以实现此类处理器的虚拟化需要借助软件的方法模拟这些指令的执行语义.

SVEE VMM 大多数情况下会直接在宿主机上执行 Guest 指令,只有在极少情况下才会执行软件的指令模拟.在 x86 的 ring 3 优先级下,由于敏感指令极少执行,所以 Guest 指令执行的总开销不大.然而,SVEE VMM 是在 ring 1 优先级下执行 Guest 的 ring 0 代码,而 x86 处理器不允许在 ring 1 下直接执行 ring 0 代码,因此会产生大量的非敏感指令的自陷而导致 Guest OS 性能明显下降.此外,由于 IA32/AMD64 体系结构设计的缺陷,一些本应产生自陷的系统指令在 ring 1 下执行时没有产生自陷,从而导致这些指令在 ring 1 下执行语义发生变化.

为了解决这两个问题,我们提出了指令扫描与动态转换技术(Instruction Scan and Dynamic Translation, ISDT).该技术的实现依赖两个关键组件:指令扫描器(Code Scanner,CS)和指令转换器(Code Translator,CT).在执行 ring 0 代码前,CS 会执行指令扫描以发现将会产生自陷的非敏感指令.然后 CT 依据 SVEE VMM 状态执行指令转换,使得指令的执行跳转到等价模拟代码段处.此外,为了进一步提高性能,CS 还会分析和转换产生自陷的敏感指令以期消除该指令可能产生的其它自陷.

SVEE 基于 QEMU 的反汇编和指令转换引擎实现了 CS 和 CT. QEMU 是一个开源的基于动态指令转换技术的计算机系统模拟器,它的核心是负责运行时将目标指令集转换到等价的宿主指令集的动态指令转换器,转换后的指令存储在指令转换缓存中以提高执行效率.由于 SVEE 的宿主操作系

统是 Windows,所以我们将使用到的 QEMU 的代码移植到了 Windows 平台下.

5.2 卷快照

顾名思义,卷快照是在快照创建时刻其对应的原始卷的一致性副本,一致性的实现是通过写时复制(Copy-on-Write, COW)技术保存差异数据来完成的.利用这种机制,SVEE VM 解决了与宿主操作系统间由于写操作导致的数据不一致问题.作为虚拟存储设备的实现基础,卷快照也必须支持写操作.此外,卷快照也必须能够由宿主操作系统的文件系统加载,也就是必须提供与文件系统标准卷一致的访问接口.此特性能够有效实现宿主操作系统内对卷快照内容的访问,从而支持快照导出到 SVEE VM 前的文件配置功能、系统卷内硬件配置信息的修改.

在 Windows 平台下,Windows XP 及之后的版本也提供了与卷快照功能类似的服务,即卷影复制服务(Volume Shadow Copy Service).但是 Windows 2000 下并没有这种服务,而且在 Windows 下,此服务不支持写操作,也不支持加载到本地 Windows 上.

与其它快照技术类似,SVEE 的卷快照也采用了 COW 机制,即在原始卷的数据被修改前复制该数据到特定的存储空间.本文称此存储区域为快照区域(Snapshots Area, SArea).

图 3 描述了在安装了卷快照驱动后的 Windows 设备驱动栈.如图 2 所示,卷快照驱动在 Windows 的设备对象栈中位于其对应的原始卷的顶端,因此它能够有效执行 COW 操作.该驱动创建了两类设备对象:一类是加载在原始卷设备对象之上负责执

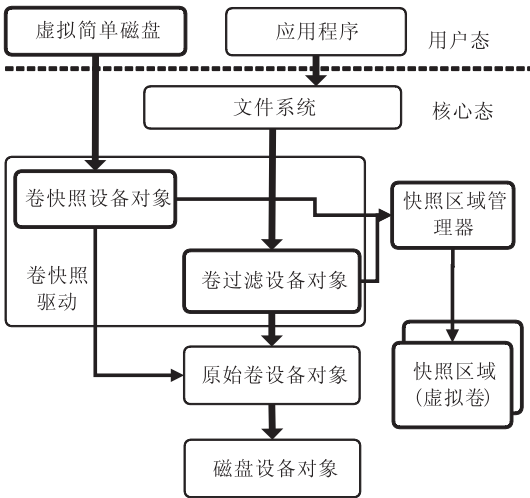


图 3 卷快照驱动体系结构

行 COW 操作的卷过滤设备对象,当其截获写原始卷的 I/O 请求包(I/O Request Package,IRP)时,将检查 IRP 的参数以判断是否将要修改的数据存在未被复制的部分,是,则阻塞该 IRP 直到复制相应数据到 SArea 的操作完成;一类是负责向文件系统提供标准卷的访问接口的卷快照设备对象,以便于被文件系统加载后用户访问快照数据,写快照的数据则被重定向到 SArea。

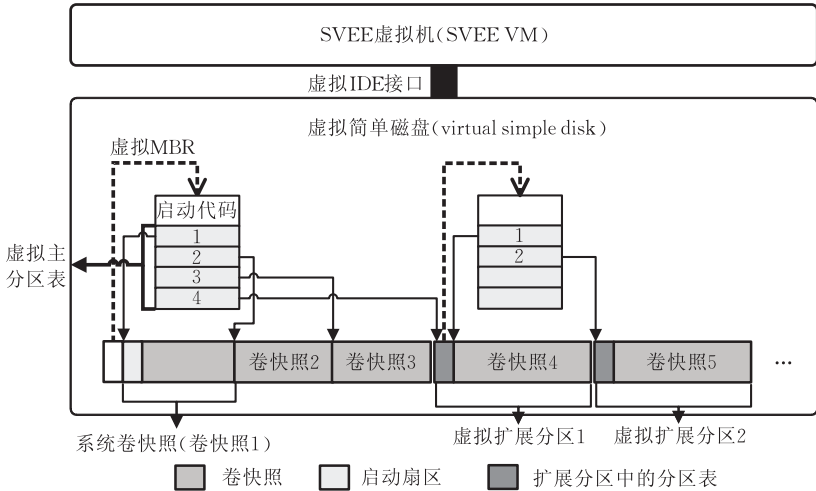


图 4 虚拟简单磁盘结构

从实现的角度,创建整个磁盘的快照比导出卷快照的方式更加直观.但是对于 SVEE 来说,组合卷快照的虚拟简单磁盘有更多优势:

(1) 便于配置导出的卷.如直接使用磁盘快照,则该磁盘内所有的卷均会被导出到 SVEE 虚拟机中,这往往违背了用户的安全性和可配置性的需求.而通过配置虚拟简单磁盘,只有用户明确需要导出的卷才能在 SVEE 虚拟机中访问;

(2) 卷格式透明.目前广泛应用的操作系统均支持多种卷格式,如单分区卷与多分区卷等,多分区卷包括镜像卷 (mirror volume)、RAID-0 卷和 RAID-5 卷等.所以如果直接导出磁盘快照,那么如该磁盘内含有分区卷,则也必须导出该卷依赖的所有磁盘.但是以卷快照为导出的基本单位则避免了这个问题。

(3) 便于操作快照数据.卷是文件系统加载的基本单位.通过加载卷快照,用户可以方便直观地访问快照数据。

6 系统测试

本节将描述 SVEE 的功能性测试与性能测试

5.3 基于卷快照的虚拟简单磁盘

对于系统级虚拟机而言,虚拟存储设备为物理磁盘,因此 SVEE 利用虚拟简单磁盘组织并导入卷快照的内容。

虚拟简单磁盘是卷快照、主分区表与扩展分区中的分区表的集合.在虚拟磁盘中,虚拟卷是与卷快照一一对应的数据结构,它保存了相应快照的详细参数.如图 4 所示。

的具体结果.测试用宿主机的操作系统为 Windows XP SP2,配置了 1G 字节 DDR2 333 内存与 1.60GHz Intel(R) Pentium(R) M 处理器。

6.1 功能测试

功能测试主要分为两个部分:(1) SVEE 的核心功能,即系统级虚拟机的本地化启动;(2) SVEE VM 作为系统级虚拟机的基本功能。

图 5 为 SVEE 虚拟机运行时的屏幕截图,Windows 桌面上标题为“Virtual Execution Environment”的窗口即为一个运行的 SVEE 虚拟机,该虚拟机内还运行着 Windows 资源管理器、Word 2007 和 MSN Messenger.由于 SVEE 虚拟机中的 Local-Booted Windows 屏幕分辨率为 800×600,因此它的桌面图标布局与宿主 Windows 的不同。

作为系统级虚拟机,SVEE VM 的虚拟化功能主要包括指令动态转换与硬件虚拟化等.指令动态转换的正确性测试由我们移植到 Windows 的 QEMU 的 test-i386 工具验证,test-i386 测试了所有 Intel x86 的用户态指令,包括 SSE、MMX 和 VM86 等指令集.测试结果显示所有这些指令在 SVEE VM 中的执行结果与在 Host Windows 下的结果完全一致.此外,我们通过在 Local-Booted Windows

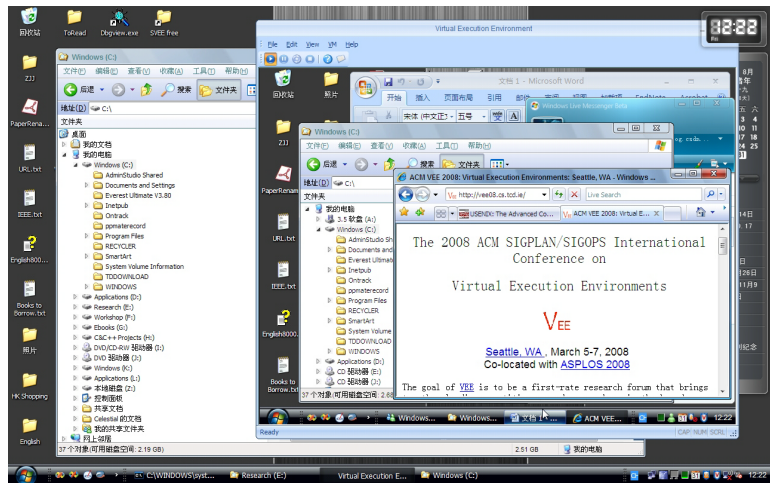


图 5 SVEE 屏幕截图

下运行测试软件 PassMark 验证 SVEE VM 中虚拟硬件,所有这些设备均工作正常,包括键盘、鼠标、IDE 磁盘、CD-ROM、网卡和显示卡等。

6.2 性能测试

性能测试包括两个部分:(1)SVEE VMM 与宿主机的性能比较;(2)快照技术对磁盘 I/O 和处理性能带来的负面影响。I/O 性能测试的工具是著名的开源 I/O 测试工具 IoMeter(2006 年 7 月发布的最新版本)。

(1) SVEE VMM 性能测试

除了敏感指令,SVEE 虚拟机中绝大部分指令可直接在宿主机处理器上执行,因此计算密集型应

用的性能与宿主机应该相差不大。但是涉及较多敏感指令的操作(如进程上下文切换、内存映射、I/O 访问、中断、系统调用等),SVEE 虚拟机会较多的性能开销。为了测试桌面应用程序的性能,本文使用了测试工具集 FreshDiagnose 7.86。由于桌面应用程序中较少执行敏感指令,因此性能应该接近于直接在宿主环境运行的性能,测试结果也证明了这一结论。如表 1 所示,测试应用性能下降百分比范围为 2.12%~8.77%,平均性能下降 4.78%。测试还与 VMware Workstation5.5.3 的性能做了比较,由于采用了类似的基于软件的动态指令转换技术来处理敏感指令,SVEE 与其性能相比,平均仅略低了不到 1%。

表 1 SVEE 虚拟机与宿主机性能比较

		Host Windows	SVEE	VMware 5.5.3	SVEE/Host Windows
CPU	MDIPS	5850	5726	5723	97.88
	MWIPS	3142	3048	3051	97.01
	Speed	1596	1589	1598	99.56
Multimedia	CPU Index	3130	3016	2884	96.36
Memory	Integer Assignment	80492	76422	77357	94.94
	Real Assignment	67520	62331	62425	92.31
	Integer Split	78695	72766	74198	92.47
	Real Split	78633	71733	74001	91.23

(2) 卷快照的性能测试

为了避免引入对虚拟化开销的分析,进而较为精确地比较卷快照驱动的开销,本节对卷快照驱动的性能测试是在宿主机环境下进行的,即直接在宿主环境加载卷快照并进行读写性能测试。因此,测试结果中并不会包括虚拟化技术引入的性能开销。

如 3.2 节所述,卷快照基于 COW 的实现方式增加了发送给原始卷的 IRP 的处理流程,因此创建了快照的原始卷的 I/O 性能将有所下降。为了定量分析快照带来的性能下降,我们做了如下定义:

(1)函数 $IoTime(Op, Target)$: Op 定义 I/O 操作类型,主要包括读写两种操作, $Target$ 表示 I/O 操作针对的卷,因此 $IoTime$ 实际定义了读写卷 $Target$ 所需的 I/O 时间;

(2)常量 $TestTime$:定义卷快照驱动,判断指定逻辑块是否已经进行了 COW 操作所需的时间;

(3)函数 $Dirty(BlockNo)$:定义块 $BlockNo$ 已经执行 COW 操作的概率。

基于以上定义,我们可以用如下公式计算具有快照的原始卷的 I/O 时间(OV 表示原始卷,SA 表

示快照区域,VS 表示卷快照).

$$IoTime(Read,OV)=TestTime+IoTime(Read,OV) \tag{12}$$

$$IoTime(Write,OV)=TestTime+P(Block\ No)\times\\(IoTime(Read,OV)+IoTime(Write,SA))+\\IoTime(Write,OV) \tag{13}$$

此外,可以用如下公式计算卷快照的 I/O 时间.

$$IoTime(Op,VS)=\\TestTime+IoTime(Op,OV\ or\ SA) \tag{14}$$

由于 $TestTime$ 主要包括两步计算操作和至多两次访存操作:首先执行右移操作计算读写地址所在逻辑块的块号,接着进行一次比特位比较操作(一般使用异或运算),这两步运算均可在一个指令周期

内完成.因此,一般情况下, $TestTime$ 相对 I/O 读写操作所需时间可以忽略不计.因此,通过上述公式,我们可以知道 COW 技术带来的读原始卷和读写快照的开销可以忽略不计.事实上,通过测试也证明这三类操作的 I/O 性能下降不到 1%.

但是对于写原始卷的操作,由式(15)可知,最大开销为增加一次写操作和一次读操作.由于 COW 算法对同一块只会执行一次写时复制操作,所以对已经执行了 COW 操作的块将不再会有此开销.

我们测试了卷快照驱动带来的处理器开销.测试所执行的 I/O 操作中,写操作占 23%,其余为读操作,I/O 地址仍为均匀分布.如表 2 所示,处理性能开销小于 3%.

表 2 卷快照 CPU 性能开销测试

	CPU Utilization/%	User Time/%	Privileged Time/%	DPC Time/%	Interrupt Time/%	Interrupts per Second
无快照的原始卷	2.785	0.0333	2.755	0.004	2.173	223.421
有快照的原始卷	4.184	0.0659	4.146	0.038	2.605	233.827
快照	4.891	0.0012	4.910	0.036	3.660	235.783

7 结束语

本文提出并实现了基于本地虚拟化技术的安全虚拟执行环境 SVEE. SVEE 提供了与其它基于虚拟机的隔离机制相同的通用特性. SVEE 最大的优势在于它在 PC 平台下同时提供了操作系统隔离、计算环境重现的能力,兼顾了系统的安全性与非可信代码的可用性.

此外,部署 SVEE 不需要修改现有操作系统与相关应用程序.针对 SVEE 的功能测试展示了其本地虚拟化功能的有效性,平均性能下降只有 4.78%.采用卷快照技术带来的 I/O 性能开销小于 10%,处理器开销则小于 3%.

因此,SVEE 通过实现基于本地虚拟化技术的操作系统隔离与计算环境重现功能,兼顾了计算资源提供者的宿主系统的安全性与被隔离代码的可用性,为大型分布式计算的资源节点提供了一种有效的安全执行环境.

参 考 文 献

[1] Figueiredo R J, Dinda P A, Fortes J A. A case for grid computing on virtual machines//Proceedings of the 23th International Conference on Distributed Computing Systems (ICDCS' 03). Providence, Rhode Island USA, 2003; 550-559

[2] Goldberg I, Wagner D, Thomas R et al. A secure environment for untrusted helper applications: Confining the wily hacker//Proceedings of the 6th USENIX Security Symposium. San Jose, California, USA, 1996; 1-13

[3] Dan A, Mohindra A, Ramaswami R et al. ChakraVyuha (CV): A sandbox operating system environment for controlled execution of alien code. IBM T. J. Watson Research Center: Technical Report 20742, 1997

[4] Acharya A, Raje M. Mapbox: Using parameterized behavior classes to confine applications//Proceedings of the 9th USENIX Security Symposium. Denver, Colorado, USA, 2000; 1-18

[5] Prevelakis V, Spinellis D. Sandboxing applications//Proceedings of the USENIX Annual Technical Conference. Washington, D. C. , USA, 2001; 119-126

[6] Scott K, Davidson J. Safe virtual execution using software dynamic translation//Proceedings of the 18th Annual Computer Security Applications Conference (ACSAC' 02). Las Vegas, Nevada, USA, 2002; 209-218

[7] Provos N. Improving host security with system call policies//Proceedings of the 12th USENIX Security Symposium. Washington, D. C. , USA, 2003; 257-271

[8] Chiueh T, Sankaran H, Neogi A. Spout: A transparent distributed execution engine for Java applets//Proceedings of the 20th International Conference on Distributed Computing Systems (ICDCS'00). Taipei, Taiwan, 2000; 394-401

[9] Malkhi D, Reiter M K. Secure execution of java applets using a remote playground. IEEE Transactions on Software Engineering, 2000, 26(12): 1197-1209

[10] Liu P, Jajodia S, Mccollum C D. Intrusion confinement by isolation in information systems. Journal of Computer Security, 2000, 8(4): 243-279

[11] Liang Z, Venkatakrishnan V N, Sekar R. Isolated program

- execution: An application transparent approach for executing untrusted programs//Proceedings of the Annual Computer Security Applications Conference (ACSAC'03). Las Vegas, NV, USA, 2003; 182-191
- [12] Sun W, Liang Z, Sekar R et al. One-way isolation: An effective approach for realizing safe execution environments//Proceedings of the Network and Distributed Systems Security Symposium (NDSS'05). San Diego, California, USA, 2005; 1-18
- [13] Calder B, Chien A A, Wang J et al. The entropia virtual machine for desktop grids//Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments. Chicago, Illinois, USA, 2005; 186-196
- [14] Chen P M, Noble B D. When virtual is better than real//Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS'01). Bavarian Alps, Germany, 2001; 133-138
- [15] Whitaker A, Shaw M, Gribble S D. Denali: A scalable isolation kernel//Proceedings of the 10th ACM SIGOPS European Workshop. Saint-Emilion, France, 2002; 10-15
- [16] Whitaker A, Shaw M, Gribble S D. Denali: Lightweight virtual machines for distributed and networked applications. University of Washington; Technical Report 02-02-01, 2002
- [17] Whitaker A, Shaw M, Gribble S D. Scale and performance in the denali isolation kernel//Proceedings of the 5th Symposium on Operating Systems Design and Implementation (SPECIAL ISSUE: Virtual machines). Boston, USA, 2002; 36
- [18] Whitaker A, Cox R S, Shaw M et al. Constructing services with interposable virtual hardware//Proceedings of the 1st Symposium on Networked Systems Design and Implementation. San Francisco, California, USA, 2004; 13-26
- [19] Santhanam S, Elango P, Arpaci-Dusseau A et al. Deploying virtual machines as sandboxes for the grid//Proceedings of the 2nd Workshop on Real, Large Distributed Systems. San Francisco, CA, 2005; 7-12
- [20] Krsul I, Ganguly A, Zhang J et al. VMPlants: Providing and managing virtual machine execution environments for grid computing//Proceedings of the ACM/IEEE Supercomputing 2004 Conference (SC'04). Washington, D. C., USA, 2004; 7
- [21] Waldspurger C A. Memory resource management in VMware ESX server//Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI'02). Boston, USA, 2002; 181-194
- [22] Zhao X, Borders K, Prakash A. SVGrid: A secure virtual environment for untrusted grid applications//Proceedings of the ACM/IFIP/USENIX 6th International Middleware Conference. Grenoble, France, 2005; 1-6
- [23] Barham P, Dragovic B, Fraser K et al. Xen and the art of virtualization//Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP'03). New York, 2003; 164-177
- [24] Bellard F. QEMU, a fast and portable dynamic translator//Proceedings of the USENIX Annual Technical Conference (USENIX'05). Marriott Anaheim, USA, 2005; 41-46
- [25] Qumranet. KVM: Kernel-based Virtualization Driver, 2006
- [26] Uhlig R, Neiger G, Rodgers D et al. Intel virtualization technology. IEEE Computer, 2005, 38(5): 48-56
- [27] Amd. AMD64 Virtualization Codenamed "pacific" Technology: Secure Virtual Machine Architecture Reference Manual. 2005; 1-124
- [28] Goldberg R P. Architectural principles for virtual computer systems [Ph. D. dissertation]. Harvard University, Cambridge, MA, 1972
- [29] Neiger G, Santoni A, Leung F et al. Intel virtualization technology: Hardware support for efficient processor virtualization. Intel Technology Journal, 2006, 10(3): 167-177
- [30] Adams K, Agesen O. A comparison of software and hardware techniques for x86 virtualization//Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'06). Seattle, WA, USA, 2006; 2-13
- [31] Sugerman J, Venkitachalam G, Lim B H. Virtualizing I/O devices on VMware workstation's hosted virtual machine monitor//Proceedings of the 2001 USENIX Annual Technical Conference. Boston, 2001; 1-14
- [32] Wen Yan. The study on an isolated execution model based on local virtualization technology [Ph. D. dissertation]. Changsha: National University of Defense Technology, 2008 (in Chinese)
(温研. 基于本地虚拟化技术的隔离执行环境研究 [博士学位论文]. 长沙: 国防科技大学, 2008)
- [33] E S J, Nair R. Virtual Machines: Versatile Platforms for Systems and Processes. San Francisco, CA, USA; Morgan Kaufmann Publishers, 2005
- [34] Robin J S, Irvine C E. Analysis of the Intel Pentium's ability to support a secure virtual machine monitor//Proceedings of the 9th USENIX Security Symposium. Denver, Colorado, USA, 2000; 129-144



WEN Yan, born in 1979, Ph.D. candidate. His major research interests include virtualization technology, and information security.

WANG Huai-Min, born in 1962, professor, Ph. D. supervisor. His major research interests include distributed computing, computer networks, and information security.

Background

On PC platforms, users often download and execute freeware/shareware to benefit from the rich software resource on the Internet. In spite of the high risk of executing untrusted programs, a significant fraction of users seem to be willing to take this risk in order to benefit from the functionality offered by these code.

In order to defend against potential malicious code, some host-based mechanisms were introduced to enhance the host security, i. e. , access control, virus detection, and so on. But access control can be fooled by authorized but malicious users, masqueraders, and misfeasors. Although virus detection and similar technologies can be deployed to detect widely prevalent malicious codes, they are limited not only in theory but in practice. In theory it is undecidable whether an arbitrary program contains a computer virus, and in practice it is also very difficult to accurately analyze the polymorphic or metamorphic virus code. Sandboxing is another approach. However, the main drawback of sandboxing-based approaches is the difficulty in policy selection. Too often, sandboxing tools incorporate highly restrictive policies that preclude execution of most useful applications.

Isolated execution is a more promising approach to bound the damage caused by undetected or detected intrusions during their latencies without negating the functionality benefits of untrusted code. But on PC platforms, existing isolation approaches cannot achieve both the OS isolation and execution environment reproduction. The former is a prerequisite to make the system be immune to privileged (kernel-

mode) malicious code, and the latter is necessary for reusing the preinstalled software and reproducing the behavior of untrusted code because the behavior of an application is usually determined by the execution environment.

To address this problem, the authors propose and implement a new local-booted virtual machine, namely Safe Virtual Execution Environment (SVEE). The untrusted code container of SVEE is a hosted system virtual machine, which boots not from a newly installed OS image but just from the underlying preinstalled host OS. In other words, SVEE loads another instance of the host OS. In this local-booted OS, no privileged operations will be restricted. Hence, the accurate behavior reproduction of untrusted code is assured while the host OS, acting as the trusted applications' container, is shielded from the effects of these untrusted code. In this local-booted OS, all the preinstalled software, maybe in size of several GBytes, can be reused directly. Thus, SVEE achieves both OS isolation and the behavior reproduction of untrusted applications.

The focus of this paper is on the architecture and implementation of execution environment reproduction, viz. the local-booting technology. The authors introduce Volume Snapshot technology to solve the file system conflicts due to sharing volumes between host OS and SVEE. In addition, they settle the deadlock problem induced by the snapshot Copy-On-Write operations for the file system using global locks. They also implement the dynamic OS migration for local-booted virtual machine.