

西安电子科技大学

硕士学位论文



基于KVM的虚拟机自省系统设计与实现

作者姓名_____李永波_____

学校导师姓名、职称_____李金库 副教授_____

企业导师姓名、职称_____朱荣昌 研究员_____

申请学位类别_____工程硕士_____

学校代码 10701
分 类 号 TP302

学 号 1303121818
密 级 公开

西安电子科技大学

硕士学位论文

基于 KVM 的虚拟机自省系统设计与实现

作者姓名：李永波

领 域：计算机技术

学位类别：工程硕士

学校导师姓名、职称：李金库 副教授

企业导师姓名、职称：朱荣昌 研究员

学 院：计算机学院

提交日期：2015 年 12 月

Design and Implementation of Virtual Machine Introspection System based on KVM

A thesis submitted to
XIDIAN UNIVERSITY
in partial fulfillment of the requirements
for the degree of Master
in Computer Technology

By
Li Yong Bo
Supervisor: Li Jin Ku Associate Professor
Zhu Rong Chang Research Fellow
December 2015

西安电子科技大学 学位论文独创性（或创新性）声明

秉承学校严谨的学风和优良的科学道德，本人声明所呈交的论文是我个人在导师指导下进行的研究工作及取得的研究成果。尽我所知，除了文中特别加以标注和致谢中所罗列的内容以外，论文中不包含其他人已经发表或撰写过的研究成果；也不包含为获得西安电子科技大学或其它教育机构的学位或证书而使用过的材料。与我一同工作的同事对本研究所做的任何贡献均已在论文中作了明确的说明并表示了谢意。

学位论文若有不实之处，本人承担一切法律责任。

本人签名： 李永波

日期： 2015.12.25

西安电子科技大学 关于论文使用授权的说明

本人完全了解西安电子科技大学有关保留和使用学位论文的规定，即：研究生在校攻读学位期间论文工作的知识产权属于西安电子科技大学。学校有权保留送交论文的复印件，允许查阅、借阅论文；学校可以公布论文的全部或部分内容，允许采用影印、缩印或其它复制手段保存论文。同时本人保证，结合学位论文研究成果完成的论文、发明专利等成果，署名为西安电子科技大学。

保密的学位论文在____年解密后适用本授权书。

本人签名： 李永波

导师签名： 唐永序

日期： 2015.12.25

日期： 2015.12.25

摘要

随着虚拟化技术的不断普及，用户可以自由的创建、迁移和共享各种类型的虚拟化操作系统与应用软件，享受这项技术所带来的便利。然而，由于计算机系统所具有的开放性，用户时刻都面临着各种各样的入侵攻击。虚拟机自省（**Virtual Machine Introspection, VMI**）技术由于能够兼顾高能见度与良好隔离性，为用户虚拟机系统提供更好的安全保障，得到广泛的研究和关注。

论文以 **KVM** 虚拟机管理器为平台，设计并实现了一种虚拟机自省系统。该系统克服了 **VMI** 中的“语义鸿沟”障碍，对 **KVM** 平台上虚拟机系统中的运行状态进行“盒外”的语义还原。它将 **VMI** 视图（“盒外”）中看到的物理页面、寄存器、设备等虚拟机系统抽象信息在“盒外”还原为虚拟机系统内部视图（“盒内”）所能看到的进程、内核模块等语义对象，以及虚拟机系统内部的系统调用、中断等语义事件，以对虚拟机系统进行有效监控。为此，需要在 **KVM** 中添加 **hooks** 以允许对内存、寄存器和设备的状态进行检查，并且允许介入和解释特定的事件，比如中断、系统调用和设备/内存/寄存器状态的更改等。

论文基于 **KVM** 开发出一套完善的针对目标虚拟机系统的 **VMI** 基础框架库，以动态链接库的形式提供一套完整的 **VMI API** 接口函数。该 **VMI** 基础框架库既包含虚拟机系统静态内容的自省，比如物理页面、寄存器、硬盘设备、**I/O** 设备等，又包含虚拟机动态事件的自省，比如系统调用、中断、寄存器/内存页面/设备等状态的改变等。与现有方案相比，该 **VMI** 基础框架库涵盖的内容更加全面，可扩展性更好。测试结果验证了 **VMI** 基础框架库的有效性和高效性。

关键词：虚拟化技术， 虚拟化安全， 虚拟机自省， **KVM**

ABSTRACT

With the growing popularity of virtualization technology, users are free to create, transfer and share various types of virtualized operating systems and application software, and enjoy the convenience brought by this technology. However, users are faced with a variety of intrusion every minute for the openness of computer systems. Virtual machine introspection (VMI) technology can provide the user virtual machine system with better security for that it can achieve both high visibility and good barrier properties. As a result, VMI technology has been widely studied.

This thesis designs and implements a virtual machine introspection system based on KVM platform. This system overcomes the "semantic gap" obstacles in VMI system and reconstructs the internal semantic views (e.g., processes, and kernel, modules) of a VM nonintrusive from the outside. The VMI system can get the "inside view" of virtual machines from the "outside of the box". This view is the abstraction information of virtual machines, such as physical page, register, equipment. Then the VMI system can restore the "inside view" to semantic object of virtual machine. As a result, the VMI system can see the semantic objects of virtual machines, like process, kernel module, also the internal system calls, interrupts and other semantic event of the virtual machine system can be monitored effectively. To achieve this, hooks are needed in the KVM to allow the VMI system to get the state of memory, registers, and equipment from the outside of virtual machines. In the same way, the VMI system can intervene and interpret the specific events (e.g., interrupts, system calls, and change device/memory/register states) of virtual machines, thus the VMI system can achieve to monitor the virtual machine from the outside.

Based on KVM, this thesis develops a comprehensive VMI framework library for the virtual machine. In the VMI framework library, there are several VMI API interface functions, and these functions provide service in the form of dynamic link library. The VMI framework library can introspect both static content and dynamic events of the virtual machines. The static content includes physical page, register, hard disk device, I/O devices, etc.; the dynamic events are like system calls, interrupts, register / memory page / changes in equipment status. Compared with the existing project, the VMI framework library are more comprehensive and has better scalability. According to a series of tests, the

KVM-based VMI system is effective and efficient in monitoring the virtual machine.

Keywords: Virtualization, Virtualization Security, VMI, KVM

插图索引

图 2.1 全虚拟化架构.....	8
图 2.2 半虚拟化架构.....	9
图 2.3 采用 Intel VT-x 技术的硬件辅助虚拟化架构	9
图 2.4 操作系统级虚拟化.....	10
图 2.5 Xen 组件结构	13
图 2.6 Xen 软件的体系结构	14
图 2.7 KVM 基本架构	15
图 2.8 基于主机代理的虚拟机自省技术.....	17
图 2.9 基于主机内核驱动的虚拟机自省技术.....	18
图 2.10 基于陷入、断点和回滚方式的虚拟机自省技术.....	19
图 3.1 基于 KVM 的虚拟机自省系统的总体架构	22
图 3.2 进程双向循环链表.....	23
图 3.3 进程监控流程.....	24
图 3.4 基于中断的系统调用监控.....	26
图 3.5 基于 SYSENTER/SYSEXIT 的系统调用监控.....	27
图 3.6 基于 SYSCALL/SYSRET 的系统调用监控.....	28
图 3.7 内存页面监控.....	29
图 5.1 进程监控.....	42
图 5.2 内核模块监控.....	43
图 5.3 39 号系统调用监控.....	43
图 5.4 所有系统调用监控.....	44
图 5.5 RAX 寄存器特殊值监控	44
图 5.6 RAX 寄存器跟踪	44
图 5.7 内存监控.....	45
图 5.8 中断监控.....	45
图 5.9 动态磁盘监控.....	45
图 5.10 设备 I/O 监控	46

表格索引

表 5.1 LMBench 对特定系统调用的性能测试	46
表 5.2 LMBench 对所有系统调用监控的性能测试	47
表 5.3 LMBench 对寄存器特定值监控的性能测试	47
表 5.4 LMBench 对内存访问监控的性能测试	48
表 5.5 LMBench 对寄存器全部变化监控的性能测试	48
表 5.6 LMBench 对特定中断事件监控的性能测试	49
表 5.7 LMBench 对所有中断事件监控的性能测试	49
表 5.8 LMBench 对设备 I/O 事件监控的性能测试	50
表 5.9 LMBench 对动态磁盘监控的性能测试	51
表 5.10 UnixBench 对各监控功能的性能测试	51

缩略语对照表

缩略语	英文全称	中文对照
HIDS	Host-based Intrusion Detection System	基于主机的入侵检测系统
NIDS	Network-based Intrusion Detection System	基于网络的入侵检测系统
VMI	Virtual Machine Introspection	虚拟机自省
VMM	Virtual Machine Monitor	虚拟机监视器
KVM	Kernel-based Virtual Machine	基于内核的虚拟机
VM	Virtual Machine	虚拟机

目录

摘要	I
ABSTRACT	III
插图索引	V
表格索引	VII
缩略语对照表	IX
第一章 绪论	1
1.1 研究背景	1
1.2 研究现状	2
1.2.1 传统入侵检测系统	2
1.2.2 虚拟机自省	2
1.3 论文的主要工作	4
1.4 论文组织结构	4
1.5 本章小结	5
第二章 虚拟机自省相关技术研究	7
2.1 虚拟化简介	7
2.1.1 虚拟化定义	7
2.1.2 虚拟化技术分类	7
2.2 主流虚拟机监视器介绍分析	11
2.2.1 XEN	12
2.2.2 KVM	14
2.2.3 VMware	16
2.3 虚拟机自省技术	16
2.3.1 虚拟机自省技术概述	16
2.3.2 虚拟机自省技术的实现方式和难点	17
2.4 本章小结	19
第三章 基于 KVM 的虚拟机自省系统设计	21
3.1 系统设计目标	21
3.2 系统总体设计方案	21
3.3 各功能模块详细设计	22
3.3.1 静态监控	23
3.3.2 动态监控	23

3.4	本章小结	30
第四章	原型系统实现	31
4.1	VMI 库函数层实现.....	31
4.2	QEMU-KVM 源代码的修改	33
4.2.1	QEMU-KVM 源代码介绍	33
4.2.2	QEMU-KVM 源代码的修改	34
4.3	KVM 源代码的修改	35
4.3.1	KVM 源代码介绍.....	35
4.3.2	KVM 源代码的修改.....	38
4.4	本章小结	40
第五章	原型系统测试与分析	41
5.1	搭建系统测试环境	41
5.1.1	原型系统安装所需软件包.....	41
5.1.2	原型系统运行.....	41
5.2	原型系统测试结果	42
5.2.1	功能测试.....	42
5.2.2	性能损耗测试.....	46
5.2.3	系统测试结果分析.....	52
5.3	本章小结	52
第六章	总结与展望	53
6.1	论文工作总结	53
6.2	下一步工作展望	54
参考文献	55
致谢	59
作者简介	61

第一章 绪论

1.1 研究背景

近几十年来,随着计算机技术和互联网技术的发展,计算机系统的计算能力和存储能力越来越强大,从 2005 年亚马逊推出 Amazon Web Services (AWS)开始,互联网产业迎来了云计算浪潮。云计算技术^[1]通过合理整合和利用系统平台的巨大硬件资源,从而为用户提供计算、存储等服务,用户可以通过网络连接到云平台,即可以享受云平台提供的优质服务。自 2005 年起,国内外各大互联网企业先后构建自己的云平台并向用户提供服务。国外企业如亚马逊、Google、IBM、微软等均在部署起的云平台并向用户推广;国内企业也推出了各种云平台,如阿里巴巴推出的阿里云服务,奇虎 360 推出的云杀毒服务、云盘存储,华为推出的云服务等。云平台通过简单安全的形式为广大用户提供如计算、存储等服务,迅速在企业用户及个人用户间普及。

作为实现云计算的关键,虚拟化在云计算普及过程中具有不可替代的支撑作用。虚拟化技术通过对计算机系统硬件资源进行逻辑抽象,根据用户对服务的需求,实时对用户分配资源,实现了资源的合理充分利用,提高了资源利用率。虚拟化技术由于能够简化服务部署、提高运行与维护效率、降低管理复杂性和提升资源利用率,被认为是云计算的基石性技术,市场潜力巨大。来自调研机构 TechNavio 的分析师预测,全球服务器虚拟化市场在 2012 年至 2016 年期间的复合年增长率为 31.07%,增长势头强劲。随着云计算技术的普及,虚拟化市场必将随之增长,对虚拟化技术的研究也必将越来越得到研究人员的重视。

通过利用虚拟化技术,用户可以自由的创建、迁移和共享各种类型的虚拟化操作系统与应用程序,享受这项技术所带来的便利。然而,不容忽视的是,随着各种云计算服务的普及,其自身的安全问题也逐渐暴露出来,引起广大研究人员和用户的广泛关注。例如在 2011 年 3 月份,Google 邮箱发生大批的邮箱数据泄漏,总共有 15 万左右用户的 Gmail 中所有邮件和聊天记录被删除,并且还有一部分用户的发现帐户被重置。在 2011 年 4 月 21 日,Amazon 公司在 Virginia 州的云计算中心发生宕机事件,影响了 Quora、Reddit、Hootsuite 和 FourSquare 等服务。2014 年 8 月,苹果公司 iCloud 遭黑客攻击,造成某些用户照片、视频等隐私信息泄露。可以看出,现代计算机系统越来越开放,用户时时刻刻都要面着各种各样的入侵和攻击。因此,作为云安全的基础,对虚拟化安全的研究有着非常重要的意义。

1.2 研究现状

1.2.1 传统入侵检测系统

为了应对外来入侵攻击,业界相关研究人员已经提出多种针对虚拟机操作系统的入侵检测方案。传统的针对虚拟机入侵检测方案主要分为两种,一种是使入侵检测软件部署于虚拟机操作系统内部运行,称为基于主机的入侵检测系统(host-based intrusion detection system, HIDS);另一种则为使入侵检测系统部署于网络节点上(通常部署于网络出口处)运行,称为基于网络的入侵检测系统(network-based intrusion detection system, NIDS)。这两种入侵检测方案均有其优缺点。HIDS 由于运行在被监控的虚拟机操作系统环境下,它对于当前虚拟机内部发生的事情具有很好的“能见度”(visibility),通常很难被绕过(而使其失效);但缺点是由于入侵检测系统与虚拟机隔离不是很好,容易受到攻击。NIDS 情况正好相反,它完全置身于被监控的虚拟机之外,与虚拟机具有很好的隔离性,针对虚拟机的攻击通常不会影响到 NIDS 的安全性;但也正是由于它与被监控的虚拟机高度隔离(仅仅通过网络相互连接),NIDS 对于虚拟机内部发生的事情具有很差的“能见度”,容易被攻击者绕过而令其失效,使得被监控网络或系统开放(即 NIDS 的“失效开放性”)。

1.2.2 虚拟机自省

在通过对传统的虚拟机入侵检测系统进行对比研究后,斯坦福大学的 Garfinkel 等人于 2003 年提出一种新型虚拟机入侵检测技术—虚拟机自省(Virtual Machine Introspection, 简称为 VMI)^[2]。VMI 借助运行于虚拟机下层的虚拟机管理器 VMM (Virtual Machine Monitor) 实现对虚拟机的监控,能够兼具 HIDS 的高“能见度”和 NIDS 的良好隔离性。VMI 通常运行于虚拟机下面的 VMM 层或一个被 VMM 管理的受信任独立虚拟机上,由于它并没有运行在被监控的虚拟机内部,所以 VMM 机制较好的隔离了 VMI 系统和被监控虚拟机。同时,VMM 使 VMI 能够检查虚拟机操作系统中运行状态和事件,比如 VMI 可以直接检查被监控虚拟机的物理内存页面、寄存器、硬盘等硬件状态以及中断、内存访问等事件,并且基于先前的数据结构、操作系统及体系结构等知识推断系统软件状态,比如虚拟机内部的当前进程列表,所以 VMI 对于被监控的虚拟机具有较好的“能见度”。Garfinkel 等人基于 VMI 思想开发出针对虚拟机的入侵检测原型系统 Livewire,它底层使用了 VMware Workstation VMM,能够对 X86 Linux 客户机完成基本的入侵检测功能。这些入侵检测功能主要包括两种类型,一种类型是轮询检测,例如应用程序完整性检测、签名检测、原始套接字检测等,另一种类型是基于事件驱动的检测,当 VMM 发现硬件状态发生改变时而发生,例如对于某些敏感寄存器状态的监测等。

后续研究者基于 VMI 技术设计出多种针对虚拟机的监控方案。例如，乔治亚理工学院的 Payne 等人基于 Xen 虚拟机监视器设计开发出针对虚拟机监控的 VMI 库，称为 XenAccess^[3]。XenAccess 将虚拟内存自省和虚拟硬盘监视结合在一起，允许对应用程序安全进行监控，并可以高效的获得目标系统上的内存状态和硬盘活动内容。微软研究院的 Wang 等人针对资源隐藏（resource hiding）类型的恶意攻击（称为“ghostware”），提出了一种将“盒内”（inside-the-box）和“盒外”（outside-the-box）视图（view）进行比较发现差异，进而检测恶意攻击的方法，称为 Strider GhostBuster^[4]。Strider GhostBuster 能够高效、成功的发现资源隐藏型的 rootkits、Trojans 等恶意攻击（比如文件隐藏、进程隐藏、模块隐藏等）。乔治梅森大学的 Jiang 等人将视图比较的方法进一步演化，设计和实现了一种能够在多种 VMM 和客户机操作系统上运行的 VMI 系统—VMwatcher^[5]，并将 VMI 中的关键技术难题“语义鸿沟”（semantic gap）进行了详尽阐述。所谓“语义鸿沟”是指从虚拟机内部和外部两个视角观察虚拟机，所看到的视图存在差异。例如，从虚拟机内部看到的是进程、文件、内核模块等语义级别的对象，而在虚拟机外部（采用 VMI 方法）看到的只是内存页面、寄存器和硬盘块等抽象内容。为此，要采用 VMI 进行异常检测，必须将其所看到的外部视图进行转换（因为 VMI 位于虚拟机外部），转换成具有虚拟机内部特定语义含义的具体对象。而这种语义转换通常是借助于虚拟机系统事先定义的数据结构而进行解析的。VMwatcher 采用一种称为“guest view casting”的技术方法对虚拟机在外部进行语义重构，将“盒内”和“盒外”的视图进行“桥接”，采用视图比较的方式发现恶意攻击，并将其应用到“交叉”恶意软件防范系统中（比如，将“盒外”的视图进行语义重构后得到“盒内”的语义对象，并进而借助于“盒外”的防病毒软件进行检测）。Process out-grafting（POG）对虚拟机系统中的关键进程进行语义重构，并进而监视其行为。

以上的系统在进行语义重构时，均需要编写相关的程序，并逐一进行手工重构，效率较低。近期的研究成果主要集中在如何自动的获得重构代码上。例如，乔治亚理工学院开发的 Virtuoso 系统^[6]和德克萨斯大学达拉斯分校开发的 VMST 系统^[7]均借助于 QEMU 自动获得语义重构代码，将二进制代码重用技术成功应用到 VMI 领域。

当前，随着针对 X86 平台的硬件辅助虚拟化技术（如 Intel 公司推出的 VT 技术^[8]，AMD 公司推出的 AMD-V 技术^[9]等）的快速发展，KVM（Kernel-based Virtual Machine）虚拟机管理器^[10]由于效率高、简单易用并被成功集成到 Linux 内核^[11]中，已经成为学术研究和产业发展的关注点。如何在 KVM 环境下，开发出一种高效可靠的 VMI 系统（或工具套件）成为一个重要的研究和应用课题。

1.3 论文的主要工作

本文首先对近年来虚拟化技术及几种主流的 VMM 进行研究,然后针对现有虚拟化环境中的安全问题,对传统的虚拟机入侵检测系统进行对比研究。基于以上研究,本文在 KVM 虚拟化平台上设计并实现了一种虚拟机自省系统,克服了 VMI 系统中存在的“语义鸿沟”障碍,实现对 KVM 平台上虚拟机操作系统中静态信息和动态事件实时监控,并在虚拟机外部进行信息还原。其中静态信息包括虚拟机操作系统中的进程、内核模块等语义对象,动态事件包括虚拟机内部的系统调用、寄存器状态修改、内存访问、磁盘访问、中断、设备 I/O 事件等。针对静态信息还原,本系统主要通过内存地址翻译,读取虚拟机内存、寄存器状态以及设备状态,从而实现在虚拟机外部对虚拟机内部信息的语义还原;针对动态事件捕获,本系统主要通过 KVM 虚拟机监视器中添加 hooks 以允许 VMI 系统对内存、寄存器和设备的状态进行检查,并且允许 VMI 系统介入和解释虚拟机中特定的事件,比如中断、系统调用和设备/内存/寄存器状态的更改等,从而实现对虚拟机系统内动态事件在虚拟机外部实时捕获。

通过以上的研究,论文基于 KVM 开发出一套完善的针对目标虚拟机系统 Linux 的 VMI 基础框架库,以动态链接库的形式提供一套完善的 VMI API 接口函数。该 VMI 基础框架库既包含虚拟机系统静态内容的自省,比如物理页面、寄存器、硬盘设备、I/O 设备等,又包含虚拟机动态事件的自省,比如系统调用、中断、寄存器/内存页面/设备等状态的改变等。与现有方案相比,该 VMI 基础框架库涵盖的内容更加全面,可扩展性更好。

基于对虚拟机操作系统的静态信息还原、动态事件捕获以及 VMI 基础框架库,本文最终实现了基于 KVM 的虚拟机自省系统,并搭建虚拟化环境对整个自省系统的可行性和监控效率进行了验证和测试。针对各个监控模块的功能测试表明基于 KVM 的 VMI 系统全面有效的实现了对虚拟机的监控,LMbench^[12]和 Unixbench^[13]的性能测试结果表明,VMI 系统对虚拟机带来的性能损失较小,不影响系统的正常运行。

1.4 论文组织结构

论文的组织结构如下:

绪论。本章首先介绍了论文研究的相关背景,明确研究目标和重要意义,然后对本文相关技术的发展现状进行说明,最后总结本文的研究工作和实验结果,并说明论文的组织结构。

虚拟机自省相关技术研究。这一章主要对虚拟机自省系统实现过程中涉及的相关理论、技术知识和虚拟化系统中存在的安全性问题加以介绍,包括虚拟化技术、虚拟机管理器(VMM)、虚拟机自省(VMI)技术以及虚拟机自省思想实现过程中存在的

关键问题，如语义鸿沟等。

基于 KVM 的虚拟机自省系统设计。本章在对主流虚拟化环境和已有入侵检测方案研究的基础上，提出了基于 KVM 的虚拟机自省系统的监控方案，同时明确该方案设计目标。然后根据系统设计目标，完成 VMI 系统总体架构设计，给出了 VMI 系统的总体流程。最后根据具体 VMI 系统的各监控对象对总体架构进行模块划分，对各个模块进一步详细设计，明确系统详细目标。

原型系统实现。在本章主要实施基于 VMI 系统总体架构设计和各个子模块详细设计，完整实现 VMI 原型系统，并将原型系统实现过程中的关键技术进行说明，包括对 KVM 内核模块和 QEMU-KVM 中代码的修改，以及 VMI 库层包含的各个监控函数接口的编码实现。

性能测试与分析。在这一章中，我们首先介绍系统测试所需环境和测试软件，然后搭建系统测试环境，并在运行虚拟机自省系统前后分别使用测试工具进行系统测试，最后对测试结果进行系统性能分析。

总结和展望。总结本文的主要研究内容和收获，并在总结的基础上展望未来研究工作。

1.5 本章小结

本章首先引入了本文的研究背景，同时国内外关于虚拟化的相关研究及发展现状做了简要介绍；然后根据虚拟化技术发展过程中存在的安全问题，确立了虚拟机自省系统的课题研究目标，并对虚拟机自省技术做了简要介绍；最后描述了本文的研究内容和组织结构，引导读者更好的理解虚拟机自省系统的设计和实现过程。

第二章 虚拟机自省相关技术研究

2.1 虚拟化简介

2.1.1 虚拟化定义

虚拟化（Virtualization）技术^[14]由 IBM 公司首次提出，并将其应用大型机系统，从 System 370 系列大型机系统开始，虚拟化技术逐步普及。大型机系统通过使用虚拟机管理器（Virtual Machine Monitor, VMM）^[15]对系统物理资源进行抽象，从而虚拟出多台能够部署完整操作系统的虚拟机（Virtual Machine, VM）^[16]实例。近年来，随着云计算的发展和应用，虚拟化在相关产业界的重要意义愈来愈明确。

在计算机科学领域中，虚拟化通常指一种对计算机各种资源的管理技术，通过对各种实体资源，例如服务器、内存、存储及网络等，进行逻辑抽象，打破实体结构间的障碍，根据用户需求更加充分合理的分配利用计算机资源。

通过利用虚拟化技术管理使用物理资源，计算机系统可以动态共享全部资源，更加灵活充分的利用资源；同时，计算机系统可以根据不同需求，实时对调整虚拟资源分配方案而无需对物理资源进行重新配置，从而实现更高的资源可扩展性。

采用虚拟化技术可实现如下目标：

提高资源利用率。采用虚拟化技术，计算机系统可以动态共享全部资源，获得更高的资源利用率。

降低管理成本。采用虚拟化技术管理系统，可以减少某些物理设备，降低系统复杂性，提高系统自动化水平，从而降低系统管理成本。

使用灵活。虚拟化技术可以动态部署配置资源，从而可以对不同业务需求做出灵活的调整。

更高的可扩展性。计算机系统可以根据不同需求，实时调整虚拟资源分配方案而无需对物理资源进行重新配置，可扩展性高。

提高系统安全性。虚拟化环境具有的隔离性，从而可以对访问虚拟机中数据和服务的行为进行控制，提高系统安全性。

2.1.2 虚拟化技术分类

在相关研究人员的多年努力下，虚拟化技术已经非常成熟，可以满足不同领域需求，下面我们将对其进行分类研究。虚拟化技术通常主要分为以下几个大类：

平台虚拟化（Platform Virtualization），主要的虚拟化对象包括计算机及其操作系统。

资源虚拟化 (Resource Virtualization)，主要面向计算机系统资源实施虚拟化，包括物理内存、存储和网络资源等。

应用程序虚拟化 (Application Virtualization)，主要涉及仿真、模拟和解释等相关方面。

平台虚拟化

平台虚拟化技术，通过使用控制程序 (Control Program，也被称为 Virtual Machine Monitor 或 Hypervisor)，通过对计算平台的抽象，实现隐藏其具体的物理属性，从而给用户一个抽象完整的资源环境，用户可以在该环境中安装操作系统等软件，以一台虚拟机的形式为自己服务。平台虚拟化技术还可以进一步细分为全虚拟化技术、半虚拟化技术以及硬件辅助虚拟化技术^[17]：

全虚拟化 (Full Virtualization)

完全虚拟化技术一般需要对计算机系统的底层硬件进行完全模拟，例如 CPU 指令集、I/O 操作、中断、内存访问、时钟、外设等。在采用完全虚拟化技术的环境中，无需对 Guest OS 和其它软件做任何修改即可直接部署运行。其主要原理是由 VMM 在 Guest OS 或者其它软件与底层硬件之间对相关特权指令实施捕捉并进行处理，从而实现对底层硬件资源的访问和使用。在利用完全虚拟化技术的虚拟化环境中，Guest OS 可以选取各种标准的操作系统，并不需要进行任何修改。知名的全虚拟化产品有：QEMU^[18]、IBM CP/CMS、VirtualBox^[19]、KVM、Vmware Workstation 和 ESXi^[20]。全虚拟化架构如图 2.1 所示：

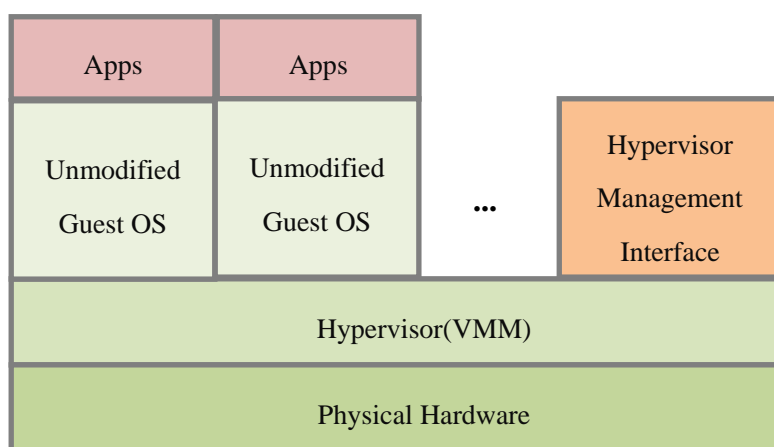


图 2.1 全虚拟化架构

半虚拟化 (Paravirtualization)

通常在半虚拟化环境中，Hypervisor 复杂管理物理资源并向上层提供资源调用接口。对于上层虚拟机来说，需要通过修改其自身部分代码，添加对底层资源的超级调用 (Hypercall) 函数，从而上层虚拟机可以通过 Hypercall 调用 Hypervisor 提供的接

口实现与 Hypervisor 通信。通常运行在半虚拟化环境中的客户机中已嵌入虚拟化代码，从而客户机与底层 Hypervisor 可以配合实现虚拟化环境，获得较好的虚拟化效果。通常半虚拟化技术能够高效的实现虚拟化环境，该环境中虚拟机性能可以非常接近物理机，另外各虚拟机之间相互隔离，互不影响。比较著名的 VMM 有 Denali^[21]、Xen^[22]。半虚拟化架构如图 2.2 所示：

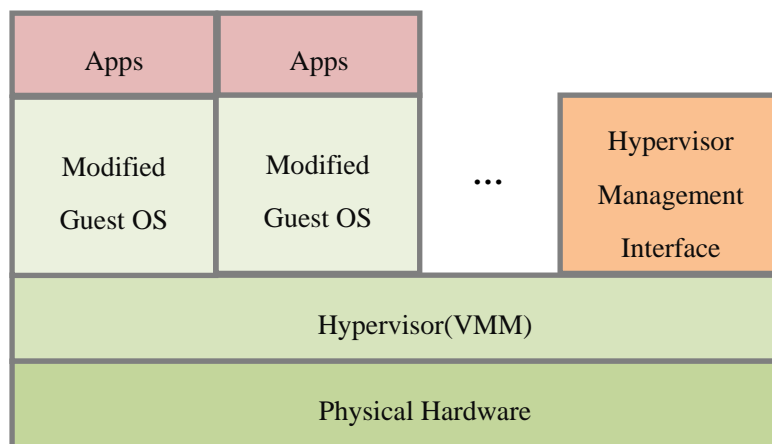


图 2.2 半虚拟化架构

硬件辅助虚拟化（Hardware-Assisted Virtualization）

硬件辅助虚拟化技术^[23]是指通过利用物理硬件（主要为中央处理器）的对虚拟化的支持实现虚拟化环境。最出名的例子莫过于 Intel 的 Intel Virtualization Technology（VT-x）和 AMD 的 AMD-V。硬件辅助虚拟化技术通过在 CPU 中添加 root 模式，从而使 VMM 能够以 root 模式进行部署，获取对底层硬件的管理特权，并且虚拟机中的特权指令、敏感指令可以自动在 hypervisor 上执行。图 2.3 所示为采用 Intel VT-x 技术的虚拟化环境架构：

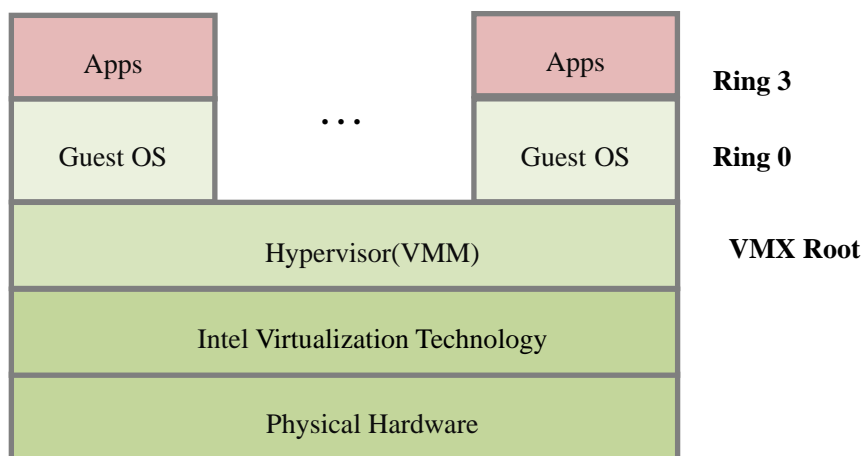


图 2.3 采用 Intel VT-x 技术的硬件辅助虚拟化架构

操作系统级虚拟化（Operating System Level Virtualization）

一般在传统的操作系统中，用户的全部进程均运行于同一个操作系统实例中，而当该操作系统中系统内核或应用程序出现缺陷时，其上运行的进程可能会遭受该缺陷的影响。针对该问题，操作系统级虚拟化技术^[24]被提出。在操作系统级虚拟化环境中无需部署 VMM，虚拟化环境是通过共享一个操作系统镜像实现，这个共享的操作系统镜像能够向上层分时提供强大的命名空间和资源隔离。通过使用轻量级的虚拟化技术，该共享的操作系统内核可以虚拟化出若干个操作系统实例，从而实现将不同的进程进行隔离，以致相互隔离的操作系统实例中的进程也完全隔离，进程间的运行互不影响。比较著名的操作系统级虚拟化产品有 Solaris Container^[25]，FreeBSD Jail^[26]和 OpenVZ^[27]等。操作系统级虚拟化环境的架构如图 2.4 所示：

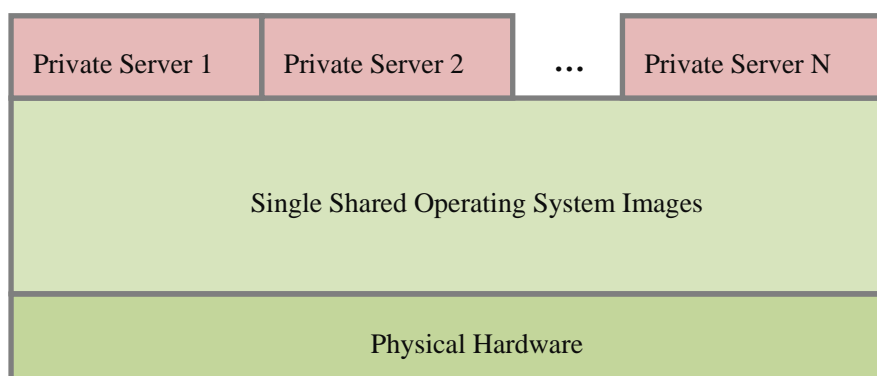


图 2.4 操作系统级虚拟化

资源虚拟化

资源虚拟化技术^[28]是指针对相关物理资源实施的虚拟化技术，该技术的应用对象如对物理内存、存储、网络资源等计算机资源。

内存虚拟化（Memory Virtualization）

内存虚拟化技术^[29]类似于现代操作系统中所提供的虚拟内存理论。在传统环境中，操作系统使用页表实现从虚拟内存到机器内存的映射，通常只需一次地址映射即可实现从虚拟内存到机器内存的访问。而在虚拟化环境中，内存虚拟化技术需要帮助 Guest OS 实现对物理内存资源的可视化，即实现对物理内存进行分区以及对虚拟机进行内存分配。可视化的目的是使 Guest OS 能够看到一个连续的地址空间，而这个连续的地址空间其实并不存在于真实的物理内存中，而是 Guest OS 通过使用内存虚拟化技术所建立的虚拟内存，该虚拟内存与真实的机器内存之间存在两次地址映射过程：第一次是将虚拟机线性地址映射为虚拟机物理地址，第二次是将虚拟机物理地址映射为主机物理地址的映射，两次地址映射过程分别由 Guest OS 和 VMM 维护实现。进一步，VMM 也应该支持 MMU 虚拟化，并且对 Guest OS 透明。

存储虚拟化 (Storage Virtualization)

存储虚拟化技术^[30]利用对信息存储系统的各个功能实施抽象、隐藏或隔离操作,实现对存储数据的管理和对应用程序与网络资源的管理分离开来,从而为系统提供简化、无缝的资源虚拟视图。利用存储虚拟化技术可以产生一个海量的存储池,用户通过虚拟化的存储资源就是一个巨大的存储池,用户无需了解其信息具体存储在何种介质中,也无需了解具体的数据存储过程,可以实现用户简便安全的存储数据。

网络虚拟化技术 (Network Virtualization)

网络虚拟化技术^[31]一般利用软件技术实现对物理网络资源进行抽象,实现对网络资源的充分利用。具体来说,网络虚拟化技术通过对网络中的交换机、网络端口、路由器和其他物理元素进行抽象和隔离,从而提高网络资源的利用率。网络虚拟化技术可以进一步细分为网络设备虚拟化技术、链路虚拟化技术以及虚拟网络技术。

应用程序虚拟化 (Application Virtualization)

由于用户需求的变化,虚拟化技术应用对象逐步由企业转向个人,根据个人用户的需求,应用程序虚拟化技术被提出,并成为近年来虚拟化技术发展的新方向。

应用程序虚拟化技术^[32]不再需要用户在本机安装其应用程序,转而将应用程序部署于云端,用户能够通过网络远程使用这些应用程序,从而解决了不同应用程序安装过程中兼容性问题以及同一应用程序不同版本的管理问题。在应用程序虚拟化环境中,通常用户实现了在线工作的需求,极大的提高了工作效率。

2.2 主流虚拟机监视器介绍分析

通过研究人员多年的开发,目前虚拟化技术已逐步成熟,业界已经出现了不少优秀的虚拟化产品。在这些优秀的产品中,虚拟机监视器 (Virtual Machine Monitor, VMM) 作为实现虚拟化环境的核心技术,有着非常重要的作用。目前业界已经有多款性能优秀的虚拟机监视器,根据各自特点可以将这些 VMM 分为两种类型:

Hypervisor VMM: 这种 VMM 能够部署在物理硬件 (Bare Metal) 上,能够完全实现对底层硬件的访问控制,并将该平台上各虚拟机隔离开来,可以使虚拟机的性能接近实体机,并对虚拟机系统的 I/O 操作进行了优化。这类虚拟机监视器也被称为 “Type-1”, 广泛应用于服务器虚拟化环境中。Microsoft Hyper-V、XEN 是一种经典的 “Type-1” 虚拟机监视器。

Hosted VMM: 这类 VMM 通常部署在主机操作系统上。因为 Hosted VMM 与底层硬件之间存在主机操作系统,所以这类 VMM 在性能方面不如 Hypervisor VMM,但是由于 Hosted VMM 的部署与使用比较简单,同时其功能也较为丰富,如对三维加速的支持等,这类 VMM 普遍被应用在桌面环境中,相对 Hypervisor VMM, HostVMM

被称为“Type-2”。VMware workstation、KVM 是两种典型的“Type-2”虚拟机监视器。

通常 VMM 具备如下三个特性：

隔离性 (Isolation)：由于 VMM 对计算机物理资源具有最高的访问控制权限，VMM 通过利用虚拟化技术，使各个虚拟机具有良好的隔离性，从而实现运行在不同虚拟机的软件互不干涉，提高系统安全性。

监控性 (Inspection)：VMM 对系统底层资源具有最高的访问权限，可以实时获取虚拟机的运行状态，包括 CPU 运行状态、寄存器状态、内存访问和 I/O 设备状态等信息，实现对虚拟机运行状态的实时监控。

介入能力 (Interposition)：VMM 需要介入客户机操作系统中的个别特殊操作（如介入特权指令的执行）。比如，在少量修改 VMM 后，当虚拟机上运行的代码试图修改某特定寄存器时，VMM 能够截获该事件，从而判断虚拟机上某些特定事件的安全性。

下面我们对几种主流的 VMM 进行介绍和分析。

2.2.1 XEN

Xen 由剑桥大学开发，并且对外开源，具体是一套可以部署于物理硬件上的虚拟机监视器，属于 Type-1 型。通过利用 Xen 对底层物理资源的抽象和管理，用户可以部署运行若干虚拟机操作系统（Guest OS）^[33]。

Xen 平台可以支持多种架构的处理器，如 x86、x86-64、Itanium、Power PC 和 ARM 等。当前在 Xen 平台上，用户可以安装使用 Linux、NetBSD、FreeBSD、Solaris、Windows 等多种虚拟机操作系统。

通常可以将 Xen 虚拟化监视器可以分为三大基本组件：Xen Hypervisor、Domain 0 和 Domain U^[34]。其中 Xen Hypervisor 一般直接部署物理硬件之上，通过对底层硬件资源的抽象和管理，向其上层运行的主机操作系统和客户机操作系统提供运行所需的各种资源，同时对各个虚拟机或者 Domain 之间进行隔离。在上层的这些域中，通常存在一个特权域（Domain 0），主机操作系统一般部署于 Domain 0 中，上层其他域称为 Domain U。由于 Domain 0 较 Domain U 对资源访问具有比较高的特权，所以可以使用 Domain 0 实现对 Domain U 的管理及服务。Xen 组件结构如图 2.5 所示：

Xen Hypervisor

Xen Hypervisor 是部署运行底层物理硬件和上层操作系统间的虚拟机监视软件。负责向部署在其上的各种类型的虚拟机操作系统提供底层物理资源的分配和调度，对 CPU

的调度及物理内存分配等。在 Xen 软件架构中，Xen Hypervisor 不仅仅是一套访问底

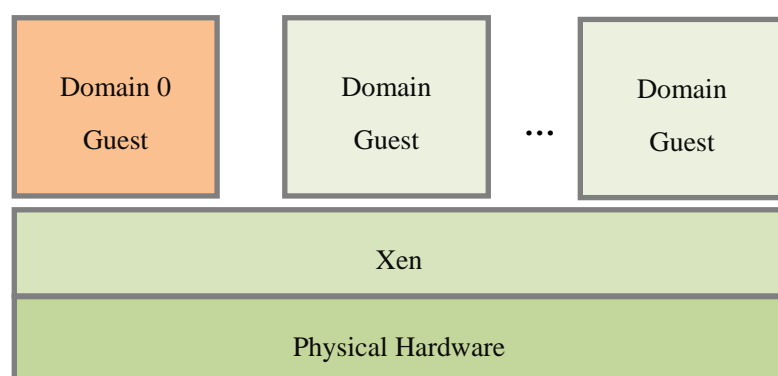


图 2.5 Xen 组件结构

层硬件的接口函数，同时还可以对虚拟机进行管理和控制，使不同虚拟机之间可以共享底层的物理资源。

Domain 0

在 Domain 0 中，我们通常可以部署运行一个修改过的操作系统。Domain 0 具有访问底层物理资源的特权，同时还能够和虚拟化环境中的其他虚拟机进行交互通信，在整个 Xen 软件架构中具有非常重要的作用。在部署 Xen 虚拟化环境时，我们均需首先安装 Domain 0，之后才可以部署运行 Domain U。Domain 0 通常在 Xen 软件架构负责系统管理员的任务，对 Domain 0 进行管理。

Domain U

在 Domain U 中，我们通常可以部署自己的客户虚拟机，但是 Domain U 中的虚拟机不能直接访问底层的物理硬件资源。部署于 Xen Hypervisor 上的全部半虚拟化客户虚拟机（Domain U PV Guests）均是经过修改的操作系统。这些操作系统通常包括基于 Linux 和基于 Unix 的操作系统。而全虚拟化虚拟机（Domain U HVM Guests）通常为 Windows 标准系统或者其他未经过修改源代码的操作系统。

为了实现虚拟机对底层硬件的访问，Xen VMM 为上层域给出了一个物理资源抽象层，包含上层虚拟机访问和管理虚拟硬件资源的 API。在 Domain 0 中，系统拥有可以直接访问底层物理资源的驱动程序，从而实现与 Xen Hypervisor 进行交互操作，并对整个虚拟化环境进行管理。Xen 软件的体系结构如图 2.6 所示：

Xen 具有以下特点：

开放性好：Xen 的软件代码开源，研究人员可以根据需求对其进行修改和移植。

平台支持广泛：Xen 虚拟化环境支持多种处理器架构和客户机操作系统。

支持动态迁移：支持动态迁移，并且动态迁移过程几乎没有宕机。

性能：Xen 平台上部署的虚拟化环境性能较好，其中在半虚拟化环境中，虚拟机性能接近物理机。

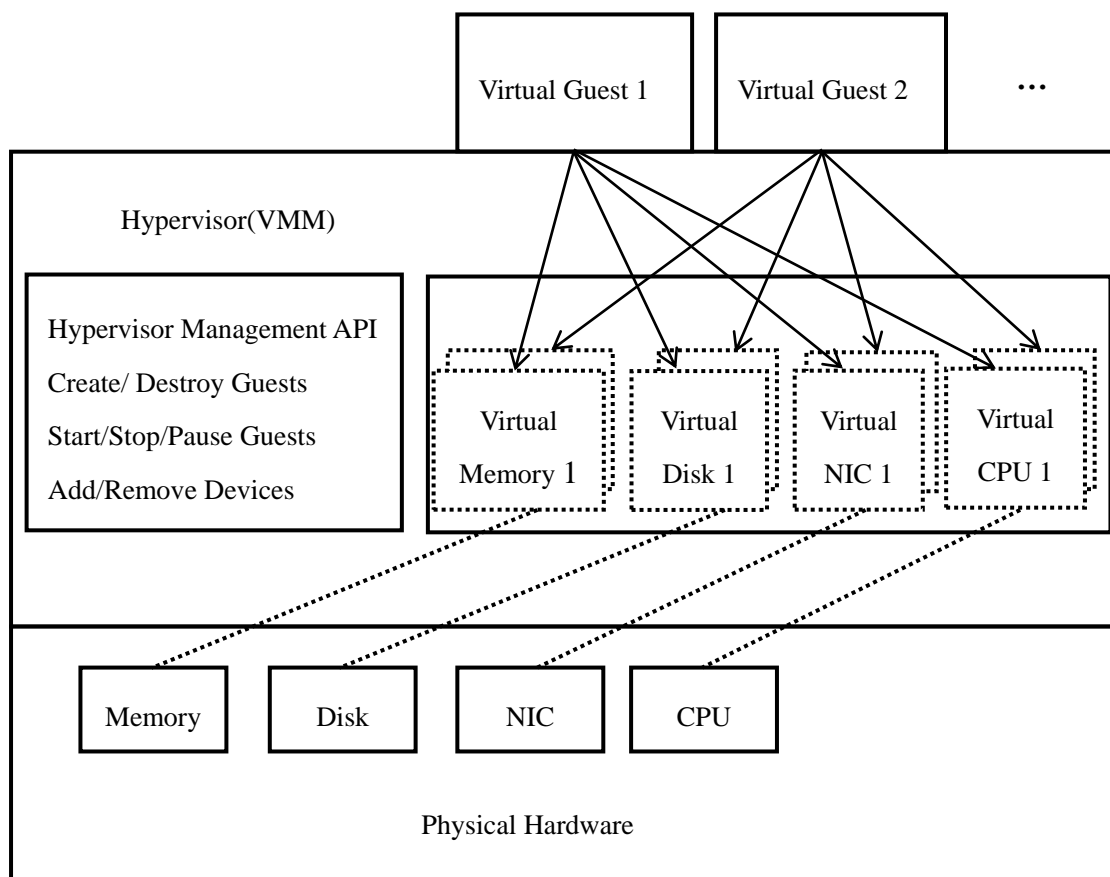


图 2.6 Xen 软件的体系结构

可操作性和可维护性：由于 XEN 与 Linux 内核整合不好，XEN 操作复杂，可维护性较差，维护成本高。

2.2.2 KVM

KVM 指基于 Linux 内核的虚拟机(Kernel-based Virtual Machine)，是一种利用硬件辅助虚拟化技术的开源虚拟机监视器^[35]。其中主要是一个可装载的内核模块，即 `kvm.ko`。该虚拟化内核模块是实现虚拟化的核心基础。根据不同的处理器架构，KVM

给出了不同的虚拟化模块，例如针对 Intel 公司的处理器提供 `kvm-intel.ko` 模块，而针对 AMD 处理器则提供 `kvm-amd.ko` 模块。自 Linux 2.6.20 之后 KVM 已经被集成在各个主要发行版本中，方便相关开发人员进行研究开发。

KVM 基本结构由两部分构成：KVM 内核模块和 QEMU 模拟器。

KVM 内核模块是 Linux 系统内核中的一个模块，用来实现虚拟机的创建、虚拟内存的分配、虚拟机中 VCPU 寄存器的读写以及 VCPU 的运行等工作。

QEMU 模拟器主要负责实现模拟虚拟机操作系统中的各个用户空间的组件，

同时向上层提供 I/O 设备的访问接口。KVM 的基本架构如图 2.7 所示：采用 KVM 虚拟化技术实现虚拟化环境，用户可以通过加载 KVM 内核模块从而将 Linux 系统内核变成一个系统管理程序，并被组织为 Linux 系统中标准的字符设备：/dev/kvm。而 QEMU 通过 KVM 所给出的 LibKVM 接口，通过调用 ioctl 实现创建、运行客户机。通过 KVM Driver 可以将 Linux 系统转变为 VMM，同时向 Linux 系统添加了新的运行模式，即客户模式。在客户模式中还包含有内核模式和用户模式，从而整个虚拟化换机中将存在三种执行模式：Linux 系统原有的内核模式和用户模式，以及新添加的客户模式。三种执行模式的具体任务如下：

客户模式：客户机的运行模式，在该模式下执行客户机操作系统中除 I/O 操作的其他代码。

用户模式：在该模式中运行客户机中的 I/O 代码，同时 QEMU 运行于该模式中。

内核模式：负责虚拟机操作系统在客户模式与其他模式之间的切换，同时处理客户机中的运行异常等。KVM 模块运行于该模式中。

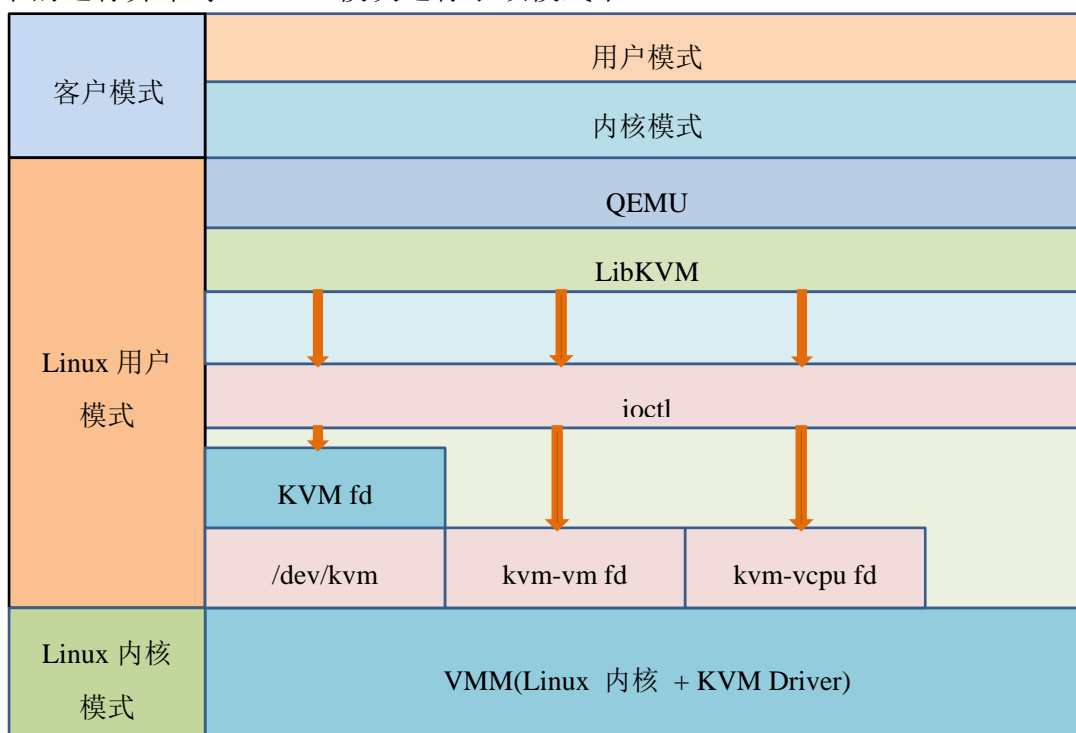


图 2.7 KVM 基本架构

在 KVM 虚拟化环境中，所有 Guest OS 均能看成系统中的一个进程，用户可利用 Linux 系统中的进程管理指令对客户机进行管理。KVM 的基本工作原理：客户机通常工作在客户模式，当产生异常或者访问底层硬件资源时，客户机利用 QEMU 通过 LibKVM 接口调用 KVM 所提供的 ioctl 从客户模式切换到内核模式。而 KVM Driver 在为客户机创建完虚拟内存及虚拟 CPU 后利用 VMLAUNCH 再从内核模式切换到客户

模式，继续运行 Guest OS，从而实现客户机对底层硬件资源的访问或者异常处理。另外 KVM 虚拟化环境采用全虚拟化技术，无需对虚拟机操作系统做任何修改。

KVM 具有以下特点：

开放性：KVM 代码开源，研究人员能根据需求进行修改移植。

系统兼容性：KVM 是 Linux 系统中的一个模块，可与 Linux 系统完美兼容。

可操作性：KVM 虚拟化环境使用 Linux 作为资源管理器，可操作性好。

动态迁移：KVM 现在已经支持动态迁移。

厂商支持：众多厂商均在推广 KVM 虚拟化技术。

2.2.3 VMware

VMware 是一家著名的虚拟化厂商，主要提供云计算和硬件虚拟化的软件和服务等一系列虚拟化产品。VMware Workstation 是一款宿主模型的工作站，其中包含针对 x86 架构的虚拟化软件，利用该软件，用户可以实现创建、运行若干个 x86 架构的虚拟机，而在每个虚拟机中均可以运行标准的各类操作系统，如 Windows、Linux 等；VMware ESX 服务器是 VMware 推出的一款企业级虚拟化产品，支持全虚拟化和半虚拟技术，属于 hypervisor 模型；VMware vSphere，是一整套虚拟化应用产品，它包含 VMware ESX Server 4、VMware Virtual Center 4.0、最高支持 8 路的虚拟对称多处理器（Virtual SMP）和 VMotion，以及例如 VMware HA、VMware DRS 和 VMware 统一备份服务等分布式服务。相比 VMware ESX 服务器，VMware vSphere 更加成熟、安全、稳定。

2.3 虚拟机自省技术

2.3.1 虚拟机自省技术概述

在虚拟化技术广泛应用的同时，其自身的安全问题也被广泛关注。在虚拟化环境中，传统的针对虚拟机的安全监控方案主要分为两类：第一类是通过在虚拟机操作系统内部安装监测工具，实时对虚拟机实施监控，称为内部监控。第二类是将监测工具安装在虚拟机操作系统外部，通常安装在 VMM 或者单独的一台客户机上，从而实时对客户机实施监控，称为外部监控。对于内部监控，由于事件是直接在虚拟机内部捕获，安全工具将能获取较为详尽的操作系统运行信息。然而，由于安全工具直接部署在被监控操作系统中，这样每台虚拟机均需部署安全工具，并且安全工具本身也容易遭受攻击。对于第二类，由于 VMM 在虚拟化环境中具有较高的操作控制权限和安全性，同时与虚拟机有良好的隔离性，所以在第二类中，监测工具一般更多被安装在 VMM 上。

外部监控的思想即为虚拟机自省：从虚拟机操作系统外部对虚拟机操作系统内部的运行状态进行实时监控。在 VMI 系统中，特权域可以访问非特权域的运行信息，可以介入被监控虚拟机的运行，对相关的状态数据进行访问控制。在虚拟化环境中，由于 VMM 通常具有比较高的控制权限、安全性和隔离性，我们可以将监控程序部署于 VMM 平台上，实时监控虚拟机的运行状态，进行安全保护。

2.3.2 虚拟机自省技术的实现方式和难点

根据是否在虚拟机操作系统中安装代理程序和是否对虚拟机监视器进行修改，通常将虚拟机自省技术分为三类^[36]，下面分别对其进行说明。

第一类是基于主机代理的虚拟机自省技术。这类自省技术通过在虚拟机操作系统中安装监视代理，以类似普通应用程序的形式在虚拟机操作系统中运行，从而获取虚拟机中的运行状态，包括内存表、注册表等。由于监视过程由虚拟机本机实施，所以基于主机代理的虚拟机自省技术监视语义精确，并且监视效率高。但同时，由于监视代理安装于被监视虚拟机内部，比较容易被攻击。这类虚拟机自省技术虽然实现简单、监视语义精确并且监视效率高，但是由于易被攻击，存在结构设计上的安全隐患，并未利用虚拟机自省技术本身的隔离性优势，只能算是一种初级虚拟机自省技术。基于主机代理的虚拟机自省技术基本结构如图 2.8 所示：

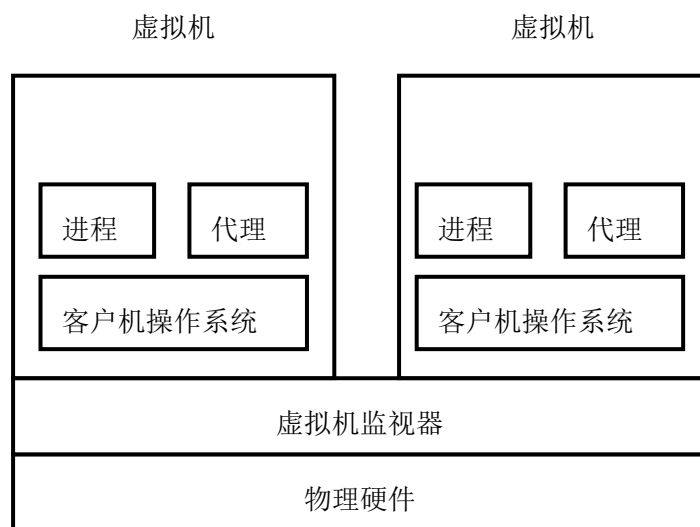


图 2.8 基于主机代理的虚拟机自省技术

第二类是基于主机内核驱动的虚拟机自省技术。这种方式原理上和第一类基本相同，也是通过向被监视虚拟机中安装代理，从而实现对虚拟机监视保护。与第一类不同的是，第二类虚拟机自省将监控代理以内核驱动形式部署在虚拟机内核中，安全性相比第一类虚拟机自省有所提高。但是由于监视代理部署在被监视虚拟机平台中，仍然容易遭受攻击。基于主机内核驱动的虚拟机自省技术结构如图 2.9 所示：

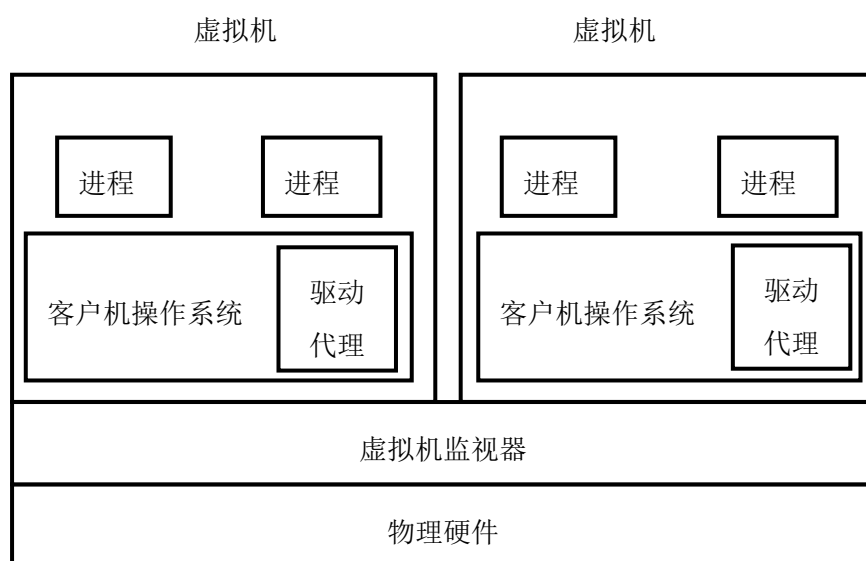


图 2.9 基于主机内核驱动的虚拟机自省技术

第三类是基于陷入、断点和回滚方式的虚拟机自省技术。相比前两类，第三类无需向被监控虚拟机安装代理，属于无代理监控。第三类 VMI 通过在修改 VMM 程序代码并在 VMM 中安装钩子函数从而获取被监控虚拟机运行状态，实时对虚拟机进行监控。第三类方式中，由于利用了虚拟化技术对权限的控制，VMM 拥有比被监控虚拟机更高的资源访问控制权限，从而虚拟机不能对底层 VMM 实施控制，使攻击者难以实施了攻击，系统安全性得到了很大提高。第三种方式获取到的被监控虚拟机的信息更加底层，存在语义鸿沟 (Semantic gap) 的技术难点，需对相关信息进行语义还原。但是由于这种方式直接由 VMM 实施对虚拟机的监控，不需要对被监控虚拟机做任何修改，其安全性相比前两种更高，消耗资源也更少。基于陷入、断点和回滚方式的虚拟机自省技术结构如图 2.10 所示：

语义鸿沟问题最早由 P. M. Chen 和 B. D. Nobl 于 2001 年 Hotos 会议上提出^[37]，是指从虚拟机操作系统内部和外部两个视角观察虚拟机，所观察到的视图存在差异。例如，从虚拟机内部看到的是进程、文件、内核模块等语义级别的对象，而在虚拟机外部（采用 VMI 方法）看到的只是内存页面、寄存器和硬盘块等抽象内容。在虚拟机自省系统设计和实现过程中，语义鸿沟是一个关键技术问题，即如何从低级语义信息中重构出操作系统级语义信息。目前已有一些解决语义鸿沟问题的方法，如 Xenaccess、VMwatcher 等^{[38][39][40]}，但这些方法还比较片面，功能比较单一，不能全面的实现低级语义还原并重构操作系统级语义的需求。

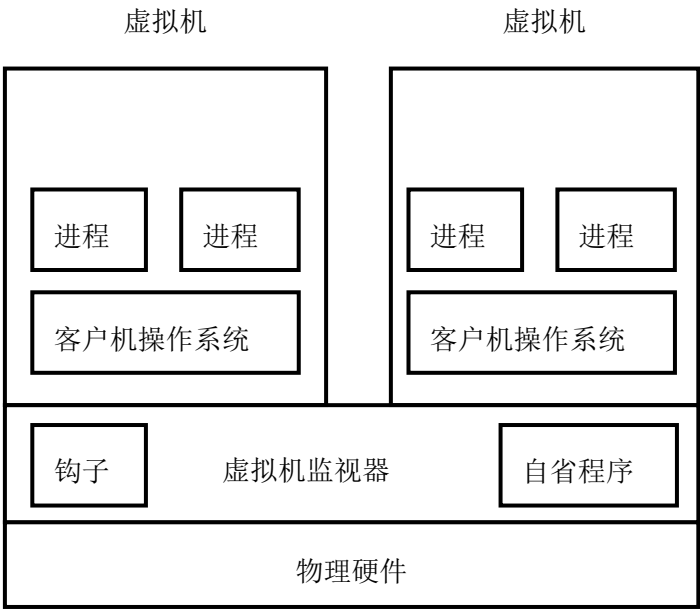


图 2.10 基于陷入、断点和回滚方式的虚拟机自省技术

2.4 本章小结

本章介绍了与课题研究相关的知识，包括虚拟化技术、虚拟机监视器和虚拟机自省技术。其中对几种主流的虚拟化技术做了详细的分析，如全虚拟化技术、半虚拟化技术、硬件辅助虚拟化技术以及操作系统级虚拟化技术等；然后分析对比了几款主流的虚拟机监视器，如 XEN、KVM 和 VMware 等。最后，针对虚拟化技术中相关的安全问题，对虚拟机自省技术做了分类介绍，熟悉了虚拟机自省系统的实施方法。

第三章 基于 KVM 的虚拟机自省系统设计

由于 KVM 的迅速发展和其自身优势，越来越多的厂商均开始选择 KVM 作为其虚拟化平台，很多云服务提供商也将其云服务从 XEN 平台迁移到 KVM 平台^[41]。针对基于 KVM 平台虚拟化环境中的安全问题，本章设计了基于 KVM 的虚拟机自省系统，从而对虚拟机的实时监控。

3.1 系统设计目标

为了实现对虚拟机操作系统运行状态的全面、高效的实时监控，本文基于虚拟化自省技术设计了一种全面的监控方案。监控方案的目标如下：

第一，设计的 VMI 系统可以实现对虚拟机操作系统的静态监控，包括自动识别操作系统类型、获取虚拟机操作系统中进程列表和已装载的内核模块列表。

第二，设计的 VMI 系统可以实现对虚拟机的动态监控，具体为在虚拟机外部实时捕获虚拟机上运行的操作系统中的各种动态事件，包括虚拟机操作系统中发生的系统调用、寄存器改变、内存页面访问、动态磁盘事件、中断和设备 I/O 事件等，同时对捕获到的动态事件进行语义还原。

第三，设计的 VMI 系统应该能够被高效的实现，并在虚拟化环境中容易被部署使用。具体来说，该方案应面向主流的虚拟化环境，并且系统性能损耗尽可能较低。

3.2 系统总体设计方案

基于对 VMI 系统提出的设计目标，本节对 VMI 系统的总体方案进行了设计，并给出了总体方案图，如图 3.1 所示。系统总体设计分为四层，分别为 VMI 库函数层、Libvirt 函数层、QEMU-KVM 层和 KVM 层。其中，VMI 库函数层、libvirt 函数层和 QEMU-KVM 层运行在用户态，KVM 模块层运行在内核态。

VMI 库函数层作为 VMI 系统的最上层，提供了监控虚拟机的各个功能函数，主要包括静态监控函数和动态监控函数两部分。系统调用这些函数接口，实现全面监控虚拟机。

Libvirt 是一套开源的 C 函数库，该函数库支持 Linux 下主流虚拟化工具，为开发人员提供了一套简单的虚拟化接口，支持多种开发语言。当前主流 Linux 平台上默认的虚拟化管理工具 virt-manager（图形化界面）、virt-install（命令行模式）等均基于 Libvirt 开发而成。Libvirt 通常包括一个 API 库、一个 daemon（libvirtd）和一个命令

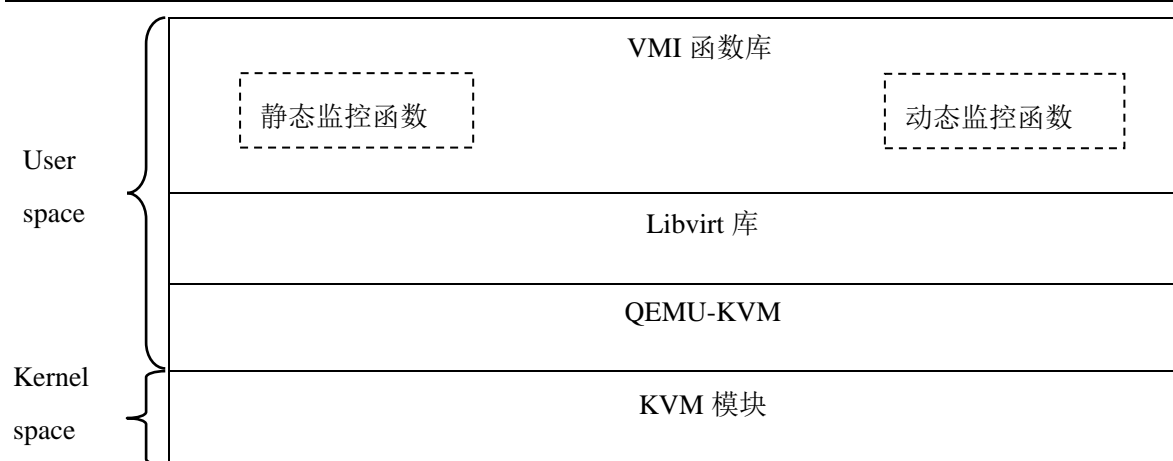


图 3.1 基于 KVM 的虚拟机自省系统的总体架构

行工具（virsh）。在原型系统中，Libvirt 层在 VMI 库函数层和 QEMU-KVM 层之间，为 VMI 库函数层提供了执行 QEMU-KVM 中各项命令的接口，同时将 VMI 库函数层的监控参数传递至 QEMU-KVM 层。

QEMU 由法布里斯·贝拉(Fabrice Bellard)编写，是一种以 GPL 许可证分发源码的模拟处理器，广泛应用于 GNU/Linux 平台上。QEMU 代码开源，研究人员可以通过对其进行修改并移植；默认支持多种架构，如 x86、AMD64、PowerPC 等；可扩展，可自定义新的指令集。而 QEMU-KVM 则是利用 KVM 内核模块和硬件辅助虚拟化技术，对 QEMU 的一种优化。在原型系统中，QEMU-KVM 层主要通过对 QEMU-KVM 源代码的修改，添加对虚拟机监控的命令，通过执行各种监控命令，从而响应上层各种函数调用。同时，QEMU-KVM 层接收上层传递的虚拟机监控参数，初始化相关数据，并通过 `kvm_vm_ioctl` 和 `kvm_vcpu_ioctl` 系统调用将监控命令和数据传入 KVM 层。

KVM 层是整个 VMI 监控系统的核心层。KVM 层运行于宿主机内核态，拥有对物理资源的访问控制特权。通过对物理资源的抽象，KVM 根据虚拟机需求将虚拟资源合适的分配给虚拟机，并对虚拟资源进行管理。在 KVM 层，我们对 KVM 内核模块源代码进行了修改，通过系统中断、陷入等方式实现在虚拟机外部对虚拟机内动态事件的捕获，通过地址翻译和对内存物理页面的访问等方式实现在虚拟机外部对虚拟机内部运行信息的语义还原。

3.3 各功能模块详细设计

VMI 系统中的监控对象主要包括 Linux 与 Windows 操作系统，主要的监控功能主要分为两类：静态监控和动态监控。下面分别对各功能模块进行详细设计。

3.3.1 静态监控

对虚拟机的静态监控主要包括三类：自动识别操作系统类型、监控虚拟机中的进程列表以及对虚拟机中已装载的内核模块进行监控。

通过研究发现，对于 Linux 和 Windows 两种操作系统的各发行版，不同版本系统中 `gdt.limit` 不同^[42]，所以我们可以将 `gdt.limit` 作为识别操作系统类型的特征码，从而我们可以通过测试不同版本操作系统中 `gdt.limit` 的值，实现对客户机操作系统的程序识别。

进程和内核模块在操作系统中以双向循环链表^[43]存在。进程的双向循环链表如图 3.2 所示：

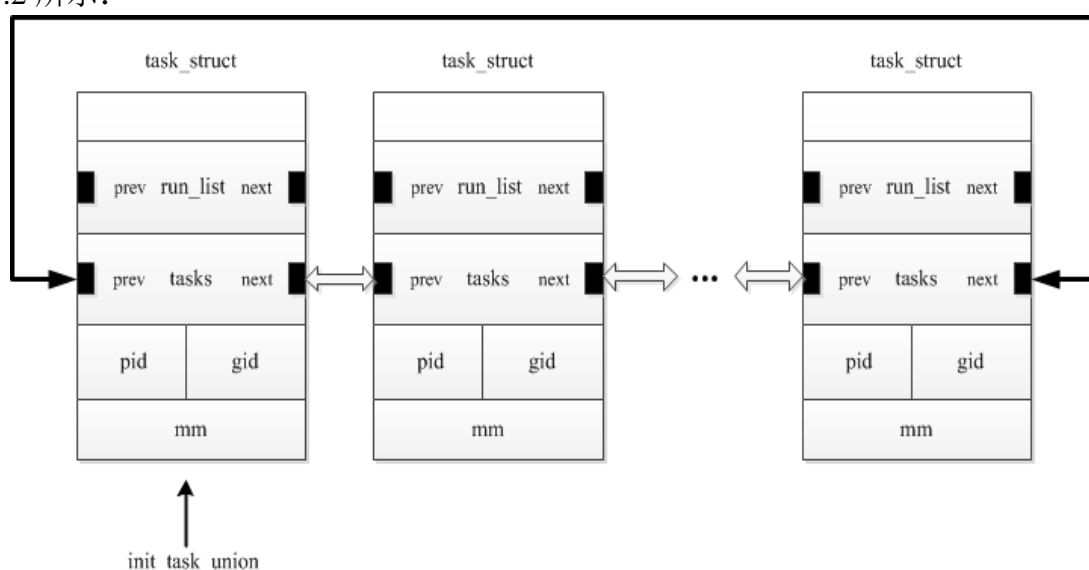


图 3.2 进程双向循环链表

VMI 系统对虚拟机操作系统进程和内核模块的监控主要是通过对各自双向循环链表的遍历实现。进程监控流程如图 3.3 所示，内核模块监控流程与进程监控流程相似。

在遍历过程中，从链表当前表项地址获取下一表项地址时，需要实现从虚拟机虚拟地址到主机物理地址的两次地址映射：第一次为将虚拟机虚拟地址映射到虚拟机物理地址，第二次是将虚拟机物理地址映射到主机物理地址。然后在虚拟机虚拟地址对应的主机物理地址上读取进程和内核模块相关信息，从而通过遍历双向循环链表实现对虚拟机操作系统中进程和内核模块的静态监控。

3.3.2 动态监控

对虚拟机的动态监控主要包括对虚拟机系统中系统调用、寄存器访问、内存页面访问、动态磁盘事件、中断和设备 I/O 等动态事件在虚拟机外部进行捕获与语义还原。

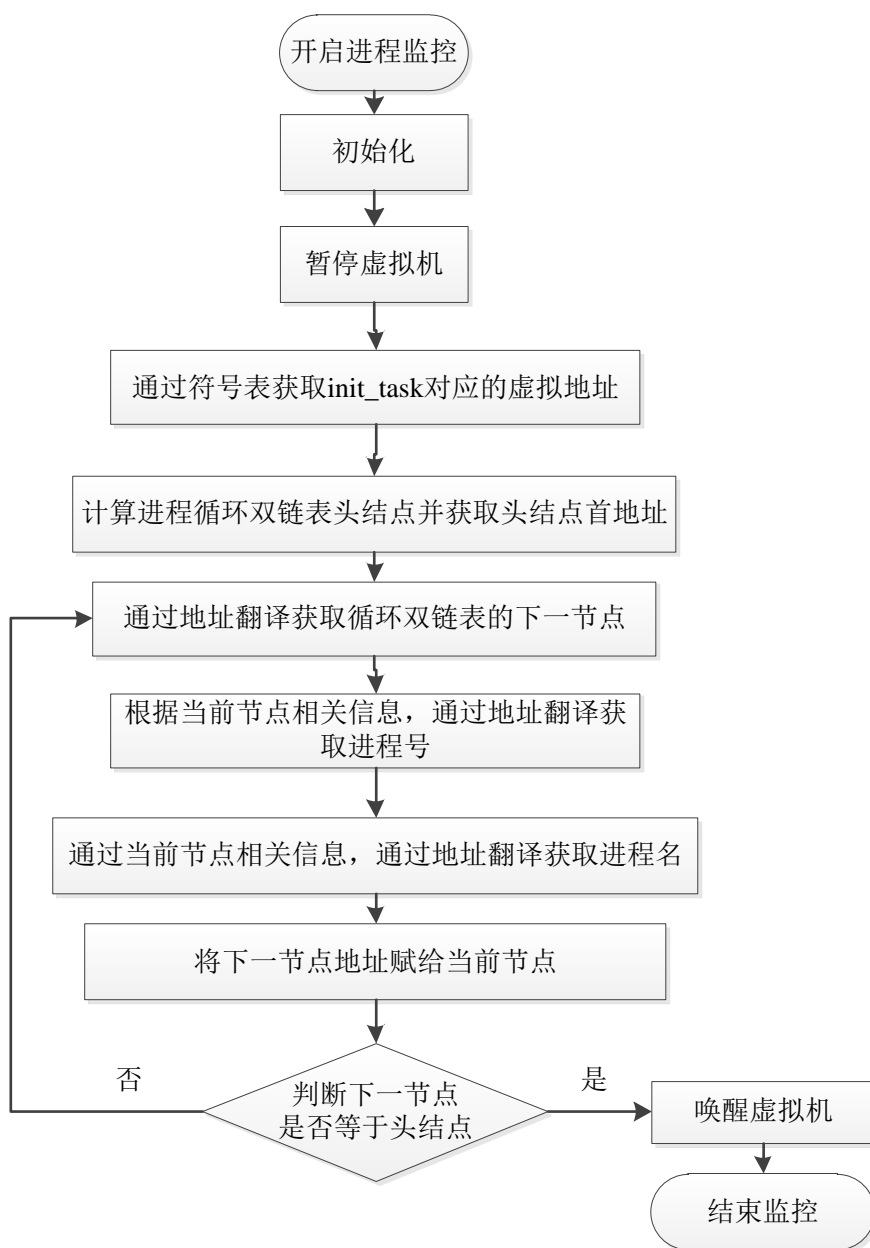


图 3.3 进程监控流程

系统调用监控

系统调用由操作系统实现并提供给应用程序的一组程序接口或应用编程接口 (Application Programming Interface, API), 是应用程序从用户态进入内核态, 执行内核态程序的唯一入口。因此, 监控系统调用对虚拟化安全具有重要意义。

在 VMI 原型系统中, 我们实现了对 x86 架构和 x86_64 架构的 Linux 和 Windows 系统的系统调用监控。由于 KVM 能捕获系统中断而不能直接捕获客户机发生的系统

调用，我们根据系统调用的不同方式，对虚拟机进行设置，使得虚拟机在发生系统调用时产生系统中断，从而使 KVM 捕获虚拟机系统调用，具体包括以下三种情况^[44]。

（1）基于中断的系统调用

基于中断的系统调用是指操作系统通过产生用户中断的方式实现系统调用。在 Linux 系统中，当发生系统调用时，系统将产生 int 0x80 中断；在 Windows 系统中，当发生系统调用时，系统将产生 int 0x2e 中断。当系统调用发生时，操作系统将产生用户中断，从而进入内核空间，执行系统调用。

由于 KVM 只能捕获系统中断，不能捕获用户中断，我们将虚拟机中断描述符表 (idt) 进行设置，使 idt 中只包含系统中断描述符，从而当用户中断发生后，虚拟机操作系统查询用户中断描述符时将地址越界，从而产生一般保护错误，KVM 捕获系统中断。

综上，动态监控基于中断的系统调用，我们将对虚拟机 idt 进行设置，使其只包含系统中断描述符。当发生系统中断时，查看中断号是否是 0x80(Linux 系统调用)或者 0x2e(Windows)，若是，收集系统调用相关数据，并模拟该系统调用，然后将控制权交还虚拟机；否则向虚拟机注入异常并交还控制。

基于中断的系统调用监控处理流程如图 3.4 所示。

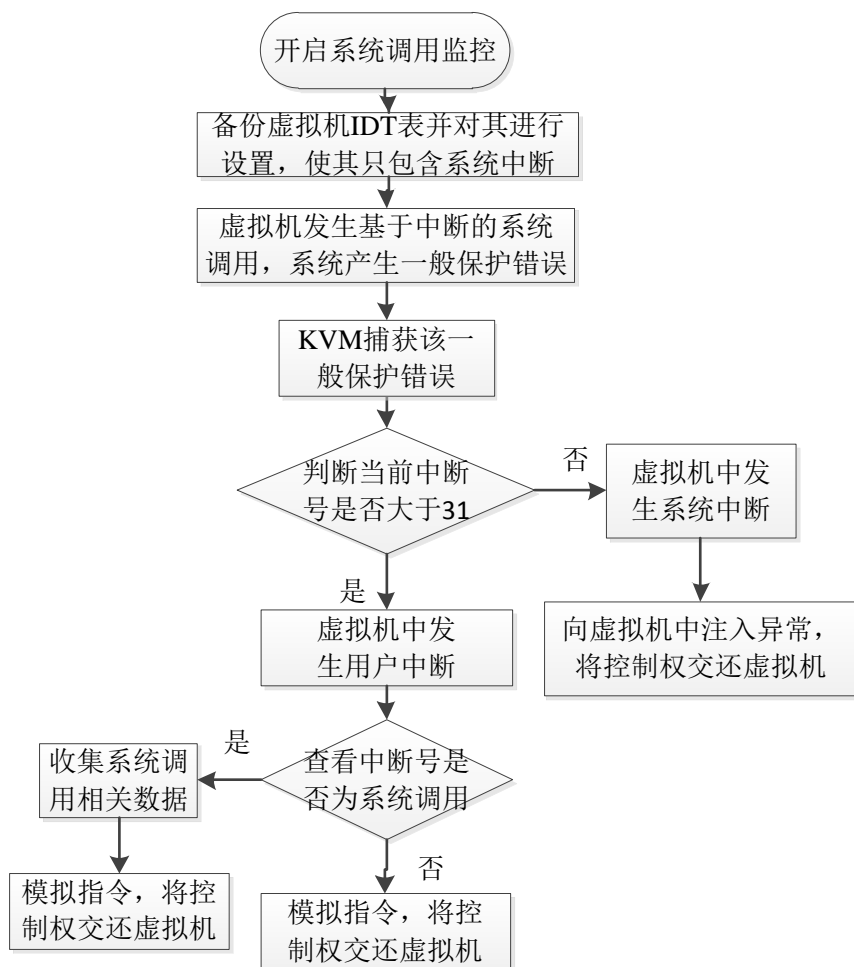


图 3.4 基于中断的系统调用监控

(2) 基于 SYSENTER/SYSEXIT 的系统调用

对于 x86 架构，若操作系统采用快速系统调用机制，系统调用则通过 SYSENTER 和 SYSRET 指令执行。由于 SYSENTER 指令执行过程中需将 SYSENTER_CS_MSR 拷贝到 CS 寄存器中。为了捕获系统中断，我们在 KVM 中保存 SYSENTER_CS_MSR 的值，然后在 SYSENTER_CS_MSR 装载 NULL，从而在发生系统调用时，客户机操作系统会向 CS 寄存器中装载 NULL，继而发生一般保护错误(GP)，引起系统中断。

对于基于 SYSENTER/SYSEXIT 的系统调用，当 KVM 捕获 GP 后，KVM 查看当前指令，从而判断系统是否发生系统调用。处理流程如图 3.5 所示。

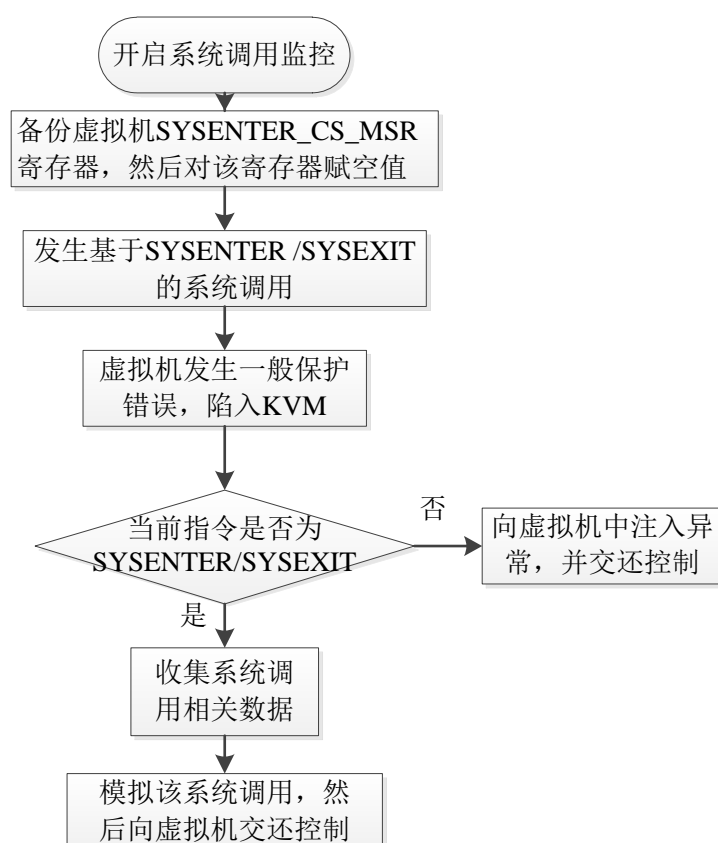


图 3.5 基于 SYSENTER/SYSEXIT 的系统调用监控

(3) 基于 SYSCALL 的系统调用

在 x86_64 体系结构中，操作系统通过 SYSCALL 和 SYSRET 指令实施快速系统调用。该机制可以通过设置 EFER.SCE 位关闭或开启。所以，对于 x86_64 体系结构，我们可以设置 EFER.SCE==0，从而当发生系统调用时产生 UD 系统中断。当 UD 系统中断产生后，KVM 查看当前指令，若当前指令为 SYSCALL，则收集数据、模拟指令并交还控制权；否则向虚拟机中注入操作码异常并交还控制权。处理流程如图 3.6 所示。

寄存器监控

寄存器是 CPU 内部的有限存贮容量的高速存贮部件，包括通用寄存器、专用寄存器和控制寄存器，常用来存储计算数据、运行指令、存储器地址和系统状态信息等。

由于寄存器在计算机系统中的重要作用，实现对寄存器的监控具有重要意义。比如，在对系统调用监控时，可以通过读取 RAX 寄存器获取当前系统调用号。

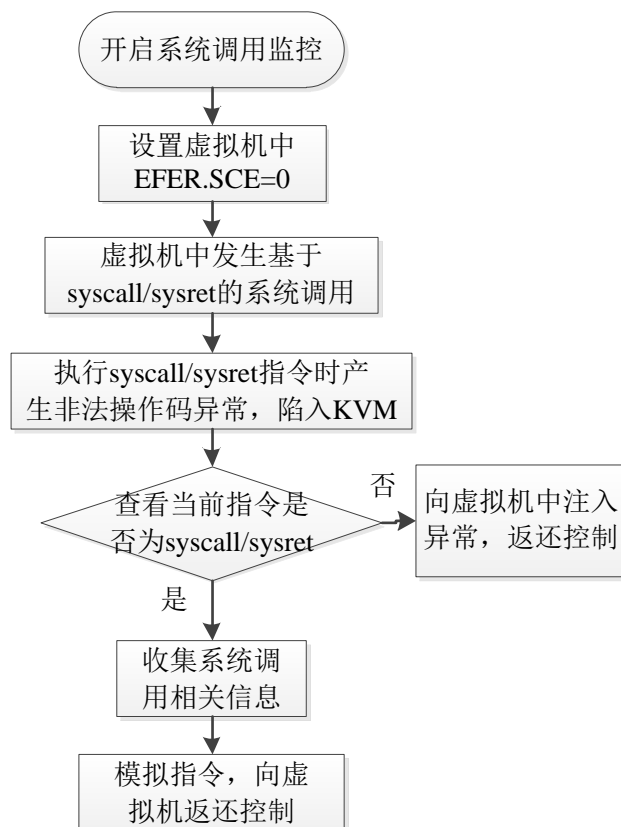


图 3.6 基于 SYSCALL/SYSRET 的系统调用监控

对于寄存器监控，我们可以通过对虚拟机操作系统进行设置，实现虚拟机操作系统的单步运行，每执行一条指令，检查被监控寄存器是否发生变化，从而实现对特定寄存器变化的监控。

内存页面监控

对于内存页面监控，原型系统通过客户机中被监控页面虚拟地址进行地址翻译^[43]，找到该虚拟地址在主机中对应的影子页表，通过对该影子页表的访问权限进行设置，使被监控页面为只读页面，从而当客户机中对被监控页面进行写操作时产生 **page fault**，并陷入 KVM。当 KVM 捕获 **page fault** 后，系统检查当前 CR2 寄存器中地址值是否指向被监控页面，若指向被监控页面，则系统捕获虚拟机中对被监控页面的写操作事件，并在日志中打印相关信息，然后通过改写被监控页面访问权限，使被访问页面可写，并由 KVM 模拟对被监控页面的写操作，而后设置被监控页面为只读文件，继续执行页面监控；若当前系统中 CR2 寄存器中地址没有指向被监控页面，则向虚拟机中注入 **page fault** 异常，将控制交还虚拟机。具体的内存页面监控流程如图 3.7 所示。

磁盘动态事件监控

磁盘是计算机系统中存储信息的重要资源，用户的各种资源信息通常以各种不同格式的文档存储在磁盘的各类文件目录中，并组成文件系统。文件系统安全是系统安

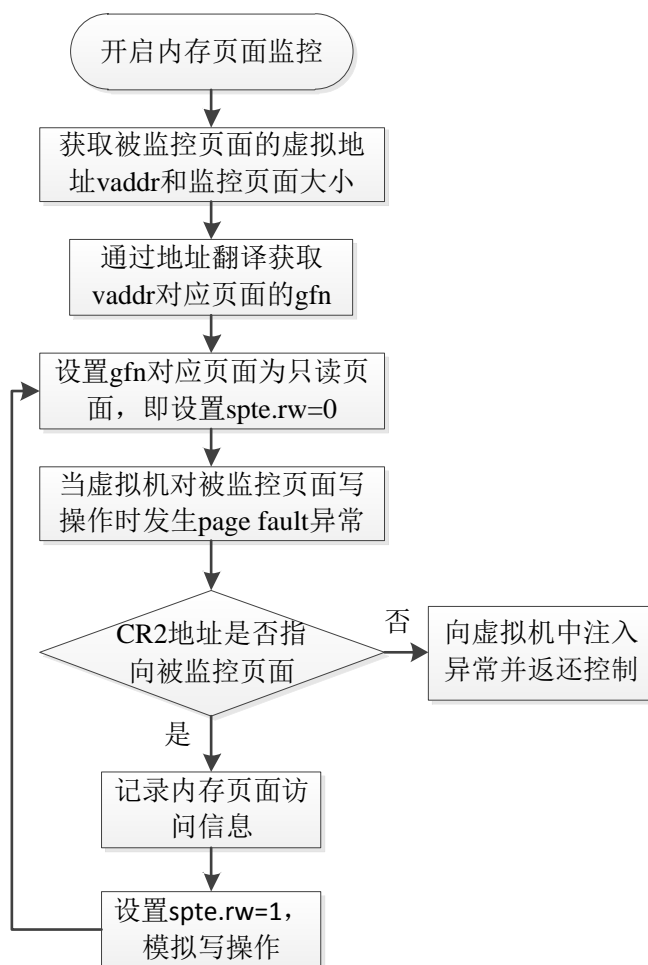


图 3.7 内存页面监控

全的重要部分，通过对磁盘动态事件的监控，可以有效实现对磁盘的安全保护。

在磁盘动态事件监控模块中，我们主要实现了对虚拟机中创建目录(mkdir)和删除目录(rmdir)等动态事件的监控。由于在客户机中创建目录和删除目录分别需调用 sys_mkdir 和 sys_rmdir 系统调用，所以在虚拟机自省系统中监控 sys_mkdir 和 sys_rmdir 系统调用，从而实现对虚拟机中创建目录和删除目录的动态监控。

中断事件监控

中断是指在操作系统运行过程中，当出现某种需要时，处理器暂听当前程序的执行转而执行处理新情况的程序的执行过程。即在操作系统运行时，出现了一个必须由处理器马上处理的新状况，从而，处理器暂停当前运行程序，转而执行新的代码，这种处理过程即为中断。操作系统中的中断事件分为两种：系统中断和用户中断。在中断描述符表（IDT）中，系统中断一般存放于前 32 个表项中，用户中断一般存放在 32 到 256 个表项中。

在基于 KVM 的虚拟环境中，Intel VT-x 技术允许虚拟机中系统中断陷入 Hypervisor，即 KVM 内核模块可以直接不过虚拟机中的系统中断，只能捕获虚拟机

操作系统中的系统中断，不能捕获用户中断。为了使 KVM 捕获虚拟机操作系统中的用户中断，我们对中断处理过程进行了研究。

操作系统对中断处理过程一般分为如下步骤：请求中断、响应中断、关闭中断、保留断点、中断源识别、保护现场、中断服务子程序、恢复现场和中断返回。其中在运行中断服务子程序前需根据中断号查找 IDT 表，找到响应的中断处理程序入口，因此，我们可以通过对 IDT 表做设置实现对用户中断事件的捕获。将 IDT 表设置为只包含系统中断，不包含用户中断，这样当虚拟机发生用户中断时，虚拟机将发生一般保护错误的系统中断，从而将用户中断转化成系统中断，实现对所有中断事件的捕获。

虚拟机设备 I/O 事件监控

现代计算机系统中配置了大量的外设，即 I/O 设备。依据它们的工作方式的不同，通常将其分为三大类：字符设备（character device）、块设备（block device）和网络通信设备。

字符设备又叫做人机交互设备，如鼠标、键盘等。用户通过这些设备实现与计算机通信。它们大多是以字符为单位发送和接受数据的，数据通信的速度比较慢。

块设备又叫外部存储器，如磁盘、光盘等。用户通过这些设备实现程序和数据的长期保存。

网络通信设备主要用于与远程设备的通信，这类设备主要有网卡、调制解调器等。

在基于 KVM 的虚拟化环境中，各个虚拟机系统中的各设备都是通过 KVM 对主机系统中各物理设备的抽象模拟实现的，所以虚拟机中对于各中设备的输入输出均要通过 KVM 模块处理，从对虚拟设备的访问映射到主机物理设备的访问。因此，对于虚拟机设备 I/O 的监控，我们可以通过在 KVM 添加钩子函数实现。当虚拟机中设备发生 I/O 事件时，原型系统通过在 KVM 中的钩子函数获取设备 I/O 的相关信息，实现对虚拟机设备 I/O 事件的监控。

3.4 本章小结

本章首先根据现有虚拟化技术及虚拟化环境中存在的安全问题，对 VMI 系统确定设计目标；然后根据设计目标完成基于 KVM 的虚拟机自省系统的总体架构设计，并给出总体架构图；最后根据监控功能对 VMI 系统进行功能模块划分，并进一步对各模块进行详细设计，完成了原型系统的设计工作。

第四章 原型系统实现

通过第三章中对原型系统的设计，本章主要对原型系统利用 C 语言编码实现，主要工作包括三方面：VMI 库函数层编码实现、对 QEMU-KVM 程序代码修改和对 KVM 程序代码的修改。

4.1 VMI 库函数层实现

VMI 库函数层包括静态监控函数和动态监控函数。其中静态监控函数具体包括操作系统类型识别函数（Get_ostype）、进程列表罗列函数（Process_list_print）和已加载内核模块罗列函数（Module_list_print），动态监控函数具体包括系统调用监控（Syscall_monitor/Syscall_trace）、寄存器监控（Register_monitor）、内存页面监控（Memory_monitor）、动态磁盘事件监控（Disk_monitor）、中断事件监控（Interrupt_monitor）和设备 I/O 事件监控（Device_io_monitor）。VMI 库函数层各个监控函数具体说明如下：

（1）操作系统类型识别函数（Get_ostype）

函数原型：Get_ostype(char *name);

函数说明：自动识别虚拟机操作系统功能，包括 Linux 和 Windows 操作系统。

参数说明：

name：虚拟机名。

（2）进程列表罗列函数（Process_list_print）

函数原型：Process_list_print(char *name);

函数说明：打印虚拟机中运行的进程列表。

参数说明：

name：虚拟机名。

（3）已加载内核模块罗列函数（Module_list_print）

函数原型：Module_list_print(char *name);

函数说明：打印虚拟机中已加载的内核模块；

参数说明：

name：虚拟机名。

（4）系统调用监控函数（Syscall_monitor/Syscall_trace）

函数原型：

Syscall_monitor_start(char *name, unsigned long condition_value);

`Syscall_trace_start(char *name);`

`Syscall_monitor_stop(char *name);`

函数说明:

`Syscall_monitor_start` 和 `Syscall_trace_start` 分别为开启对特定系统调用和全部系统调用的监控, `Syscall_monitor_stop` 为关闭系统调用监控。

参数说明:

`name`: 虚拟机名;

`condition_value`: 监控特定系统调用时的系统调用号。

(5) 寄存器监控 (Register_monitor)

函数原型:

`Register_monitor_start(char *name, char *condition_reg, unsigned long condition_value);`

`Register_trace_start(char *name, char *condition_reg);`

`Register_monitor_stop(char *name);`

函数说明:

`Register_monitor_start` 和 `Register_trace_start` 分别为开启某寄存器特定值监控和所有变化监控, `Register_monitor_stop` 为关闭寄存器监控。

参数说明:

`name`: 虚拟机名;

`condition_reg`: 被监控寄存器;

`condition_value`: 监控特定系统调用时的系统调用号。

(6) 内存页面监控 (Memory_monitor)

函数原型:

`Memory_monitor_start(char *name, unsigned long address, int size);`

`Memory_monitor_stop(char *name, unsigned long address, int size);`

函数说明:

`Memory_monitor_start` 为开启虚拟机内存页面监控函数, `Memory_monitor_stop` 为关闭虚拟机内存页面监控函数。

参数说明:

`name`: 虚拟机名;

`address`: 需监控的内存地址;

`size`: 需要监控的内存页面数。

(7) 动态磁盘事件监控 (Disk_monitor)

函数原型:

```
Disk_monitor_start(char *name, diskmon_function function);
```

```
Disk_monitor_stop(char *name);
```

函数说明:

Disk_monitor_start 和 Disk_monitor_stop 分别为开启和关闭动态磁盘事件监控。

参数说明:

name: 虚拟机名;

function: 被监控的动态磁盘事件类型, 包括创建文件目录事件和删除文件目录事件。

(8) 中断事件监控 (Interrupt_monitor)

函数原型:

```
Interrupt_monitor_start(char *name);
```

```
Interrupt_monitor_stop(char *name);
```

函数说明:

Interrupt_monitor_start 和 Interrupt_monitor_stop 分别实现开启和关闭中断事件监控。

参数说明:

name: 虚拟机名。

(9) 设备 I/O 事件监控 (Device_io_monitor)

函数原型:

```
Device_io_monitor_start(char *name, int port);
```

```
Device_io_monitor_stop(char *name);
```

函数说明:

Device_io_monitor_start 和 Device_io_monitor_stop 分别为开启和关闭设备 I/O 监控。

参数说明:

name: 虚拟机名;

port: 要监控的设备端口号。

4.2 QEMU-KVM 源代码的修改

4.2.1 QEMU-KVM 源代码介绍

QEMU-KVM 源代码可以在其官网下载, 原型系统使用的版本是 qemu-kvm-1.2.0。QEMU-KVM 源代码中文件非常多, 下面分别对源代码中主要文件目录和主要文件分别介绍。

主要目录包括 hw/、include/、linux-user/、tcg/、tests/、docs/和 target-XXX/等目录。其中：

hw/目录主要包含了所有包含的硬件设备；

include/目录主要包含一些头文件；

linux-user/目录主要有 linux 下用户模式的一些程序；

tcg/目录中有动态翻译工具 tcg 的主要程序，这些程序将 TCG OP 转化为 host binary 的部分，这个目录下也包含了若干以架构名字命名的目录，存放着针对该架构进行处理的程序；

tests/目录主要包含各种测试代码；

docs/主要包含各种文档；

target-XXX/目录中包含了 qemu-kvm 所支持的所有处理器架构，如 alpha、arm、cris、i386、lm32、m68k、microblaze、mips、openrisc、ppc、s390x、sh4、sparc、unicore32 和 xtensa 等。

主要比较重要的文件有：/vl.c、/cpus.c、/exec-all.c、/exec.c、/cpu-exec.c、/hmp-commands.h 和/monitor.c。

/vl.c 文件中定义了 QEMU-KVM 的 main 函数，它也是程序执行的初始点，利用这个函数我们可以创建虚拟化环境。它通过对相关参数进行解析，初始化物理内存设备、CPU 参数以及 KVM 等等。在完成初始化工作后，程序将跳转到相关分支程序，这些程序包含在一些文件中，如如：/cpus.c、/exec-all.c、/exec.c 和/cpu-exec.c 等

/hmp-commands.h 头文件中主要包含 QEMU-KVM 的命令接口，原型系统中 VMI 层的各监控函数接口即是通过 Libvirt 中 exec_qemp_cmd 函数接口调用该文件中定义的相关 QEMU 命令实现将相关监控命令和参数传递至 KVM 层中。

/monitor.c 文件中各函数主要负责对/hmp-commands.h 中定义的各监控命令进行解释和参数初始化，原型系统中各监控命令的解释和参数传递也是定义在该文件中。

4.2.2 QEMU-KVM 源代码的修改

QEMU-KVM 在原型系统中的主要作用实现 VMI 库函数层中各监控功能与底层 KVM 模拟的信息传递。具体来说，VMI 库函数层的各个监控功能通过调用 Libvirt 库提供的 exec_qemp_cmd 函数接口执行 QEMU-KVM 相应命令，将监控命令及参数传递给 QEMU-KVM 层；然后 QEMU-KVM 层通过对 QEMU-KVM 相应命令的解析，提取相关监控参数，并对相关参数进行初始化；最后 QEMU-KVM 通过与 KVM 之间的 IOCTL 接口将监控命令和参数传递给 KVM 层，最终在 KVM 层实施监控操作。

对 QEMU-KVM 源代码的修改主要集中在/hmp-commands.h、/monitor.c 和 /kvm_vmi.h 三个文件中，下面分别对其进行说明。

在/hmp-commands.h 头文件中, QEMU-KVM 中主要定义了各种 QEMU 命令, 如帮助 (help)、创建虚拟机 (create)、销毁虚拟机 (destory)、启动虚拟机 (start)、关闭虚拟机 (shutdown)、重启虚拟机 (reboot)、查看虚拟机信息 (info) 等。对/hmp-commands.h 头文件的修改主要是向其中添加新的 QEMU 命令, 从而实现接收上层监控信息的功能。添加的 QEMU 命令主要涉及到为动态监控功能提供 QEMU-KVM 层接口, 主要包括开启全部系统调用监控 (start_sctrace)、关闭特定系统调用监控 (stop_sctrace)、开启特定系统调用监控 (start_scmon)、关闭特定系统调用监控 (stop_scmon)、开启寄存器监控 (start_regmon)、关闭寄存器监控 (stop_regmon)、开启内存页面监控 (start_memon)、关闭内存页面监控 (stop_memon)、开启磁盘动态监控 (start_diskmon)、关闭磁盘动态监控 (stop_diskmon)、开启中断监控 (start_intmon)、关闭中断监控 (stop_intmon)、开启设备 I/O 监控 (start_iomon)、关闭设备 I/O 监控 (stop_iomon) 等。

在/monitor.c 文件中, QEMU-KVM 主要实现了各种监控解析函数, 具体解析/hmp-commands 头文件中的各种命令。对/monitor.c 的修改主要为向其中添加监控解析函数, 如系统调用监控对应的解析函数 do_start_sctrace()、do_stop_sctrace()、do_start_scmon()和 do_stop_scmon(), 寄存器监控对应的解析函数 do_start_regmon()和 do_stop_regmon(), 内存页面对应的解析函数 do_start_memon()和 do_stop_memon(), 磁盘监控对应的解析函数 do_start_diskmon()和 do_stop_diskmon(), 中断监控对应的解析函数 do_start_intmon() 和 do_stop_intmon(), 设备 I/O 对应的解析函数 do_start_iomon()、do_stop_iomon()。在各相关监控命令的解析函数中, QEMU-KVM 通过 kvm_vm_ioctl 或 kvm_vcpu_ioctl 函数接口调用 IOCTL 系统调用, 将解析过的监控信息传递至 KVM 层, 由 KVM 层具体实施监控。

在/kvm_vmi.h 头文件中, QEMU-KVM 主要定义了/monitor.c 中监控解析函数所涉及的相关数据结构, 如 kvm_syscall_data 结构体、kvm_regmon_data 结构体、kvm_memon_data 结构体、kvm_diskmon_data 结构体、kvm_intmon_data 结构体以及 kvm_iomon_data 结构体等。通过定义这些结构体, 存储/monitor.c 中解析出的监控相关数据, 并由 QEMU-KVM 将这些数据传递至 KVM 层。

最后通过将修改过的 QEMU-KVM 代码与标准 QEMU-KVM 代码对比, 生成标准 QEMU-KVM 的补丁文件: qemu-kvm-1.2.0.patch。

4.3 KVM 源代码的修改

4.3.1 KVM 源代码介绍

原型系统中使用的 KVM 版本为 kvm-kmod-2.6.37, 源代码可以在官网下载。KVM

源代码中主要包含/include 目录、/ia64 目录、/x86 目录、/powerpc 目录、/usr 目录和 /script 目录等，其中/include 目录主要包含各头文件；/ia64、/x86、/powerpc 目录主要包含对各自架构系统的虚拟化代码，原型系统中对 KVM 代码的修改重要集中在/ia64 目录和/x86 目录；/usr 目录主要是针对各架构系统资源的虚拟化代码。对 KVM 源代码中关键函数介绍如下：

(1) 初始化函数 kvm_init()

函数原型：

```
kvm_context_t kvm_init(struct kvm_callbacks *callbacks, void *opaque);
```

函数说明：

该函数在用户态创建一个虚拟机上下文，用以在用户态保存基本的虚拟机信息，这个函数是创建虚拟机第一个需要调用的函数，函数返回一个 kvm_context_t 结构体。

参数说明：

callbacks 为结构体 kvm_callbacks 变量，该结构体包含指向函数的一组指针，用于在客户机执行过程中因为 I/O 事件退出到用户态的时候处理的回调函数；

opaque 一般未使用。

(2) 虚拟机内核创建函数 kvm_create()

函数原型：

```
kvm_create(kvm_context_t kvm, unsigned long phys_mem_bytes, void **phys_mem);
```

函数说明：

该函数主要用于创建一个虚拟机内核环境。

参数说明：

kvm_context_t 表示传递的用户态虚拟机上下文环境；

phys_mem_bytes 表示需要创建的物理内存的大小；

phys_mem 表示创建虚拟机的首地址。

(3) 系统调用函数 ioctl()

函数原型：

```
static long kvm_vm_ioctl (struct file *filp, unsigned int ioctl, unsigned long arg);
```

函数说明：

用于在内核中创建和虚拟机相关的数据结构。

参数说明：

filp 表示指向文件的指针，用于返回虚拟机的文件描述符；

ioctl 表示相关系统调用命令；

arg 表示相关参数。

(4) 虚拟处理器创建函数 kvm_create_vcpu()

函数原型:

```
int kvm_create_vcpu(kvm_context_t kvm, int slot);
```

函数说明: 该函数用于创建虚拟处理器。

参数说明:

kvm 表示对应用户态虚拟机上下文;

slot 表示需要创建的虚拟处理器的个数。

(5) 虚拟机内存创建函数 `kvm_create_phys_mem()`

函数原型:

```
kvm_create_phys_mem(kvm_context_t kvm, unsigned long phys_start, unsigned  
len, int log, int writable);
```

函数说明:

该函数主要用于创建虚拟机的内存空间。

参数说明:

kvm 表示用户态虚拟机上下文信息;

phys_start 为分配给该虚拟机的物理起始地址;

len 表示内存大小;

log 表示是否记录脏页面;

writable 表示该段内存对应的页表是否可写。

(6) 虚拟处理器调度运行函数 `kvm_run()`

函数原型:

```
kvm_run(kvm_context_t kvm, int vcpu, void *env);
```

函数说明:

该函数主要用于调度运行虚拟处理器。

参数说明:

kvm 表示用户虚拟机上下文信息;

vcpu 表示虚拟处理器;

env 表示相应架构虚拟处理器的运行状态信息。

(7) 异常处理函数 `vmx_handle_exit()`

函数原型:

```
vmx_handle_exit(struct kvm_run *kvm_run, struct kvm_vcpu *vcpu);
```

函数说明:

该函数主要负责在 KVM 内核模块中实现对虚拟机系统的异常处理。

参数说明:

kvm_run 表示当前虚拟机实例的运行状态信息;

vcpu 表示对应的虚拟处理器。

以上函数在虚拟机系统创建、运行以及异常处理中有着非常关键的作用。

原型系统涉及到 KVM 源代码中的重要文件包括 /x86/kvm_main.c、/ia64/kvm_main.c、/x86/kvm-x86.c、/ia64/kvm-ia64.c、/x86/vmx.c、/x86/kvm_main.h、/ia64/kvm_main.h、/x86/kvm-x86.h、/ia64/kvm-ia64.h、/x86/vmx.h 等。

4.3.2 KVM 源代码的修改

KVM 层作为整个虚拟机自省系统的核心层，具体实施各监控功能，包括监控事件的捕获及语义还原。对 KVM 源代码的修改主要分为两部分：

对虚拟机底层环境进行设置，使其在发生被监控事件时陷入 KVM，由 KVM 实施对被监控事件的捕获；

在 KVM 捕获到被监控事件后，需要在 KVM 层对所捕获的虚拟机内事件进行语义还原，并将还原出的语义信息输出到系统日志中。

对于第一部分的代码修改主要集中在 kvm_main.c、kvm_x86.c、kvm_ia64.c、kvm_main.h、kvm_x86.h、kvm_ia64.h 以及新添加的 vmi.c 和 vmi.h 中。其中在 kvm_main.c 中主要是通过添加 vmi_kvm_init 函数实现对 KVM 层监控环境的初始化，具体如对 shadow_idt 的初始化等；在 kvm_x86.c 和 kvm_ia64.c 中主要实现 kvm_arch_vm_ioctl() 函数的修改，具体为在 kvm_arch_vm_ioctl() 函数中添加相应的监控命令。在新添加的 vmi.c 文件中，主要包含对 kvm_arch_vm_ioctl() 函数中添加的新命令的具体实现，主要作用是具体对虚拟机底层做设置，实现当虚拟机中发生被监控事件时，虚拟机陷入 KVM，从而使 KVM 实现捕获虚拟机中的各动态事件。以上 C 语言程序对应的头文件中实现对相关数据的声明。

对于第二部分的代码修改主要集中在 /x86/vmx.c、/x86/vmx.h、/x86/vmi.c 和 /x86/vmi.h 中。其中对 vmx.c 的修改主要是在 handle_exception() 函数添加异常处理，当虚拟机中发生有 VMI 系统导致的异常时，由该函数识别异常是 VMI 系统导致的异常还是虚拟机系统的自然异常，如果是虚拟机系统的自然异常，将异常注入虚拟机，并将控制权返还给虚拟机系统，如果是由 VMI 系统导致的异常，则通过调用 vmi.c 中定义的各数据收集函数实现对虚拟机中发生的事件进行数据收集，并将收集到的数据打印到系统日志中，完成对异常进行处理并将控制权返还给虚拟机。在 vmi.c 文件中定义了各类监控功能的数据收集函数。/x86/vmx.h 和 /x86/vmi.h 中定义了其对应 C 语言程序中设计的相关数据结构声明。

下面对原型系统涉及在 KVM 模块源代码中修改、添加的相关关键函数做一介绍：

(1) 系统调用函数 kvm_arch_vm_ioctl()

函数原型：

```
long kvm_arch_vm_ioctl(struct file *filp,unsigned int ioctl, unsigned long arg);
```

函数说明:

该函数为 KVM 源代码提供的系统调用处理函数,主要负责 KVM 中对应用层 QEMU-KVM 的相关系统调用的处理。原型系统对该函数的修改主要涉及对将 QEMU-KVM 中的相关监控命令及参数通过系统调用传递到 KVM 层,通过 KVM 层进一步处理。

(2) 开启系统调用监控函数 start_scmonitor()

函数原型:

```
int start_scmonitor(struct kvm *kvm,int64_t idt_index,char* syscall_reg, int func,enum mode mode);
```

函数说明:

该函数是原型系统对 KVM 源代码新添加的函数,存在于/x86/vmi.c 文件中。该函数通过对相关数据进行设置,如将中断描述符表截断,只保留其系统中中断描述符表项;将 MSR_EFER 寄存器中 SCE 位置 0;对 MS_SYSENTER_CS 寄存器赋 NULL 等,从而当虚拟机系统中发生系统调用事件时可以陷入 KVM 中,使 KVM 层捕获系统调用事件。

(3) 开启寄存器监控函数 start_regmonitor()

函数原型:

```
int start_regmonitor(struct kvm *kvm,char *reg,int64_t reg_val,int flag);
```

函数说明:

该函数是原型系统中 KVM 在/x86/vmi.c 文件中新添加的函数,主要通过设置虚拟机系统为单步运行,从而实现对被监控寄存器的跟踪和监控。

(4) 开启中断监控函数 start_intmonitor()

函数原型:

```
int start_intmonitor(struct kvm *kvm,int64_t idt_index);
```

函数说明:

该函数同样是原型系统中新添加的,添加在/x86/vmi.c 文件中,主要对虚拟机系统中中断描述符进行设置,保留中断描述符表中关于系统中断的表项,删除其中关于用户中断的表项,从而当虚拟机系统中发生中断事件时,虚拟机系统可以陷入 KVM 内核,使 KVM 可以捕获虚拟机系统中的中断事件。

(5) 开启内存访问监控 start_memonitor()

函数原型:

```
int start_memonitor(struct kvm *kvm,unsigned long addr,int size);
```

函数说明:

该函数主要负责对虚拟机系统内存访问的监控。在/x86/vmi.c 文件中添加该函数，首先将被监控页面虚拟地址进行地址翻译，获得该虚拟地址在主机中对应的影子页表，通过该影子页表的访问权限进行设置，使被监控页面为只读页面，从而当虚拟机中对被监控页面进行写操作时产生 page fault，并陷入 KVM，从而实现捕获虚拟机中的内存访问事件。

(6) 异常处理函数 handle_exception()

函数原型：

```
static int handle_exception(struct kvm_vcpu *vcpu);
```

函数说明：

该函数存在于 KVM 源代码中/x86/vmx.c 文件中，主要负责在 KVM 层对虚拟机系统中的异常进行处理。原型系统对该函数的修改主要针对在开启监控过程中，对由监控操作产生的系统异常进行处理，如对虚拟机系统中的一般保护错误、页面保护错误、未定义指令执行错误等系统异常。原型系统通过该函数实现对监控产生的各种异常进行处理，并调用 handle_gp()函数、handle_ud()函数、handle_page_fault()函数、handle_io 函数实现对相关监控信息的语义还原并将相关信息进行打印到日志文件中。之后，将控制权交还虚拟机系统，使虚拟机系统能够继续正常运行。

原型系统中对 KVM 源码的修改主要包含以上两部分，通过捕获相关事件、还原相关语义信息，实现对虚拟机系统的全面监控。最后通过将修改过的 KVM 代码与标准 KVM 代码对比，生成标准 KVM 的补丁文件：kvm-kmod-2.6.37.patch。

4.4 本章小结

本章主要根据原型系统的总体设计，通过编码实现原型系统。主要工作包括三方面：首先是完成 VMI 库函数层中各监控函数的编码工作，并给出函数接口；然后是对 QEMU-KVM 源代码的分析和修改，提供了原型系统中对 KVM 层调用的接口；最后通过对 KVM 源代码的分析和修改，实现了系统对虚拟机操作系统的各项监控功能。

第五章 原型系统测试与分析

5.1 搭建系统测试环境

5.1.1 原型系统安装所需软件包

为了构建原型系统，需要从网上下载多个标准软件包，并配合我们交付的原型系统代码才能完成。

需要下载的软件包包括：

- (1) Ubuntu-12.04-desktop-amd64 镜像文件；
- (2) libvirt-1.2.0；
- (3) QEMU-KVM-1.2.0；
- (4) kvm-kmod-2.6.37；
- (5) linux 内核文件，内核版本为 2.6.37.

原型系统安装实施步骤：

- (1) 安装 Ubuntu 操作系统

原型系统使用的是 Ubuntu-12.04-desktop-amd64 版本操作系统，标准安装。

- (2) 通过下载的 2.6.37 内核更新系统内核版本。

- (3) 安装标准 libvirt-1.2.0，实现 VMI 库函数层与 QEMU-KVM 层的通信。

(4) 安装 QEMU-KVM-1.2.0，在标准 QEMU-KVM-1.2.0 上用生成的 qemu-kvm-1.2.0.patch 打补丁，然后通过 GCC 编译，安装 QEMU-KVM。

(5) 安装 KVM-2.6.37 模块，在标准 kvm-kmod-2.6.37 源代码上使用生成的 kvm-kmod-2.6.37.patch 打补丁，然后通过 GCC 编译，安装 KVM。

- (6) 安装虚拟机

虚拟机操作系统可选择 Linux 系统或 Windows 系统，标准安装。

- (7) 安装 VMI 库函数

a)将编写的各监控函数进行编译安装。

b)编写配置文件 vmi.conf，配置文件中包含虚拟机中的进程列表表项中各元素的偏移量，可使用开源工具 linux-offset-finder 和 windows-offset-finder^[45]分别获取 Linux 和 Windows 系统的相关值。

5.1.2 原型系统运行

系统安装完成后，原型系统所提供的各库函数以动态链接库(libvmi.so)的形式提供服务。用户在调用库函数时需添加 vmi.h 头文件，并在编译时调用动态链接库 lvmi。

示例如下：

通过编写 test.c 进行测试，使用 GCC 编译，如编译进程监控功能：

```
gcc process_list_test.c -o process_list_test -I /usr/include -L /usr/local/lib/ -ldvmi
```

通过命令行运行相关监控功能，如对进程监控：

```
./process_list_test
```

其他监控功能运行与进程监控类似。

5.2 原型系统测试结果

5.2.1 功能测试

我们对原型系统中的库函数进行了功能测试，各个函数功能测试结果如下。

(1) 进程打印 (Process_list)

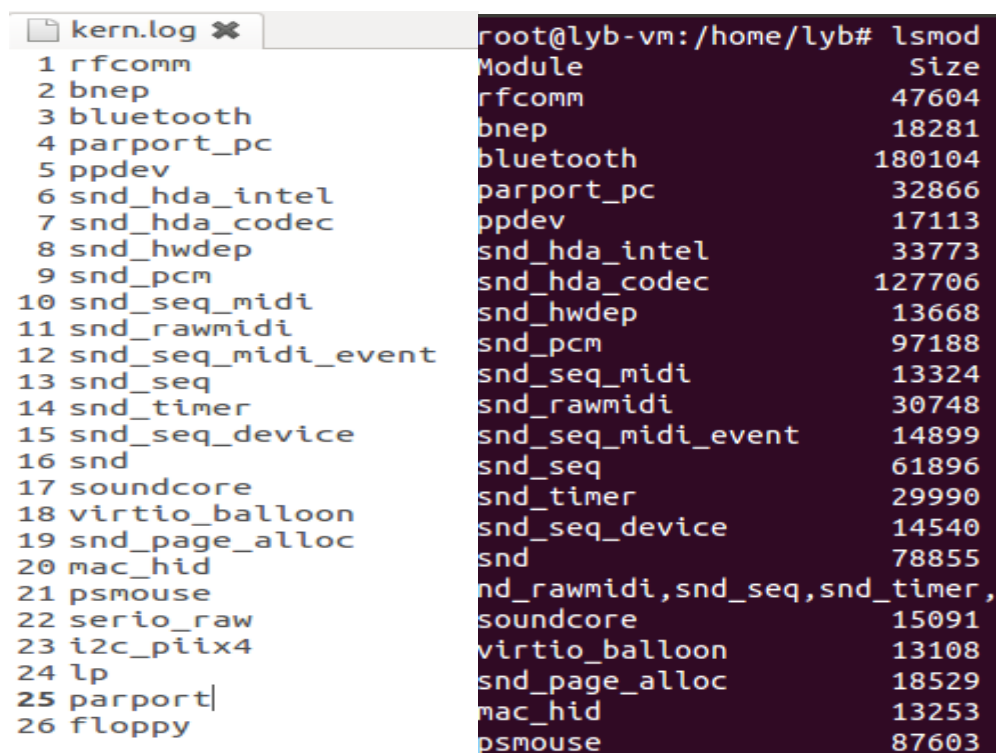
图 5.1（左侧）为借助 VMI 在主机 kern.log 日志中打印出的进程列表，图 5.1（右侧）为虚拟机内部用 ps 命令得到的进程列表。

kern.log		root@lyb-vm:/home/lyb# ps -e	
1	Process listing for VM ubuntu (id=19)	PID	TTY
2	current_process:0xffffffff81c0d020		TIME
3	Next list entry is at: ffff88003d638238		CMD
4	[0] swapper/0 (struct addr:ffffffff81c0d020)	1	?
5	[1] init (struct addr:ffff88003d638000)	2	?
6	[2] kthreadd (struct addr:ffff88003d6396f0)	3	?
7	[3] ksoftirqd/0 (struct addr:ffff88003d63ade0)	6	?
8	[6] migration/0 (struct addr:ffff88003d658000)	7	?
9	[7] watchdog/0 (struct addr:ffff88003d6596f0)	8	?
10	[8] cpuset (struct addr:ffff88003d65ade0)	9	?
11	[9] khelper (struct addr:ffff88003d65c4d0)	10	?
12	[10] kdevtmpfs (struct addr:ffff88003d65dbc0)	11	?
13	[11] netns (struct addr:ffff88003d698000)	12	?
14	[12] sync_supers (struct addr:ffff88003d6996f0)	13	?
15	[13] bdi-default (struct addr:ffff88003d69ade0)	14	?
16	[14] kintegrityd (struct addr:ffff88003d69c4d0)	15	?
17	[15] kblockd (struct addr:ffff88003d69dbc0)	16	?
18	[16] ata_sff (struct addr:ffff88003d740000)	17	?
19	[17] khubd (struct addr:ffff88003d7416f0)	18	?
20	[18] md (struct addr:ffff88003d742de0)	19	?
21	[19] kworker/0:1 (struct addr:ffff88003d7444d0)	20	?
22	[20] kworker/u:1 (struct addr:ffff88003d745bc0)	21	?
23	[21] khungtaskd (struct addr:ffff88003ceb8000)	22	?
24	[22] kswapd0 (struct addr:ffff88003ceb96f0)	23	?
		24	?

图 5.1 进程监控

(2) 内核模块罗列 (Module_list)

图 5.2（左侧）为借助 VMI 在主机 kern.log 日志中打印出的各内核模块信息，图 5.2（右侧）为在虚拟机内部用 lsmod 命令打印的内核模块信息。



Module ID	Module Name	Size (bytes)
1	rfcomm	47604
2	bnep	18281
3	bluetooth	180104
4	parport_pc	32866
5	ppdev	17113
6	snd_hda_intel	33773
7	snd_hda_codec	127706
8	snd_hwdep	13668
9	snd_pcm	97188
10	snd_seq_midi	13324
11	snd_rawmidi	30748
12	snd_seq_midi_event	14899
13	snd_seq	61896
14	snd_timer	29990
15	snd_seq_device	14540
16	snd	78855
17	soundcore	15091
18	virtio_balloon	13108
19	snd_page_alloc	18529
20	mac_hid	13253
21	psmouse	87603
22	serio_raw	
23	i2c_piix4	
24	lp	
25	parport	
26	floppy	

图 5.2 内核模块监控

(3) 系统调用监控

监控虚拟机中 39 号系统调用,当虚拟机中发生 `getpid()` 系统调用时打印出有关信息,如图 5.3 所示:

```
WARNING: netlink message dropped:
"kvm:scmon: added rule: if (rax == 39) => rcx+0 hex
"
kvm_start_scmon
WARNING: netlink message dropped:
"kvm:start_syscall_trace: system running in PAE mode
"
WARNING: netlink message dropped:
"kvm:start_syscall_trace: system running in long mode (x86_64)
"
WARNING: netlink message dropped:
"set syscall_reg
"
WARNING: netlink message dropped:
"kvm:start_syscall_trace: cpu0: GP trap set
"
WARNING: netlink message dropped:
"TEST2
"
WARNING: netlink message dropped:
"kvm:start_syscall_trace: using empty gate 0x81
"
WARNING: netlink message dropped:
"kvm:syscall_mon: rax == 27 occurred: rcx+0 hex = rcx=0x400570
"
```

图 5.3 39 号系统调用监控

监控虚拟机中所有系统调用(System call trace), 结果如图 5.4 所示。

```

WARNING: netlink message dropped:
"kvm:start_syscall_trace: using empty gate 0x81
"
WARNING: netlink message dropped:
"kvm:syscall trace(c): :0x3C10A000:0:0x3C47C067 228
"
WARNING: netlink message dropped:
"kvm:syscall trace(c): :0x3C10A000:0:0x3C47C067 254
"
WARNING: netlink message dropped:
"kvm:syscall trace(c): :0x3C10A000:0:0x3C47C067 228
"
WARNING: netlink message dropped:
"kvm:syscall trace(c): :0x3C10A000:0:0x3C47C067 7
"
WARNING: netlink message dropped:
"kvm:syscall trace(c): :0x3A3C1000:0:0x3A3C3067 0
"
WARNING: netlink message dropped:
"kvm:syscall trace(c): :0x3A3C1000:0:0x3A3C3067 228
"

```

图 5.4 所有系统调用监控

(4) 寄存器监控(Register monitor)

监控寄存器特殊值变化, 如监控 RAX 寄存器, 若 RAX 为特殊值 0x27 时, 则打印监控日志, 如图 5.5 所示。

```

[33061.001499] rax:0x27
[33061.584858] rax:0x27
[33063.993324] rax:0x27
[33067.984718] rax:0x27
[33068.984421] rax:0x27
[33070.800717] rax:0x27
[33071.811905] rax:0x27
[33071.976199] rax:0x27
[33076.788583] rax:0x27

```

图 5.5 RAX 寄存器特殊值监控

监控 RAX 寄存器, 当 RAX 有变化时打印寄存器值, 如图 5.6 所示。

```

rax:0xFFFFFFFF81C0D020
rax:0x0
rax:0xFFFFFFFF
rax:0x0
rax:0x196
rax:0xEB00
rax:0xFFFF88003FC0EB00
rax:0x11D15
rax:0x0
rax:0x1
rax:0xFFFFFFFF81C01FD8
rax:0x0
rax:0x4
rax:0x23BD4FA8
rax:0x359

```

图 5.6 RAX 寄存器跟踪

(5) 内存监控(Memory monitor)

监控虚拟机中的对地址 0x00000000229e010 的写事件，结果如图 5.7 所示。

```
loaded kvm module (kvm-kmod-2.6.37)
WARNING: netlink message dropped:
"kvm:mem_mon: address == 0x00000000229e010, size = 1
"
WARNING: netlink message dropped:
"kvm:mem_mon: write event occurred on page address == 0x00000000229e000
"
```

图 5.7 内存监控

(6) 中断监控(Interrupt trace)

监控虚拟机中所有的中断事件，结果如图 5.8 所示。

```
WARNING: netlink message dropped:
"start_interrupt_trace
"
INTR_TYPE_EXT_INTR
vector:239
INTR_TYPE_EXT_INTR
vector:239
INTR_TYPE_EXT_INTR
vector:239
INTR_TYPE_EXT_INTR
vector:239
INTR_TYPE_EXT_INTR
vector:239
INTR_TYPE_EXT_INTR
vector:239
INTR_TYPE_EXT_INTR
vector:239
INTR_TYPE_EXT_INTR
vector:239
```

图 5.8 中断监控

(7) 动态磁盘监控 (Disk monitor)

通过 VMI 系统，对虚拟机内部创建文件夹和删除文件夹事件进行监控，监控结果如图 5.9 所示。

```
WARNING: netlink message dropped:
"kvm:disk_mon: rax == 54 occurred: rdi+0 hex = rdi=0x1E8E200
"
WARNING: netlink message dropped:
"kvm:disk_mon: rax == 53 occurred: rdi+0 hex = rdi=0x24712E0
"
WARNING: netlink message dropped:
"kvm:disk_mon: rax == 53 occurred: rdi+0 hex = rdi=0x7FFFB9580E5D
"
WARNING: netlink message dropped:
"kvm:disk_mon: rax == 53 occurred: rdi+0 hex = rdi=0x7FFFB9580E62
"
```

图 5.9 动态磁盘监控

(8) 设备 I/O 监控(Device i/o monitor)

通过 VMI 系统，对虚拟机内部 I/O 事件进行监控，监控结果如图 5.10 所示。

```

port:0xc010
port:0xc010
port:0xc010
port:0xc010
port:0xc010
port:0xc010
port:0xc010
port:0xc010
port:0xc010
port:0xc010
port:0xc010
port:0xc010
port:0xc010
port:0xc010
port:0xc010

```

图 5.10 设备 I/O 监控

经过功能测试，以上各功能模块均运行正常。

5.2.2 性能损耗测试

我们进行了两项系统性能损耗测试，分别是 Lmbench 和 UnixBench 的测试。将原系统与修改后系统的性能测试结果进行比较，从而得出系统的性能损耗百分比。

(1) 利用 Lmbench 进行测试

Lmbench 是一套简单可移植的，根据 ANSI/C 标准制定的为 UNIX/POSIX 小型系统性能测评工具。Lmbench 的主要测试项目包括拷贝内存、读/写内存、管道、读取缓存文件、TCP、上下文切换、进程创建、信号处理、上层系统调用和处理器时钟比率计算等。由于 Lmbench 程序由 C 语言编写，所以该测评工具具有良好的可移植性。通过使用 Lmbench 对原型系统进行测试，可以有效的反应出原型系统的性能损耗。利用 Lmbench 对原型系统的各项监控功能测试结果如下：

a) 利用 VMI 系统对特定系统调用监控 (syscall)，LMbench 测试结果如表 5.1 所示：

表 5.1 LMbench 对特定系统调用的性能测试

测试项目	未调用 syscall	调用 syscall	损耗百分比
Basic system parameters	743.20	733.15	1.35%
Processor	452.13	429.31	5.05%
Basic integer operations	3.42	2.81	17.83%
Basic float operations	2.27	2.06	9.25%
Basic double operations	3.17	3.05	3.78%
Context switching	1.83	1.74	4.92%
Local Communication Latencies in microseconds	18.47	13.78	25.39%

File & system latencies	78.97	58.91	25.39%
Local Communication Bandwidths in MB/s	3563.12	3445.31	3.31%
Memory latencies in nanoseconds	724.12	696.15	3.86%

b) 利用 VMI 系统对所有系统调用监控 (systrace), LMBench 测试结果如表 5.2 所示:

表 5.2 LMBench 对所有系统调用监控的性能测试

测试项目	未调用 systrace	调用 systrace	损耗百分比
Basic system parameters	743.20	723.15	2.70%
Processor	452.13	398.56	11.85%
Basic integer operations	3.42	2.45	28.36%
Basic float operations	2.27	1.89	16.74%
Basic double operations	3.17	2.75	13.25%
Context switching	1.83	1.71	6.56%
Local Communication Latencies in microseconds	18.47	14.92	19.20%
File & system latencies	78.97	62.89	20.35%
Local Communication Bandwidths in MB/s	3563.12	3105.27	12.85%
Memory latencies in nanoseconds	724.12	685.15	5.38%

c) 利用 VMI 系统监控寄存器特定值 (regmon), LMBench 测试结果如表 5.3 所示:

表 5.3 LMBench 对寄存器特定值监控的性能测试

测试项目	未调用 remon	调用 remon	损耗百分比
Basic system parameters	743.20	727.15	2.16%
Processor	452.13	441.21	2.42%
Basic integer operations	3.42	3.01	11.99%
Basic float operations	2.27	2.12	6.61%

Basic double operations	3.17	2.83	10.73%
Context switching	1.83	1.65	9.84%
Local Communication Latenicies in microseconds	18.47	16.51	10.61%
File & system latenices	78.97	61.76	21.79%
Local Communication Bandwidths in MB/s	3563.12	3251.24	8.75%
Memory latencies in nanoseconds	724.12	692.41	4.38%

d) 利用 VMI 系统监控内存访问 (memon), LMbench 测试结果如表 5.4 所示:

表 5.4 LMbench 对内存访问监控的性能测试

测试项目	未调用 memon	调用 memon	损耗百分比
Basic system parameters	743.20	732.36	1.46%
Processor	452.13	433.23	4.22%
Basic integer operations	3.42	3.02	11.70%
Basic float operations	2.27	2.16	4.85%
Basic double operations	3.17	2.91	8.21%
Context switching	1.83	1.68	8.20%
Local Communication Latenicies in microseconds	18.47	17.32	6.23%
File & system latenices	78.97	63.32	20.22%
Local Communication Bandwidths in MB/s	3563.12	3425.34	3.87%
Memory latencies in nanoseconds	724.12	653.42	9.77%

e) 利用 VMI 系统监控寄存器所有变化 (regtrace), LMbench 测试结果如表 5.5 所示:

表 5.5 LMbench 对寄存器全部变化监控的性能测试

测试项目	未调用 regtrace	调用 regtrace	损耗百分比
Basic system parameters	743.20	731.23	1.61%

Processor	452.13	443.12	1.99%
Basic integer operations	3.42	3.05	10.82%
Basic float operations	2.27	2.23	1.76%
Basic double operations	3.17	2.87	9.46%
Context switching	1.83	1.70	7.10%
Local Communication Latencies in microseconds	18.47	17.17	7.04%
File & system latencies	78.97	63.90	19.08%
Local Communication Bandwidths in MB/s	3563.12	3315.47	6.95%
Memory latencies in nanoseconds	724.12	701.62	3.11%

f) 利用 VMI 系统监控特定中断事件(intmon), LMBench 测试结果如表 5.6 所示:

表 5.6 LMBench 对特定中断事件监控的性能测试

测试项目	未调用 intmon	调用 intmon	损耗百分比
Basic system parameters	743.20	733.60	1.29%
Processor	452.13	444.43	1.70%
Basic integer operations	3.42	3.15	7.89%
Basic float operations	2.27	2.18	3.96%
Basic double operations	3.17	2.90	8.52%
Context switching	1.83	1.71	6.56%
Local Communication Latencies in microseconds	18.47	17.29	6.39%
File & system latencies	78.97	62.91	20.34%
Local Communication Bandwidths in MB/s	3563.12	3425.40	3.87%
Memory latencies in nanoseconds	724.12	700.84	3.21%

g) 利用 VMI 系统监控所有中断 (intrace), LMBench 测试结果如表 5.7 所示:

表 5.7 LMBench 对所有中断事件监控的性能测试

测试项目	未调用 intrace	调用 intrace	损耗百分比
Basic system parameters	743.20	732.01	1.51%
Processor	452.13	446.73	1.19%
Basic integer operations	3.42	3.13	8.48%
Basic float operations	2.27	2.06	9.25%
Basic double operations	3.17	2.74	13.56%
Context switching	1.83	1.72	6.01%
Local Communication Latencies in microseconds	18.47	15.39	16.68%
File & system latencies	78.97	65.20	17.44%
Local Communication Bandwidths in MB/s	3563.12	3218.05	9.68%
Memory latencies in nanoseconds	724.12	705.04	2.63%

h) 利用 VMI 系统监控设备 I/O 事件(iomon), LMBench 测试结果如表 5.8 所示:

表 5.8 LMBench 对设备 I/O 事件监控的性能测试

测试项目	未调用 iomon	调用 iomon	损耗百分比
Basic system parameters	743.20	725.95	2.32%
Processor	452.13	431.04	4.66%
Basic integer operations	3.42	3.24	5.26%
Basic float operations	2.27	2.06	9.25%
Basic double operations	3.17	3.08	2.84%
Context switching	1.83	1.65	9.84%
Local Communication Latencies in microseconds	18.47	16.59	10.18%
File & system latencies	78.97	74.90	5.15%
Local Communication Bandwidths in MB/s	3563.12	3342.43	6.19%
Memory latencies in nanoseconds	724.12	702.91	2.93%

i) 利用 VMI 系统进行动态磁盘监控 (diskmon)，Lmbench 测试结果如表 5.9 所示

表 5.9 Lmbench 对动态磁盘监控的性能测试

测试项目	未调用 diskmon	调用 diskmon	损耗百分比
Basic system parameters	743.20	723.18	2.69%
Processor	452.13	422.47	6.56%
Basic integer operations	3.42	2.75	19.59%
Basic float operations	2.27	1.96	13.66%
Basic double operations	3.17	2.98	5.99%
Context switching	1.83	1.64	10.38%
Local Communication Latencies in microseconds	18.47	13.58	26.48%
File & system latencies	78.97	60.59	23.28%
Local Communication Bandwidths in MB/s	3563.12	3431.06	3.71%
Memory latencies in nanoseconds	724.12	686.53	5.19 %

(2) 利用 UnixBench 进行测试

UnixBench 是一款测试 Linux、Unix 系统基本性能的经典开源工具，测试项目包含了文件复制、管道的吞吐量、上下文切换、进程创建、系统调用、基本的 2D 和 3D 图形测试等系统基本性能，它的优点在于提供了对系统性能的一种评价体系，为系统评分，如此方便对系统作对比测试；但相比 Lmbench，Unixbench 在网络性能测试欠缺。

使用 Unixbench 对原型系统测试：在运行原型系统对虚拟机实施监控前后分别运行 UnixBench 对系统测试三次，计算性能损耗，测试结果如表 5.9 所示：

表 5.10 UnixBench 对各监控功能的性能测试

平均值 监控项	运行监控程序	未运行监控程序	性能损耗 (%)
syscall	1602.7	2006.6	20.13%
systrace	1525.6	2006.6	23.97%

regmon	1979.2	2006.6	1.37%
regtrace	1954.9	2006.6	2.55%
memon	1945.7	2006.6	3.04%
intmon	1978.9	2006.6	1.38%
intrace	2001.2	2006.6	0.27%
iomon	2002.3	2006.6	0.21%
diskmon	1561.7	2006.6	22.17%

5.2.3 系统测试结果分析

通过对原型系统的功能测试,原型系统的各个监控功能对虚拟机中运行状态监控响应正常、及时,原型系统运行平稳,达到了预期设计的各项功能目标。在虚拟机内部运行 Lmbench 和 UnixBench 以分别对原型系统进行性能测试,由测试结果可知运行原型系统各项监控功能对虚拟机系统的整体性能损耗基本可以接受。

5.3 本章小结

本章首先针搭建了基于 KVM 的虚拟化环境,并在虚拟化环境中部署原型系统,完成了对基于 KVM 的虚拟机自省系统的原型系统测试环境搭建。然后对原型系统中的各项监控功能逐一进行了测试,测试内容包括功能测试和性能测试。测试结果表明,原型系统的各项监控功能响应正常,监控效率基本可以接受,达到了初始设计目标。

第六章 总结与展望

6.1 论文工作总结

云计算和虚拟化技术作为近几年的一个研究热点方向正在快速发展,为社会提供服务,企业和个人均享受到这些技术带来的便利。然而,虚拟化作为云计算的关键基础,其自身安全问题已经变成限制云计算相关产业发展的一个重大问题,所以对于虚拟化技术以及虚拟化安全的研究工作具有重要的意义。通过利用 VMI 系统实时对虚拟化环境的运行状态进行监控,能够及时发现虚拟化环境中的异常情况,进而对异常运行情况做出处理,保证虚拟化环境的正常稳定运行。所以对于实现虚拟化安全,虚拟机自省技术具有重要的研究意义和价值。本文通过对虚拟化技术、虚拟化安全问题和虚拟机自省技术的研究分析,设计实现了一种基于 KVM 的虚拟机自省系统,主要实现在虚拟机外部对虚拟机内部运行状态的静态和动态监控,并搭建 KVM 虚拟化环境对 VMI 系统进行功能和性能测试。

本文的主要工作和贡献如下:

(1) 介绍本文的研究背景,明确本文的研究目标及意义,然后对本文的主要工作进行简要说明,最后说明本文的组织结构。

(2) 对本文涉及的相关知识进行详细阐述,例如虚拟化技术、主流虚拟机管理器(VMM)、虚拟化安全、虚拟机自省技术(VMI)以及虚拟机自省技术中存在的技术难点,如语义鸿沟等问题。

(3) 对基于 KVM 的虚拟机自省进行系统设计。通过对虚拟化环境中的安全问题的研究,明确 VMI 系统需实现的监控功能;然后根据设计目标,对虚拟机自省系统进行概要设计,给出系统总体架构图,同时根据不同监控功能对系统进行模块划分,进一步细化系统设计;最后根据各模块的具体功能,对个模块进行详细设计,给出了模块监控流程图。

(4) 原型系统实现。根据系统的概要设计,通过对 KVM 模块代码、QEMU-KVM 代码的修改,给出在 VMM 层对虚拟机内部的运行状态的监控接口。然后编写 VMI 库函数层各个功能的函数,这些函数通过利用 libvirt 库函数调用 VMM 层提供的监控接口,从而实现基于 KVM 的虚拟机自省系统。最后,对系统设计实现过程中涉及到的关键函数代码做了说明。

(5) 搭建 KVM 虚拟化环境,并在 KVM 上部署虚拟机自省系统,对原型系统进行功能验证和性能测试。测试结果表明,原型系统各个监控功能模块响应正常,系统工作稳定,总体性能可接受。

6.2 下一步工作展望

本文中所设计的虚拟机自省原型系统虽然基本达到了设计目标,但仍然存在可以在下一步工作中改进的地方:

原型系统中个别监控功能对于捕获到的虚拟机内部事件的语义还原不够全面,如在对虚拟机进行 I/O 事件监控时,原型系统只还原了虚拟机内部发生 I/O 事件设备号,没有对 I/O 事件的其他信息做进一步处理。

原型系统中个别监控功能性能损耗较高,如系统调用监控,由于使用中断捕获和陷入的技术,当虚拟机中发生系统调用事件,VMM 需暂停虚拟机,对系统调用事件进行处理,造成虚拟机性能损耗较高,需在下一步工作中进一步处理。

在原型系统中,由于 VMI 系统对虚拟机的实时监控信息存放在主机日志文件中,当监控信息较多时,将不便查看监控信息,下一步工作中可以通过在原型系统中添加数据库,用于存储监控信息。

原型系统对虚拟机的各项监控操作均由在终端执行命令行的方式实现,下一步工作中可以考率添加图形界面,方便用户实施监控,提高系统的可操作性和友好性。

参考文献

- [1] Mell, Peter, and Tim Grance. "The NIST definition of cloud computing." (2011).
- [2] T. Garnkel, M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. In the Proceeding of the 10th Annual Network and Distributed Systems Security Symposium, February 2003.
- [3] Payne, Bryan D., M. D. P. De Carbone, and Wenke Lee. "Secure and flexible monitoring of virtual machines." Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual. IEEE, 2007.
- [4] Wang, Yi-Min, et al. "Detecting stealth software with strider ghostbuster." Dependable Systems and Networks, 2005. DSN 2005. Proceedings. International Conference on. IEEE, 2005.
- [5] Jiang, Xuxian, Xinyuan Wang, and Dongyan Xu. "Stealthy malware detection through vmm-based out-of-the-box semantic view reconstruction." Proceedings of the 14th ACM conference on Computer and communications security. ACM, 2007.
- [6] B. Dolan-Gavitt, T. Leek, M. Zhivich, J Giffin, W. Leeb, "Virtuoso: Narrowing the Semantic Gap in Virtual Machine Introspection", in Proc. 2011 IEEE Symp. Security and Privacy, Oakland, CA, May 2011.
- [7] Fu, Yangchun, and Zhiqiang Lin. "Space traveling across vm: Automatically bridging the semantic gap in virtual machine introspection via online kernel data redirection." Security and Privacy (SP), 2012 IEEE Symposium on. IEEE, 2012.
- [8] Intel® Virtualization Technology (Intel® VT) [Online]. Available: <http://www.intel.com/content/www/us/en/virtualization/virtualization-technology/intel-virtualization-technology.html>
- [9] AMD Virtualization (AMD-V™) Technology [Online]. Available: <http://www.amd.com/en-us/solutions/servers/virtualization>
- [10] KVM Official Website [Online]. Available: http://www.linux-kvm.org/page/Main_Page
- [11] Linux Kernel Official Website [Online]. Available: <https://www.kernel.org/>
- [12] LMBench Official Website [Online]. Available: <http://www.bitmover.com/lmbench/>
- [13] Unixbench Official Website [Online]. Available: <https://code.google.com/p/byte-unixbench/>
- [14] Wikipedia. Virtualization [Online]. Available: <https://zh.wikipedia.org/wiki/Virtualization>
- [15] Rosenblum, M., & Garfinkel, T. (2005). Virtual machine monitors: Current technology and future trends. Computer, 38(5), 39-47.
- [16] Varian, Melinda. "Vm and the vm community: Past, present, and future." SHARE. Vol. 89.

1997.

- [17] Marshall, D. (2007). Understanding Full Virtualization, Paravirtualization, and Hardware Assist.
- [18] Bellard, Fabrice. "QEMU, a Fast and Portable Dynamic Translator." USENIX Annual Technical Conference, FREENIX Track. 2005.
- [19] Watson, J. (2008). Virtualbox: bits and bytes masquerading as machines. Linux Journal, 2008(166), 1.
- [20] King, Samuel T., George W. Dunlap, and Peter M. Chen. "Operating System Support for Virtual Machines." USENIX Annual Technical Conference, General Track. 2003.
- [21] Whitaker, Andrew, Marianne Shaw, and Steven D. Gribble. Denali: Lightweight virtual machines for distributed and networked applications. Technical Report 02-02-01, University of Washington, 2002.
- [22] Williams, D. E. (2007). Virtualization with Xen (tm): Including XenEnterprise, XenServer, and XenExpress: Including XenEnterprise, XenServer, and XenExpress. Syngress.
- [23] Fishman, A., Rapoport, M., Budilovsky, E., & Eidus, I. (2013). HVX: Virtualizing the cloud. Proc. HotCloud.
- [24] Yu, Yang. Os-level virtualization and its applications. ProQuest, 2007.
- [25] Xavier, Miguel G., et al. "Performance evaluation of container-based virtualization for high performance computing environments." Parallel, Distributed and Network-Based Processing (PDP), 2013 21st Euromicro International Conference on. IEEE, 2013.
- [26] Reshetova, Elena, et al. "Security of OS-level virtualization technologies." Secure IT Systems. Springer International Publishing, 2014. 77-93.
- [27] Kang, Junbin, et al. "MultiLanes: providing virtualized storage for OS-level virtualization on many cores." FAST. 2014.
- [28] Ren L, Zhang L, Zhang Y B, et al. Resource virtualization in cloud manufacturing[J]. Computer Integrated Manufacturing Systems, 2011, 17(3):511-518.
- [29] Gandhi, Jayneel, Arkaprava Basu, Mark D. Hill, and Michael M. Swift. "Efficient memory virtualization." MICRO, 2014.
- [30] Clark, T. (2005). Storage virtualization: technologies for simplifying data storage and management. Addison-Wesley Professional.
- [31] Wang, A., Iyer, M., Dutta, R., Rouskas, G. N., & Baldine, I. (2013). Network virtualization: technologies, perspectives, and frontiers. Lightwave Technology, Journal of, 31(4), 523-537.
- [32] Reiersen, K., Gora, F., Kekeh, C. K., Morgan, P. A., Rovine, J. W., & Sheehan, J. M., et al. (2013). Application virtualization. US, US8413136 B2.
- [33] Matthews, J. N., Dow, E. M., Deshane, T., Hu, W., Bongio, J., Wilbur, P. F., & Johnson, B.

- (2008). Running Xen: a hands-on guide to the art of virtualization. Prentice Hall PTR.
- [34] Barham P, Dragovic B, Fraser K, et al. Xen and the art of virtualization[J]. ACM SIGOPS Operating Systems Review, 2003, 37(5): 164-177.
- [35] Kivity, A., Kamay, Y., Laor, D., Lublin, U., & Liguori, A. (2007, July). kvm: the Linux virtual machine monitor. In Proceedings of the Linux Symposium (Vol. 1, pp. 225-230).
- [36] Litty, L., Lagar-Cavilla, H. A., & Lie, D. (2009, May). Computer Meteorology: Monitoring Compute Clouds. In HotOS.
- [37] Chen, Peter M., and B. D. Noble. "When Virtual Is Better Than Real." Hot Topics in Operating Systems, Workshop on IEEE Computer Society, 2001:0133.
- [38] Jain, Brijnesh, et al. "Sok: Introspections on trust and the semantic gap." Security and Privacy (SP), 2014 IEEE Symposium on. IEEE, 2014.
- [39] Inoue, H., Adelstein, F., Donovan, M., & Brueckner, S. (2011, June). Automatically Bridging the Semantic Gap using C Interpreter. In Proc. of the 2011 Annual Symposium on Information Assurance .
- [40] Saberi, Alireza, Yangchun Fu, and Zhiqiang Lin. "Hybrid-bridge: Efficiently bridging the semantic gap in virtual machine introspection via decoupled execution and training memoization." Proceedings of the 21st Annual Network and Distributed System Security Symposium (NDSS'14), San Diego, CA. 2014.
- [41] Deshane, T., Shepherd, Z., Matthews, J., Ben-Yehuda, M., Shah, A., & Rao, B. (2008). Quantitative comparison of Xen and KVM. Xen Summit, Boston, MA, USA, 1-2.
- [42] Quynh, Nguyen Anh. "Operating system fingerprinting for virtual machines." Proc. DEFCON 18 (2010).
- [43] Bovet D P, Cesati M. Understanding the Linux kernel[M]. " O'Reilly Media, Inc.", 2005.
- [44] Intel Corporation. Intel® 64 and IA-32 Architectures Software Developer's Manual Vol 3: System Programming Guide. 2014
- [45] Payne B.D.. XenAccess Library [Online]. Available: <https://code.google.com/p/xenaccess/>

致谢

时光飞逝，两年半的研究生生活即将结束，在此论文完成之际，我要对所有在学习和生活上给予我指导、帮助、关心和支持的老师、同学、朋友和亲人致以最衷心的感谢和最真诚的祝福！

首先我要感谢我的研究生导师李金库副教授。在我攻读硕士学位期间，得到了李老师的悉心指导。李老师渊博的学识、严谨的治学态度、高深的学术造诣、勤恳的工作作风和朴实的生活方式都深深地感染了我，是我以后工作和生活的榜样。在学习和科研工作中，每当我遇到难题感到疑惑时，李老师总是认真耐心的为我答疑解惑。在生活上，每当我遇到困难时，李老师都给予了无微不至的关心和帮助。在此，对李老师致以最诚挚的感谢和祝福！

感谢我的师兄胡祥、程坤和王维明，感谢你们在我研究生阶段给予的帮助和指导，谢谢你们！

感谢我的师弟吴晓润和师妹付丽蓉，感谢他们在项目过程中的努力付出。你们的付出使项目能够按时交付，与你们一起学习工作的日子很充实很快乐。在这里，谢谢师弟师妹，祝愿你们学业有成！

感谢我的研究生室友杨佳旭、豆超和孔亚兵同学，与你们一起度过研究生生活是我们的缘分，感谢你们在我有困难时给予的关心和帮助，谢谢你们！

感谢我的本科室友林日三同学，与你本科四年室友，研究生两年半同学，谢谢你在近七年的时间里对我的帮助和照顾，当我研究生阶段遇到困难时，谢谢你对我的关心和鼓励，我会永远珍惜这份珍贵的友谊。在此，对你表示最真心的谢意，祝你前途似锦！

感谢我的女朋友。在研究生学习和生活中，你一直默默的支持、关心、鼓励着我。当我遇到困难时，你始终陪伴着我，关心鼓励我，不断的给予我信心，使我保持乐观积极的心态。

最后感谢我的家人。感谢你们无私的付出与支持，感谢你们给予的一切。正是你们的付出、培养和支持，我才有机会学习和成长，拥有直面困难的勇气，并在人生的道路上勇往直前。谢谢你们！

作者简介

1. 基本情况

李永波，男，陕西咸阳人，1989年8月出生，西安电子科技大学计算机学院计算机技术专业2013级硕士研究生。

2. 教育背景

2009.08~2013.07 西安电子科技大学，本科，专业：计算机科学与技术

2013.08~至今 西安电子科技大学，硕士研究生，专业：计算机技术

3. 攻读硕士学位期间的研究成果

华为公司合作项目，虚拟机内核安全保护与自省技术研究，2013.10-2015.6，已解题，项目负责人。



西安电子科技大学
XIDIAN UNIVERSITY

地址：西安市太白南路2号

邮编：710071

网址：www.xidian.edu.cn