软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2014,25(10):2235−2250 [doi: 10.13328/j.cnki.jos.004678]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel/Fax: +86-10-62562563

# VM 内部隔离驱动程序的可靠性架构*

郑 豪[1], 董小社[1], 王恩东[2], 陈宝可[1], 朱正东[1]

[1](西安交通大学 计算机科学与技术系,陕西 西安 710049)

[2](高效能服务器和存储技术国家重点实验室,山东 济南 250013)

通讯作者: 朱正东, E-mail: zdzhu@mail.xjtu.edu.cn

**摘 要**: 利用虚拟化技术来整合资源已成为高性能服务器提高资源利用率的重要手段,虚拟化技术的可靠性对于高性能服务器所提供服务的质量至关重要.然而,驱动故障严重影响了虚拟机中操作系统的可靠性,也同样影响到整个服务器的可靠性.为此,提出一种在虚拟机内部通过隔离故障驱动程序来提高虚拟机可靠性的架构,该架构通过监视驱动程序所使用的内存信息来建立驱动可写权限的授权表,并在虚拟机监视器中设置虚拟机内核空间对应影子页表的写保护来捕获虚拟机的写操作,进而结合授权表判断被隔离驱动程序写操作的正确性.目前,该架构能够在无需修改驱动程序的情况下,在虚拟机内部实现对驱动程序的隔离.实验结果表明:该架构可以隔离 84.63%的注入故障造成的系统崩溃失效,并且对于驱动性能的影响小于 20%,提高了虚拟化环境的可靠性.

**关键词**: 虚拟化;可靠性;驱动隔离

**中图法分类号**: TP316

中文引用格式: 郑豪,董小社,王恩东,陈宝可,朱正东.VM 内部隔离驱动程序的可靠性架构.软件学报,2014,25(10):2235−2250. http://www.jos.org.cn/1000-9825/4678.htm

英文引用格式: Zheng H, Dong XS, Wang ED, Chen BK, Zhu ZD. Reliability architecture to isolate the driver inside the VM. Ruan Jian Xue Bao/Journal of Software, 2014,25(10):2235−2250 (in Chinese). http://www.jos.org.cn/1000-9825/4678.htm

## Reliability Architecture to Isolate the Driver Inside the VM

ZHENG Hao[1], DONG Xiao-She[1], WANG En-Dong[2], CHEN Bao-Ke[1], ZHU Zheng-Dong[1]

[1](Department of Computer Science and Technology, Xi'an Jiaotong University, Xi'an 710049, China)

[2](State Key Laboratory of High-End Server & Storage Technology, Ji'nan 250013, China)

Corresponding author: ZHU Zheng-Dong, E-mail: zdzhu@mail.xjtu.edu.cn

**Abstract**: Using virtualization technology to integrate resources has become an important mean to improve the resource utilization of current high-performance servers. Thus the reliability of virtualization technology is very crucial to the service quality of high-performance server. However, the driver fault greatly impacts not only the reliability of operating system inside the virtual machine but also the reliability of the servers. In light of issue, this paper presents a driver isolation architecture inside the virtual machine to improve its reliability. It establishes the authorization table by monitoring the memory information which are used by the driver, captures the driver's write operations by setting the write protection of the shadow page table corresponding to the kernel space of the virtual machine, and judges the correctness of write operations of the isolated driver with the authorization table. Currently, the architecture can isolate drivers inside the virtual machine without modifying them. Experimental results show that the architecture can isolate 84.63% injection faults which cause system crashes with the performance loss less than 20%, and therefore effectively improves the reliability of the virtualization environment.

**Key words**: virtualization; reliability; driver isolation

* 基金项目: 国家高技术研究发展计划(863)(2008AA01A202, 2012AA01A306); 国家科技攻关计划(2011BAH04B03); NSF 青年基金(61202041)

收稿时间: 2014-01-25; 修改时间: 2014-07-07; 定稿时间: 2014-07-31

当前,高性能服务器常常通过虚拟化技术为更多的用户提供服务,以提高服务器资源的利用率.用户可以重用已有的操作系统和应用程序来定制自己的虚拟机(VM)环境,对于不同的 VM,都认为它拥有整个系统的所有资源,而且 VM 之间具有相互隔离性.然而在 VM 内部,占操作系统代码比重最大、由第三方开发、缺乏完善测试的驱动程序所存在的可靠性问题[1−3]严重影响着 VM 内操作系统的可靠性.考虑到驱动程序故障大多为瞬时故障[4],如果存在一种在 VM 内部实现对故障驱动进行隔离的架构,就可以提高 VM 内操作系统的可靠性.同时,由于已有驱动程序数量庞大,驱动隔离架构需确保对驱动程序的兼容性,即:在不修改驱动程序的情况下,防止 VM 内部驱动程序故障引起整个 VM 内核的崩溃,从而提高 VM 服务的可靠性.

由于 VM 本身具有隔离特性,当前存在多种利用虚拟化技术的隔离性[5−8]来提高操作系统可靠性的方法.虽然这些方法利用了虚拟化技术可重用性的优点,可实现对已有驱动程序的兼容,但这些方法只是提供了一个替代整机承受系统崩溃的 VM 实例,并未解决 VM 内部操作系统的可靠性问题.如果 VM 内核的驱动出现错误,仍然会引起 VM 的崩溃,从而造成提供给用户的服务中断,甚至影响整机的可靠性.而且,为隔离一个驱动程序而运行一个单独的 VM 实例,也会带来不小的资源浪费.此外,还存在多种为了解决驱动程序所引起的操作系统可靠性问题的方法,但都存在某些局限性,不适合直接应用到虚拟化环境中.例如,微内核架构[9−11]和用户态驱动架构[12,13]与传统的操作系统架构和驱动架构不兼容,而且其性能损失较为严重.硬件架构的方法[14−16]往往较为复杂,开发难度较大,而且有些对底层硬件有特殊要求,有些需要修改驱动,也存在较大的性能损失.类型安全语言[17,18]一般都需要修改或者重写驱动程序,同样存在兼容性问题.

本文提出了一个通过在 VM 内隔离驱动程序,以提高 VM 内部操作系统可靠性的架构.该架构针对造成操作系统崩溃的最主要驱动故障(空指针和无效指针等非法写操作)[1−3],研究这类写操作故障的检测、隔离和恢复方法.通过监视 VM 中驱动程序的运行情况,建立驱动程序可写内存范围的授权表;在虚拟机监视器(VMM)中将 VM 的影子页表设置为只读,捕获驱动程序的写操作;结合授权表来判断驱动程序写操作的正确性,在发现驱动程序故障时,及时卸载驱动程序,防止驱动程序故障的扩散.目前,该架构已在 KVM 和 Linux 2.6.28.10 的环境中实现了对 6 种不同驱动程序的隔离.实验结果表明:本架构可以有效检查和隔离驱动程序错误,从而提高 VM 内部操作系统的可靠性.

本文第 1 节分析提高操作系统可靠性的研究现状.第 2 节介绍实现本架构的方法.第 3 节介绍架构的功能设计.第 4 节为本架构的实现.第 5 节是本架构的测试与效果分析.最后是结论和后续工作.

## 1 相关工作

利用虚拟化技术来隔离驱动程序的方法,如 Xen[5,6],L4Ka[7]和 iKernel[8]是将驱动隔离到独立的 VM 实例中,利用 VM 实例本身的隔离性来隔离驱动程序.然而,这种方法只是利用 VM 替代整机来承受可能出现的系统崩溃,从而换取整机可靠性的提高,对于 VM 内部操作系统的可靠性并未提高.为了避免因一个 VM 内部驱动程序故障引起的 VM 崩溃造成的服务中断,常常为每个驱动程序都建立一个 VM 实例来对外提供服务,这样也造成了较大的性能损失.VirtuOS[19]进一步划分单个 VM 内的操作系统,将不同功能隔离到不同的 VM 功能实例中,从而限制 VM 内核故障的影响范围,然而在每个 VM 功能实例内部驱动故障问题仍然存在.

硬件隔离方法,如 Palladium[20]利用硬件的段保护机制实现对内核态或用户态扩展程序的隔离,但是该方法的编程模型较为复杂.FPD[21]采用硬件页保护机制实现对用户态扩展程序的隔离,提供判断策略来决定用户态的各种系统调用是否合法,但该方法未实现对内核态扩展程序的隔离.Nooks[15]通过复杂的同步和更新机制为驱动程序建立私有页表来限制其写权限,同时确保所有驱动和内核的交互都需要切换页表和堆栈,其开发难度大,不易于隔离新驱动程序,而且性能开销较大.Mondrix[14]混合软、硬件隔离技术,提供 32 位细粒度的权限控制以及保护域间切换的权限控制,该方法同样存在效率问题,而且需要特定硬件,存在兼容性问题.

软件隔离方法,如 XFI[22]以低性能开销隔离简单的内核扩展程序,但是不能处理传统操作系统中的复杂扩展程序.BGI[23]通过手动注释各种可能内核和驱动间接口的方法扩展了 XFI,从而可以处理更复杂的驱动接口,并利用授权表来限制驱动的运行,但它依赖于 Windows 驱动模型良好定义的 API.LXFI[24]将 BGI 扩展到了复杂

的 Linux 驱动接口,并可通过代理者来划分共享模块的特权级.FGFI[25]以单一入口粒度隔离驱动程序,并使用已有的电源管理代码来保存和恢复设备状态.但是这些方法都需要程序员清晰地注释内核和驱动接口,并修改驱动程序来添加运行时检查,使得对驱动程序的透明性较差.

SPIN[17],Sigularity[18]等以类型安全语言编写的操作系统和内核扩展程序可提供强大的内存隔离保证,但需要重写内核和扩展程序,并且不兼容已有的操作系统及其大量扩展程序,影响了这类方法的推广.SafeDrive[26]通过类型推断和程序员注释,由 Deputy 编译器自动生成运行时检查来保证驱动所操作的各种数据类型的安全,但它需要修改驱动程序的源代码,影响了对驱动程序的透明性,而且该方法的隔离性较差.Dingo[27]利用软件协议说明语言定义驱动的正确行为,并能有效检查驱动的同步故障和违反规范的行为失效,但同样也需要重写驱动程序.

有些系统将驱动程序隔离到独立的用户态进程地址空间,例如微内核[9−11]以及多种用户态驱动架构[12,13],这样,驱动程序的故障虽然只会造成对应的用户态进程的失效,不会造成系统崩溃,但微内核技术效率低,需要在用户态和内核态之间频繁地传递大量数据,对于高速硬件设备将造成严重的延时和性能损失;与传统宏内核操作系统不兼容,为了在传统操作系统中支持用户态驱动,需要修改内核并重写驱动.

综上分析可知:硬件隔离方法基本上都需要修改操作系统内核,其中有些方法还需要修改驱动程序,甚至要求特殊的硬件架构,开发难度较大,且大多数不能实现对驱动程序的透明性;软件隔离方法的安全隔离性能较差,且需要修改驱动程序,影响与原有驱动程序的兼容性;语言方法虽然性能较好,但需要修改驱动程序,同样对驱动程序不透明;微内核和用户态驱动存在与传统操作系统和驱动严重的兼容问题,且性能较差;当前,虚拟化技术虽然具有可重用性的优点,但是没有真正解决 VM 中驱动程序的可靠性问题.为此,本文设计了一种在 VM 内的隔离故障驱动程序的架构,该架构能够实现在 VM 内部对驱动程序的细粒度隔离,有效地捕捉驱动程序的异常,避免驱动程序故障造成 VM 崩溃以及 VM 所提供服务的中断.

## 2 原 理

在全虚拟化技术中,客户机(即 VM 实例)虚拟地址到宿主机物理地址的转换需要通过两级页表映射实现,如图 1 所示.第 1 级页表映射位于 VM 内,为客户机虚拟地址到客户机物理地址的地址映射,如图 1 中的 VM 页表 $g$ 所示;第 2 级页表映射位于 VMM 中,为客户机物理地址到宿主机机器地址的地址映射,如图 1 中的影子页表 $f$ 所示.如果客户机要访问物理内存的内容,需复合两级页表映射 $f$∘$g$,并将其交给 MMU 来完成相应的操作.

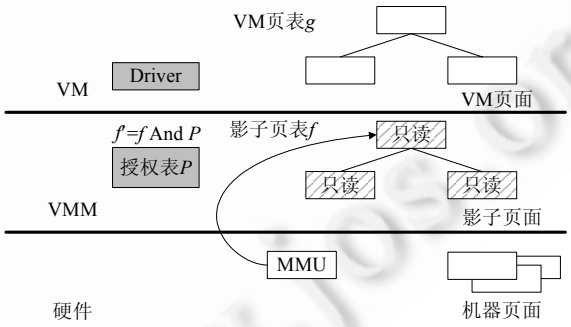



Fig.1 Two-Level page table mapping schematic in the full virtualization technology

图 1 全虚拟化技术中两级页表映射示意图

由以上分析可知:VM 内发生的任何写操作,都要经过 VMM 中第 2 级页表(也称影子页表)的地址映射 $f$ 才能最终执行.即,真正记录写操作物理地址的页表位于 VMM 中.如果访问的地址在影子页表中不存在或为只读,则 VM 内的写操作将触发 VMM 的缺页异常.VMM 的缺页异常将会根据引起缺页的具体原因进行相应的处理.因此,通过对影子页表的写权限进行适当的设置就可以捕获到写操作,以便进一步在缺页异常中验证写操作的

正确性.

因此,本文驱动隔离架构的主要原理如下:

(1) 在驱动程序进行写操作前,将被隔离驱动所在 VM 的所有内核空间对应的影子页表 $f$ 设置为只读(图 1 中斜线所示,后面简称 VM 的影子页表),则可在 VMM 中捕捉到驱动的写操作;

(2) 在 VMM 中建立一个与被隔离驱动程序对应的记录 VM 内部可写内存范围的授权表 $P$,如图 1 阴影所示;

(3) 当 VMM 的缺页异常捕捉到驱动程序的写操作后,结合授权表 $P$ 来判定驱动程序写操作的正确性;

(4) 如果正确,则允许写操作;否则,隔离驱动程序的写操作故障,防止驱动故障的扩散.

如图 1 所示,此时写操作目标地址的第 2 级页表地址映射由第 2 级页表本身的地址映射 $f$ 和授权表 $P$ 联合实现,即图中的 $f'$.

## 3 架 构

本架构基于全虚拟化技术实现,其核心思想是:当驱动程序通过 VMM 中的影子页表实现写操作时,结合授权表来判断其写操作的正确性.它是通过驱动监视、隔离环境和错误处理三大功能模块来实现,如图 2 所示:驱动监控功能负责将驱动的运行状态和使用的内存信息等实时报告给隔离环境功能;隔离环境功能根据驱动所使用内存信息建立驱动程序对应的授权表,并根据驱动的运行状态设置 VM 对应的影子页表,捕获驱动程序的写操作,建立起驱动隔离运行环境,在捕获到驱动写操作后,判断驱动写操作的正确性;当驱动出现写操作错误时,错误处理功能则负责停止故障驱动的工作,并释放驱动资源,以便移除故障驱动.本架构无需修改驱动程序,可确保对驱动程序的兼容性.

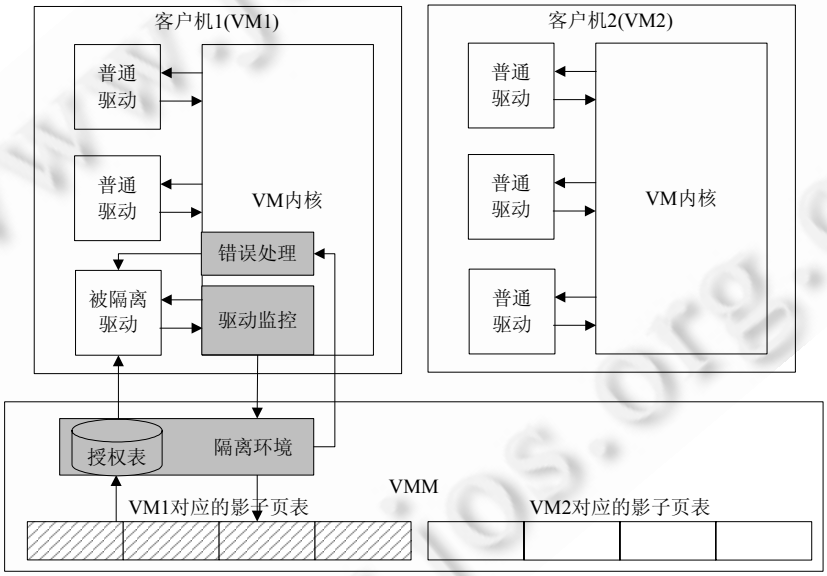



Fig.2 Architecture of driver isolation inside VM

图 2 VM 内隔离驱动程序的功能架构

### 3.1 驱动监视

驱动监视功能负责监视驱动程序运行中的各种信息,主要包括:驱动运行所需的内存信息、运行状态信息以及用于区分写操作指令来源的可信内核范围信息等.

首先,为了构建图 1 所示的授权表 $P$,需要监视驱动程序正常运行所需的内存范围,这部分内存是驱动程序

具有写权限才能正常运行的内核对象集合.具体内核对象包括:

(1) 堆内存:驱动通过内核的内存管理函数申请分配/释放的内存;

(2) I/O 内存:驱动通过内核的端口映射函数映射/释放的内存;

(3) 栈内存:调用驱动函数的进程所在的栈,以及驱动程序本身所创建的内核线程所在的栈;

(4) 授予内存:内核传递给驱动函数使用的内核对象,或者驱动需要操作的全局内核变量.

其次,为了确保驱动程序写操作在执行前被捕获,以便进行正确性判断,需要及时地将图 1 中的影子页表 $g$ 设置为只读.这要求本架构能及时捕获驱动程序的各种运行状态消息,这些状态信息包括:

(1) 被隔离的驱动程序加载进入和卸载退出内核的消息,如图 3(a)、图 3(b)所示,前者用于通知 VMM 中的隔离环境功能进行授权表等隔离信息的创建和销毁工作;

(2) 内核开始调用和结束调用驱动函数的消息,如图 3(c)、图 3(d)所示,后者用于通知隔离环境功能适时地进行影子页表的设置工作,以便捕获驱动程序的写操作.

最后,为了便于在 VMM 的缺页异常中判断捕获的 VM 写操作的正确性,还需要获取和记录可信内核范围等其他信息.相对于被隔离驱动程序,VM 内核可信任的地址范围包括 VM 内核代码地址范围和未被隔离的其他驱动的地址范围.该信息用于辅助隔离环境功能进行写操作来源判断,以便进行适当的处理.

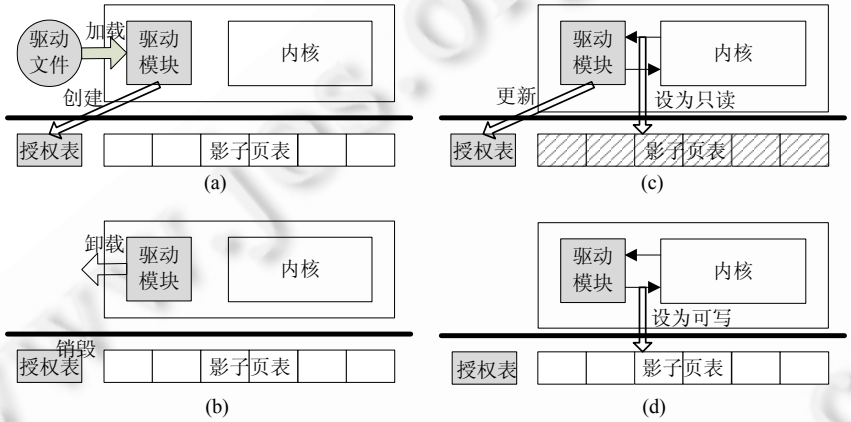



Fig.3 Schematic view of driver running states

图 3 驱动程序的运行状态信息示意图

### 3.2 隔离环境

隔离环境为驱动程序提供一种限制权限的隔离运行环境,由被隔离驱动程序对应的授权表和设置了写保护的被隔离驱动所在 VM 的影子页表组成.它只提供驱动正常运行的最小内存资源.

授权表是被隔离驱动程序对 VM 内核地址空间具有写访问权限的地址集合.被隔离驱动程序只对属于授权表的 VM 内核空间具有写权限,对 VM 内核空间的其他部分只具有读权限.如图 3(a)~图 3(c)所示,授权表在驱动程序加载进内核时建立,在卸载出内核时销毁,并在运行过程中不断地被更新.通过接收驱动监视功能发出的被隔离驱动使用的内存信息,就可以实时地更新被隔离驱动程序可写内存范围的授权表.通过授权表,可以实现对 VM 内核任意粒度的内存保护.

为了捕获驱动程序的写操作,需要在驱动程序被调用前设置驱动程序所在 VM 对应的影子页表为只读.如图 3(c)、图 3(d)所示,通过接收驱动监视功能发出的驱动调用状态信息,隔离环境可及时设置 VM 对应的影子页表为只读,从而捕获 VM 的写操作.同样,在驱动程序被调用过程中,为了可持续捕获驱动程序的写操作,被允许的驱动程序写操作结束后,需将开放的影子页面的写权限重新设置为只读.为了避免将所有的影子页面设置为只读带来的性能损失,本文使用影子页面缓存的方法来优化影子页面设置的时机.其主要思想是:将最近访问到

的影子页面先送入到缓存中,在延时一定时间后再将其设置为只读,以避免常用影子页面的写权限被反复设置(具体见第 4.2 节).

在驱动运行过程中,捕获到驱动程序的写操作后,结合授权表判断其写操作的正确性,具体流程如图 4 所示.首先,根据驱动监视功能报告的可信内核范围,判断写操作来源.如果写操作来自被隔离驱动,则结合授权表继续判断驱动程序对于写操作地址是否具有写权限.如果驱动具有写权限,则将写操作地址对应的影子页面改为可写并执行写操作.在写操作结束后,将该页面送入影子页面缓存中,以便将来重新设置其写权限为只读.如果驱动不具有写权限,则跳过该指令,并报告被隔离驱动错误.如果写操作指令来自内核,则直接允许写操作,不需要结合授权表进行上述判断.

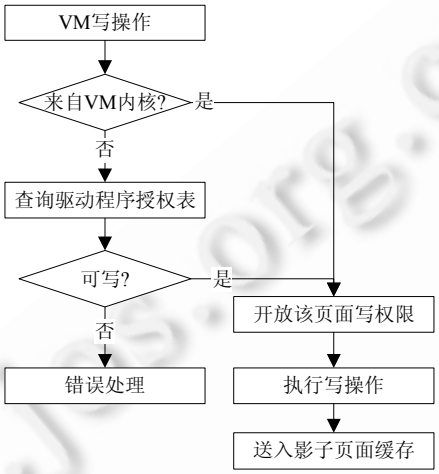



Fig.4 Process of correctness judgement of driver write operation

图 4 驱动程序写操作正确性判断流程

### 3.3 错误处理

错误处理功能主要用于在驱动程序出错后对驱动程序进行必要的处理,具体功能包括:跳过驱动程序的写操作、停止驱动程序进程、释放驱动程序所使用资源和卸载出错驱动程序等.

首先,在隔离环境检测到驱动故障后,向 VM 注入中断唤醒错误处理功能.在错误处理功能正式移除故障驱动之前,为了避免进一步执行驱动程序,VMM 将跳过故障驱动的指令.同时,为了避免影子页面缓存泄漏对驱动写操作的捕获,在检测到驱动故障后,VMM 设置整个 VM 对应的影子页面为只读,并停用影子页面缓存.

其次,故障驱动所在 VM 在接收到 VMM 发出的驱动故障信号后,需及时退出使用驱动程序的进程.为此,在内核中需要对驱动函数的调用进行适当的修改.主要包括以下步骤:

(1) 在调用驱动程序之前,将各寄存器的值以及驱动的出错返回地址记录到驱动程序栈中;

(2) 如果驱动程序正常运行,在驱动调用完成之后撤销上述记录信息;

(3) 如果驱动程序运行时出现错误,则恢复各寄存器值,并直接跳转到出错返回地址.

由于在检测到驱动错误时将直接跳转到出错返回地址,而不是按照正常的流程退出,此时进程可能持有部分锁资源.为了避免系统死锁,首先需要卸载驱动程序使用的锁资源,然后再卸载其他资源.各种内存资源是用于建立隔离环境中授权表的资源,这部分是驱动使用的主要资源;而驱动的各种注册信息是内核调用驱动的接口,通过移除这些信息就可避免内核进一步使用故障的驱动程序.

释放了故障驱动所使用的资源后,就可移除故障驱动模块.由于此时驱动程序已经出现错误,不能直接调用驱动程序自身的卸载函数来移除驱动模块,而要通过停止驱动程序,然后直接调用内核的模块卸载函数来释放驱动模块.如果需要,可以在卸载完故障驱动之后重新加载新的驱动程序,以便继续提供相应的服务.

## 4 实 现

目前,本架构已在以 KVM 作为底层 VMM,以 Linux 2.6.28.10 作为 VM 内核的环境中实现了上述功能,并已实现对多种驱动程序的隔离.

### 4.1 驱动监视

驱动监视功能需要获取驱动所使用的各种资源信息、驱动运行状态以及可信内核范围.

#### 4.1.1 内存资源

驱动所使用的内存信息包括堆内存、I/O 内存、栈内存和授予内存的信息,下面通过实例来说明具体的内存信息获取过程.驱动涉及的堆内存是通过调用内存管理函数进行操作的内存,包括如下两个部分:

(1) 通过通用内存管理函数分配/释放的内存,这部分内存最为常见,如内存高速缓存的分配/释放函数 *kmalloc* 和 *kfree*;

(2) 通过某些特定数据结构的内存管理函数分配/释放的内存,如 *usb* 驱动中对 *urb* 结构体的分配/释放函数 *usb_alloc_urb* 和 *usb_free_urb*.

为了避免过多地修改 VM 内核,本框架选取这些内存管理函数都会调用到的底层内存管理函数作为驱动所使用内存信息的获取点,如在*__cache_alloc* 函数中获取上述 *kmalloc* 和 *usb_alloc_urb* 分配的内存信息.如图 5 中第 9 行所示,通过向*__cache_alloc* 函数添加的注入内存信息语句来捕获驱动申请的 *slab* 内存信息.

```
1   /* linux2.6.28.10/mm/slab.c */
2   static __always_inline void * __cache_alloc(struct kmem_cache *cachep,gfp_t flags,void *caller)
3   {
4       ... ...
5       local_irq_save(save_flags);
6       objp=__do_cache_alloc(cachep,flags);
7       local_irq_restore(save_flags);
8       ... ...
9       DRIVER_AUTH_ADD(cachep,obj_size(cachep));
10      return objp;
11  }
```

Fig.5 Obtain the heap memory information

图 5 堆内存信息的获取

与堆内存类似,I/O 内存是通过调用内核的 I/O 端口映射函数来申请,从而将硬件设备的资源映射到内存中.如,通过 *ioremap_nocache* 函数将硬件寄存器和内存等 I/O 空间资源映射到内核的内存中.同样,可在相应函数中添加注入语句实时地获得 I/O 内存的使用情况.

内核通过执行驱动接口中的函数指针字段实现对驱动的调用,如图 6 中第 11 行所示.栈内存主要用于驱动函数内部变量的运算,其一般为一个或多个内存页面.通过内核的一些专用函数(如 Linux 系统中的 *current_thread_info*(·)函数),或者获取驱动函数的本地变量地址所在的页面,就可获取调用驱动函数的当前进程的栈资源.为保证驱动的正常运行,在执行接口函数前需要获取当前进程的内核栈,并将栈信息注入给隔离环境,如图 6 中第 5 行所示.驱动程序可能被多个进程调用,这些不同的进程栈都需要被注入到授权表中.

另外,内核还会通过这些接口函数的参数传递内核数据结构给驱动使用,这些授权内存信息也需要注入给隔离环境.这类内存常常以指针形式出现在驱动接口函数的参数中.如图 6 中第 6 行所示,内核通过 *cmd* 指针参数传递一个 *struct scsi_cmnd* 类型的指令给底层驱动,并授权驱动程序操作该内存对象.除了指针参数指向的内存对象外,该内存对象的指针字段所指向的其他内存对象也可能被驱动使用,也需要一并注入.如图 6 中第 7 行、第 8 行所示,*struct scsi_cmnd* 结构体的 *sense_buffer* 和 *cmnd* 字段所指向的数据结构.由于驱动的栈内存和授权内存都涉及到驱动的接口函数,因此,同一接口函数可能同时涉及到这两类内存信息的注入.

以 *usb-storage* 为例,表 1 所示为监视各种内存资源需要修改的内核位置.可以看出:通过在底层函数监视堆内存,只需修改两个函数就可实现对 U 盘驱动程序所使用的堆内存进行监视.由于在 Linux 2.6 中驱动程序的功

能较为强大,使得驱动程序的接口也较为复杂.相对于监视堆内存,监视栈内存需要修改的函数较多,但其数量仍然很有限.其中,部分函数还需同时添加对授予内存的监视代码.总体来说,利用本框架隔离一个驱动程序,需要修改的内核函数很少,可以很容易地实现对驱动程序的隔离.

```
1    /* linux2.6.28.10/drivers/scsi/scsi.c */
2    int scsi_dispatch_cmd(struct scsi_cmnd *cmd)
3    {
4        ... ...
5        DRIVER_AUTH_ADD(current_thread_info(·),THREAD_SIZE);
6        DRIVER_AUTH_ADD(cmd,sizeof(struct scsi_cmnd));
7        DRIVER_AUTH_ADD(cmd→sense_buffer,SCSI_SENSE_BUFFERSIZE);
8        DRIVER_AUTH_ADD(cmd→cmnd,MAX_COMMAND_SIZE);
9        DRIVER_STATE_BEGIN_CALL(·);
10       DRIVER_STACK_SAVE(·);
11       rtn=host→hostt→queuecommand(cmd,scsi_done);
12       DRIVER_STACK_QUIT(·);
13       DRIVER_STATE_FINSH_CALL(·);
14       ... ...
15   }
```

Fig.6 Obtain the stack and grant memory information

图 6 栈和授权内存信息的获取

**Table 1** Modification places for monitoring memory usage of *usb-storage*

**表 1** 监视 *usb-storage* 内存资源需要修改的内核位置

| 函数 | 监视的内存类型 |
|---|---|
| *__cache_alloc* | 堆内存 |
| *__cache_free* | 堆内存 |
| *kthread_create* | 栈内存 |
| *kthread_stop* | 栈内存 |
| *usb_probe_interface* | 栈内存和授予内存 |
| *usb_unbind_interface* | 栈内存和授予内存 |
| *scsi_dispatch_cmd* | 栈内存和授予内存 |
| *scsi_alloc_sdev* | 栈内存和授予内存 |
| *ioctl_probe* | 栈内存 |
| *scsi_add_lun* | 栈内存 |

4.1.2 资源跟踪

驱动程序在运行过程中所使用的内核资源,最主要的是第 4.1.1 节所介绍的内存资源.除此之外,还有许多其他内核资源,例如自旋锁和互斥锁等.为了便于驱动出现错误后移除故障驱动,同样需要实时记录这些内核资源.对这些资源的跟踪与第 4.1.1 节所述的内存资源的跟踪方法一致,都将被记录到一个驱动资源使用的哈希表中,以便错误处理时使用.所不同的是,内存资源需要被注入到 VMM 中,以便建立被隔离驱动程序的授权表,而其他内核资源则不需要注入.

4.1.3 驱动状态

为了获取被隔离驱动进入和退出内核的信息,需要修改内核的加载/卸载系统调用.如图 3 所示,对于加载函数,在加载成功之后,需要为被隔离驱动程序创建相应的隔离域信息,并发出相应的建立授权表的指令;对于卸载函数,则做相反操作.

为了及时捕获驱动程序的写操作,需要在驱动程序被调用之前,通知 VMM 驱动程序被调用,并将 VM 的影子页表设置为只读;退出时,则需要做相反操作.内核对驱动程序的调用都是通过对标准接口函数调用来实现的,因此可以在接口函数调用语句前通知驱动即将被调用,在接口函数调用语句后通知驱动调用已经结束,如图 6 中第 9 行和第 13 行所示.

4.1.4 可信内核范围

可信内核范围包括 VM 内核的代码段和未被隔离驱动程序的代码范围,用于区别写操作来源.VM 内核代

码范围可通过查询 Linux 的/boot 目录下的 System.map 文件获得.该文件中*_text* 和*_etext* 符号记录着内核代码的起止位置.未隔离驱动程序的代码范围,可通过未隔离驱动程序对应的 struct module 结构体的 *module_core* 和 *core_size* 字段来获取.

4.1.5 消息注入

上述所有获取的驱动和内核信息,最终都需要注入到 VMM 中.本架构通过 VMCALL 陷入指令将这些信息主动注入到 VMM 中的隔离环境中.该指令通过寄存器 EAX,EBX,ECX,EDX 和 ESI 向 VMM 传递参数.因此,在调用 VMCALL 指令前,需将相关信息放置于这些寄存器中.

由于每个被隔离驱动在 VMM 中都有各自的隔离域信息以及授权表信息,为了区分不同的隔离驱动程序,在注入时需要指定是哪个驱动程序的注入信息.考虑到各种消息都是在修改过的相应内核函数中注入,而这些函数都和驱动接口函数直接或间接相关.因此,如果在进程调用驱动接口函数时就指定该进程是和某一隔离域相关,则后续进程在调用到这些函数时,也就知道应该注入到哪个隔离域中.为此,本架构需修改内核的进程结构体 *struct task_struct*,为其添加一个隔离域字段 *isolation_domain*.该字段进程在开始调用驱动接口函数时设置,步骤如下:

(1) 由于隔离域信息中记录了被隔离驱动的代码范围,通过查询驱动接口函数指针的地址(例如图 6 中第 11 行的函数指针地址 host→hostt→queuecommand)是否在其范围内,就可确定其属于哪个隔离域;

(2) 根据查询得到的隔离域信息,设置进程结构体 *task_struct* 的 *isolation_domain* 字段.这样,在后续注入信息时,就可直接根据当前进程的隔离域字段 *isolation_domain* 获得该注入信息属于哪个驱动程序.

### 4.2 隔离环境

实现隔离环境的关键是授权表的管理和影子页表的设置.为了确保本文的隔离架构切实可行,隔离环境需要在确保隔离效果的基础上,避免过多的性能开销.

4.2.1 授权表管理

VM 内被隔离驱动程序对应的授权表是驱动程序对 VM 内核空间写访问权限的地址集合.由于驱动运行涉及的内存不断发生变化,本架构需要实时地对授权表进行建立、销毁、添加、删除、合并和查询等操作.如图 7 所示,授权表中的所有表项以红黑树和升序链表的形式共同组织起来.红黑树主要用于在授权表中快速查询特定地址是否存在,而升序链表用于快速获得相邻地址范围的节点以便更新.由于红黑树中记录的是一段内存地址范围,对其节点保存的内存信息进行更新处理较为复杂.

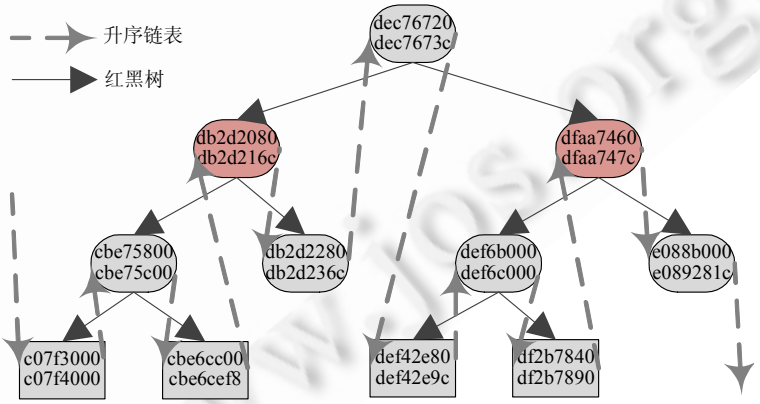



Fig.7 Structure of the authorization list

图 7 授权表结构示意图

4.2.2 页面设置

通过将被隔离的驱动程序所在的 VM 整个内核空间对应的影子页表设置为只读,就可在缺页异常中捕获

驱动写操作.理论上,本架构只在驱动运行期间将 VM 对应的影子页表设置为只读,其他情况下仍然保持可写权限.在驱动运行过程中,被允许的写操作所打开的影子页面权限,在写操作完成后,需重新被设置为只读.这样就可确保持续地捕获驱动程序的写操作.为实现影子页表的设置功能,本架构通过以下几个步骤来设置影子页表的写权限:

(1) 获取与 VM 虚拟地址对应的 VM 中各级页表及表项信息:若存在,继续下一步骤;否则,返回 *NULL*;

(2) 逐级查询 VM 虚拟地址对应的影子页表:若存在,继续下一步;否则,返回 *NULL*;

(3) 将查找到的影子页表的表项设置为只读.

然而,这种写操作捕获方法用在与内核频繁交互的驱动程序隔离,将带来严重的性能损失.部分经常访问的内存地址的写操作权限可能会被反复地打开和关闭.根据局部性原理,刚访问过的影子页面很可能在最近再次被访问到.为此,本架构利用一个影子页面缓存来延迟关闭最近访问到的影子页面的写权限,从而避免影子页面的写操作权限被反复地打开和关闭,如图 8 所示.

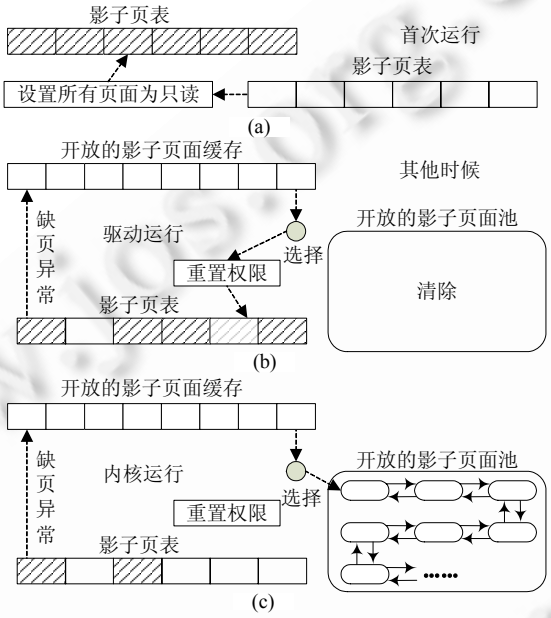



Fig.8 Schematic view of shadow page caching algorithm

图 8 影子页面缓存算法原理示意图

首先,当被隔离驱动程序第 1 次被调用时,将设置被隔离驱动所在的 VM 对应的影子页表为只读,以便捕获驱动程序的写操作,如图 8(a)所示.在驱动运行过程中,如果开放了某个影子页面,则将其信息记录到开放影子页面缓存中.记录在缓存中的页面,在执行完写操作以后将暂时不被重新设置为只读.如果缓存满了,最久未被使用的缓存信息将被替换,并将被替换页面设置为只读,如图 8(b)左半部分所示.

其次,当被隔离驱动未调用时,由于之前设置所有影子页表为只读,此时仍有大量影子页面处于只读状态.考虑到遍历整个影子页表来重新打开写权限较为费时,仍然利用缺页异常来根据需要打开写权限.同时,考虑到内核在未调用驱动时访问的影子页面在调用驱动后也很可能被使用到,为了避免这些页面的写权限被反复设置,此时打开写权限的页面,仍然被送入开放影子页面缓存,而被替换出的影子页面则被送入开放影子页面池,如图 8(c)所示.这样,下次调用驱动时只需设置页面池中的影子页面即可,节省设置页面只读的时间,也避免将最近使用的页面设为只读.

最后,当驱动程序不是第 1 次被调用时,只需将开放影子页面池中的页面设为只读,即可保证驱动写操作的捕获,如图 8(b)右边部分所示.驱动运行期间影子页面的开放和设置,仍然如图 8(b)左半部分所示.

4.2.3 写操作判断

当 VMM 出现缺页异常时,需要先判断写操作来源再进行相应处理.为此,本架构通过修改 KVM 的缺页异常函数 *FNAME*(*page_fault*)来添加相应的写操作判断功能.

当写操作指令来自可信任内核时,则具有写权限.KVM 通过 *mmu_set_spte* 函数将只读页面设置成可写,并记录下开放写操作的影子页表,以便在经过一定的延时时间后由影子页面缓存操作重新将其设为只读.

当写操作指令来自被隔离驱动程序,且写操作地址位于该驱动的授权表范围内时,也进行上述操作;否则,跳过非法的写操作指令.跳过指令的操作步骤为:首先,读取当前指令和指令长度;然后,计算出下一指令的位置;最后,让 KVM 直接执行下一指令,避免执行驱动的非法指令.

### 4.3 错误处理

在驱动程序出错之后,为了及时退出驱动程序,本架构将强迫使用驱动程序的进程直接跳转到出错返回代码.因此,需要提前保存驱动的跳转地址以及出错前的寄存器状态.如图 6 的第 10 行和第 12 行的宏所示,其展开的宏如图 9 第 1 行~第 22 行所示.其中,图 6 第 10 行的宏表示驱动程序被调用前的恢复准备工作,在调用驱动程序之前,将受影响的寄存器信息(图 9 中第 3 行~第 10 行)以及出错返回地址信息会被保存在栈中(图 9 中第 4 行、第 5 行).图 6 第 12 行的宏表示驱动程序出错后的跳转位置(图 9 中第 18 行)以及进一步的驱动故障恢复处理(图 9 中第 20 行).驱动程序在运行过程中,可能出现驱动调用嵌套的情况.即:内核调用驱动函数之后,驱动函数又调用内核函数;接着,该内核函数又再次调用驱动函数,以此类推.对于出现驱动调用嵌套的情况,每一层对驱动的调用都需要保存寄存器和出错返回地址信息,并在出错时逐层跳转出驱动程序.

```
/* 寄存器保存,出错返回地址保存 */
1   #define STACK_SAVE(s, a)                                    \
2   __asm__ __volatile__ (                                      \
3   "movl %0, %%eax\n\t"                                        \
4   "movl %1, %%ecx\n\t"                                        \
5   "movl %%ecx, ("PC"*4)(%%eax)\n\t"                           \
6   "leal (%%esp), %%ecx\n\t"                                   \
7   "movl %%ecx, ("ESP"*4)(%%eax)\n\t"                          \
8   "movl %%ebp, ("EBP"*4)(%%eax)\n\t"                          \
9   ..... .....                                                 \
10  "movl %%ebx, ("EBX"*4)(%%eax)\n\t"                          \
11  : : "m"(s),  "m"(a): "memory")
12 #define DRIVER_STACK_SAVE(·)                                 \
13  unsigned long reg=DRIVER_GET_STACK(·);                      \
14  unsigned long ret_addr=&& out;                              \
15  STACK_SAVE(reg,ret_addr)
/* 出错返回地址处,触发驱动恢复 */
16 #define DRIVER_STACK_QUIT(·)                                 \
17  if (IS_DRIVER_FAULT(·)) {                                   \
18  out:                                                        \
19      DRIVER_PUT_STACK(·);                                    \
20      DRIVER_RECOVERY_TRIGGER(·);                             \
21      return;                                                 \
22  }
/* 寄存器恢复,调转到出错返回地址 */
23 #define STACK_RESTORE(s)                                     \
24  __asm__ __volatile__ (                                      \
25  "movl %0, %%ecx\n\t"                                        \
26  "movl ("EBX"*4)(%%ecx), %%ebx\n\t"                          \
27  ..... .....                                                 \
28  "movl ("EBP"*4)(%%ecx), %%esi\n\t"                          \
29  "movl ("ESP"*4)(%%ecx), %%edi\n\t" \
30  "movl ("PC"*4)(%%ecx), %%edx\n\t" \
31  "jmp *%%edx\n\t"                                            \
32  : : "m"(s) : "memory")
33 #define DRIVER_STACK_RESTORE(·)                              \
34  unsigned long reg=DRIVER_QUERY_STACK(·);                    \
35  STACK_RESTORE(region)
```

Fig.9 Save and trigger the jump information of the driver

图 9 驱动跳转信息的保存和触发

在 VMM 中的隔离环境功能检测到驱动出错之后,将调用 *kvm* 的 *kvm_set_irq* 函数将中断注入到 VM 中,从而唤醒错误处理功能所注册过的驱动故障中断处理函数,该函数将启动正式驱动错误处理流程.如图 9 中第 23 行~第 35 行所示,在驱动程序出错后,驱动故障中断处理函数被唤醒,恢复寄存器的值(图 9 中第 25 行~第 30 行),并迫使使用驱动的进程直接跳转到出错返回地址(图 9 中第 31 行).

在所有使用驱动程序的进程退出之后,驱动程序将根据驱动监视功能所跟踪的驱动资源使用情况来释放驱动资源.首先,为了避免死锁,先释放驱动使用的锁资源;然后根据后进先出的原则,依次释放其他资源.资源释放是通过调用各自的内核资源释放函数来完成的,例如,网卡中的 *skb* 资源通过 *kfree_skb* 释放,而 I/O 映射资源则调用 *iounmap* 来释放.

在释放完驱动的资源之后,错误处理功能最终调用内核通用模块卸载函数 *free_module* 来释放驱动模块,避免故障驱动被继续使用.这样就可以在不破坏内核的情况下,及时隔离驱动故障,并移除故障驱动.由于大多数

驱动故障是瞬时故障,根据用户需要,可以重新加载驱动程序以继续提供硬件设备的服务功能.如果某个故障重复出现,很可能为确定性的故障,则不能继续重新加载驱动程序.同时,根据系统记录的故障日志,可以分析驱动故障的原因,以便修复驱动故障.

## 5 评 估

目前,本架构已经实现了 6 种常见驱动程序的隔离,见表 2.本节选用这些驱动程序对 VM 内部驱动隔离架构的特性进行评估,包括对 VM 内核可靠性的提高和对 VM 性能的影响.

**Table 2** Drivers which are isolated by the driver isolation architecture inside VM

**表 2** VM 内驱动隔离架构所隔离的驱动程序

| 驱动 | 描述 |
|---|---|
| e1000 | Intel PRO/1000网卡驱动 |
| rtl8139 | RealTek 8139C+系列芯片的网卡驱动 |
| usb_storage | USB大容量存储器设备驱动 |
| sd_mod | scsi磁盘驱动 |
| ens1370 | Ensoniq ES1370芯片的声卡驱动 |
| intel8x0 | Intel ICH (i8x0)系列芯片的声卡驱动 |

### 5.1 可靠性

本架构的重要目标是通过内存隔离来检测和限制驱动程序的写操作故障,防止驱动故障的扩散.为了检测架构对造成内存破坏的驱动故障的隔离效果,本文使用自动故障注入工具进行故障测试.

自动故障注入工具采用在 Rio File Cache[28],Nooks[15]以及 Mondrix[14]中使用的自动故障注入工具,我们将其移植到了 Linux 2.6.28.10 内核环境中.在加载了测试驱动模块后,通过指定注入的驱动名称、模拟的故障类型、注入故障个数以及随机数种子等,该工具通过随机改变内存中驱动程序的单一指令或多个指令,以模拟各种常见的编程错误.同时,在应用层运行使用到被测试驱动的应用程序,以触发注入的故障.

为了衡量对驱动故障的隔离效果,在测试中分别定义如下 VM-Native 和 VM-Isolation 两种测试环境.

(1) VM-Native:运行未修改的 KVM 和未修改的 VM 内核的测试环境;

(2) VM-Isolation:运行采用本文方法修改过的 KVM 和修改过的 VM 内核的测试环境.

每个注入故障都将分别在 VM-Native 和 VM-Isolation 两种环境下进行测试,以对比对驱动故障的隔离效果.为了充分体现本架构的隔离性能,本文对 6 个驱动分别进行了 200 次故障注入测试.

根据注入故障对 VM-Native 环境的破坏情况,本文将其分为 3 类:

(1) 崩溃失效:表明 VM 内核发生了致命性错误,出现无响应、宕机或自动重启等崩溃现象;

(2) 非致命性失效:表明 VM 未崩溃,但其中部分功能处于不正常状态;

(3) 静默失效:表明被破坏的数据暂时未被使用到,未表现出不正常.

根据 VM-Isolation 环境对造成 VM-Native 不同破坏情况的注入故障的隔离情况,本文将隔离效果分为隔离、捕获和丢失.

(1) 隔离:表示驱动故障被本架构成功地检查到,且故障驱动被顺利卸载;

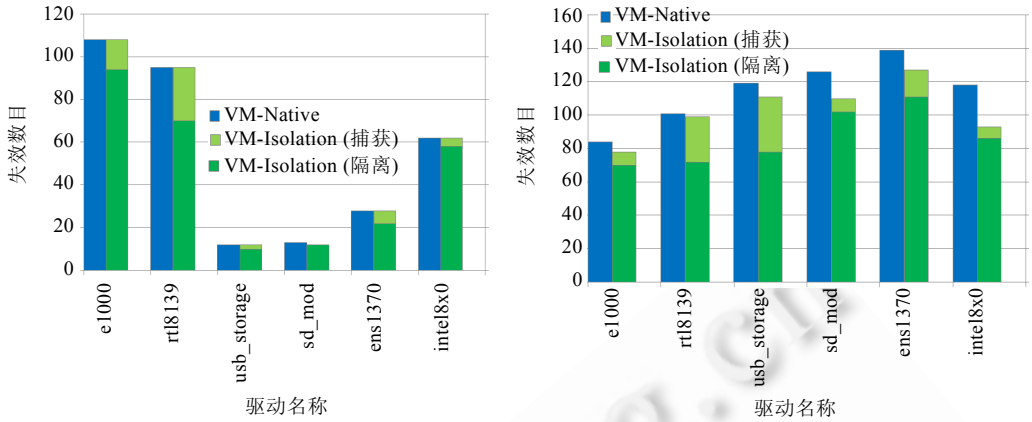
(2) 捕获:表示驱动故障被本架构检测到,但是故障驱动未被成功卸载;

(3) 丢失:表示本架构未能检测到驱动故障.

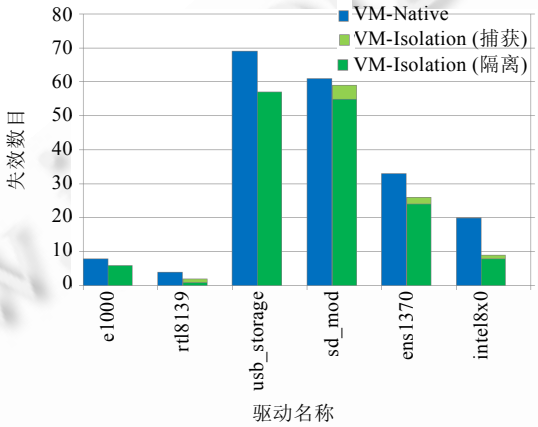
图 10 所示为 VM-Isolation 对造成 VM-Native 所在的 VM 各种失效的注入故障的隔离效果.如图 10(a)所示,在所有的故障注入测试中,有 318 次造成了系统崩溃,其中,266 次崩溃被本架构成功地阻止并卸载了故障驱动,防止故障驱动继续破坏 VM 内核,隔离率为 83.64%;还有 51 次也成功地捕获到了驱动故障,但未能卸载故障驱动,这种情况主要是因为没有及时结束正在使用故障驱动的进程,使得故障驱动无法卸载.

由于注入故障的位置是随机的,这些故障经过触发和传播后,破坏的 VM 内核位置也有所不同.有些异常行为可能不会造成 VM 内核的严重破坏,如网卡驱动可能出现路由不存在或者发送环满等非致命失效.考虑到注

入的故障造成的失效很大部分属于此类失效,为此,本文也需要分析本架构对此类失效的隔离效果.




(a) 崩溃失效 (b) 非致命失效




(c) 静默失效

Fig.10 Isolation effect of VM-Isolation to faults which cause crash, non-fatal and silent failures in VM-Native

图 10 对于造成 VM-Native 环境系统各种失效的注入故障,VM-Isolation 环境对这些故障的隔离效果

图 10(b)所示为 VM-Isolation 对造成 VM-Native 所在 VM 非致命性失效的注入故障的隔离效果.在所有 6 种驱动的故障注入测试中,有 687 次对 VM-Native 造成了非致命性失效,而 VM-Isolation 成功隔离了 519 次驱动故障,并捕获到了 99 次驱动故障,隔离率为 75.54%.不过,仍然有 69 次失效没有被检测到.分析这些未被检测到的失效可知,其中很大一部分是由于注入故障使得驱动读取不存在的页面造成的,这种读操作失效并非本架构所能防止的驱动故障.例如在 *usb* 驱动未检测到的错误中,超过一半都属于这种类型,另外几种是由于输入输出错误造成的;网卡驱动中也有一大部分属于这种类型,其他则是由未能完成 *scp* 传递造成的.总体上,VM-Isolation 减少了很多造成 VM-Native 非致命性失效的情况,特别是隔离了大多数驱动写操作引起的失效.

如果自动注入工具将故障注入到了不常执行的代码路径,这些故障就很可能不会被触发,也可能是在触发以后未造成明显的失效现象.不过,在这个过程中仍然可能存在写内存的操作,如果这些写操作位于驱动的授权表之外,虽然用户没有明显感觉到失效,但仍然可能被本架构所检测到.为此,本文同样分析 VM 内驱动隔离架构对这类失效的检查效果,图 10(c)所示为 VM-Isolation 对造成 VM-Native 所在静默失效的注入故障的隔离效果.总共有 195 次注入故障处于静默状态,本架构隔离到了 151 次故障,并检查到了 8 次故障,隔离率为 77.43%,达到了很好的驱动隔离效果.

实验结果表明:本架构能够消除 83.64%的系统崩溃,达到了架构的设计目标.同时,本架构对非致命性失效

和静默失效也有很好的隔离效果,分别达到 75.54%和 77.43%,从而可以在驱动程序出现严重故障前,通知驱动程序进行提前恢复.在与本架构采用类似测试方法的硬件隔离架构 Nooks[15]和 Mondrix[14]中,Nooks 针对致命性和非致命性故障的隔离率为 99%和 44.58%;Mondrix 对致命性、非致命性和静默故障的隔离率为 90%,39.1%和 2.5%.相对来说,在致命性故障上,本架构在虚拟化环境中同样实现了较高隔离率.同时,在非致命性故障和静默故障上,本架构对这两类故障的隔离率则大为提高.因此,本架构成功地将硬件隔离方法扩展应用到了虚拟化环境中.综上所述,本架构可有效防止 VM 内部的驱动程序故障对 VM 内核的破坏,避免 VM 内核的崩溃,提高了 VM 内核的可靠性.

### 5.2 性 能

VM 内驱动隔离架构的性能开销包括驱动内存使用的跟踪开销、影子页表缺页异常开销和写操作正确性检查开销等.本文通过一些基准测试程序来评估本架构对驱动正常运行造成的性能开销.由于本文研究的驱动隔离架构位于 VM 内,因此性能影响测试也在 VM 内完成.测试中所使用的计算机配置为 Intel Core2 Duo CPU E8400、1.9G 主频和 4GB RAM,宿主机和客户机操作系统都为 Centos 5.8,运行 KVM 的 QEMU 版本为 qemu-kvm-1.2.0.网卡驱动测试利用两台相同的机器来完成.

网络驱动的性能开销使用 netperf 性能测试工具来测试 TCP 的发送和接收性能.在 TCP 测试中,发送缓存为 16 384bytes,接收缓存为 87 380bytes,发送的消息大小为 16 384bytes.由表 3 可知:*e*1000 和 *rtl*8139 两种驱动,TCP 的吞吐率下降 1%~16%,CPU 使用率增加大约 2%~25%.由于网卡驱动会频繁地调用中断处理程序,其数据发送、接收以及状态查询也都通过中断来完成,这使得其驱动调用次数也较多,影子页表被设置的次数和写操作引起的缺页次数增多,最终导致网卡驱动的性能损失相对较大.但是总体来说,本架构对于网卡驱动的性能损失都在可接受的范围内.

**Table 3** Performance of driver isolation architecture inside VM

**表 3** VM 内驱动架构的性能开销

| 驱动名称 | 基准程序 | 吞吐率(M/s)/时间(s) | | 相对性能(%) | CPU使用率(%) | |
|---|---|---|---|---|---|---|
| | | VM-Native | VM-Isolation | | VM-Native | VM-Isolation |
| e1000 | TCP Send | 93.94 | 82.96 | 88.3117 | 28.10 | 46.78 |
| e1000 | TCP Receive | 92.90 | 91.06 | 98.0193 | 49.45 | 73.84 |
| rtl8139 | TCP Send | 93.95 | 78.92 | 84.0021 | 50.19 | 58.05 |
| rtl8139 | TCP Receive | 94.09 | 93.44 | 99.3092 | 46.01 | 48.45 |
| storage | Untar | 0.93 | 0.88 | 94.6237 | 14.26 | 37.69 |
| sd_mod | Untar | 2.20 | 2.15 | 97.7273 | 40.00 | 45.00 |
| ens1370 | mplayer | 79.47 | 79.49 | 99.9748 | 0.09 | 0.17 |
| intel8x0 | mplayer | 79.53 | 79.55 | 99.9749 | 0.09 | 0.20 |

为了衡量本架构对硬盘驱动的影响,分别在 scsi 硬盘和 U 盘中解压缩文件进行基准测试.由表 3(声卡驱动的性能为运行时间,其余驱动为吞吐率)可知:usb_storage 的吞吐率下降 5.4%,CPU 使用率增加大约 23.43%;而 sd_mod 的解压缩时间增加 2.3%,CPU 使用率增加大约 5%.这种性能损失的不同,与驱动的数据处理方式有关.usb_storage 驱动会创建自己的进程来处理块设备请求,该进程在没有请求时会处于睡眠状态.当需要处理新数据时,内核需要调度该进程,带来一些时间开销.而且在 VM-Isolation 中,由于影子页表被设置为只读,内核调度过程的写操作也会产生缺页异常处理,因此其开销会相对较大.而 sd_mod 由内核进程直接调用其接口函数来处理块设备请求,无需进程调度,开销相对较小.

声音基准通过 mplayer 以 128kbit 每秒播放一个 MP3 文件来测试.由表 3 可知,两种声卡驱动的时间增加都小于 0.1%,CPU 使用率增加小于 0.11%.可见,本架构对声卡程序的性能基本无影响.虽然声卡驱动和网卡驱动都存在中断,但声卡驱动并不使用中断处理数据请求,使其驱动被调用次数明显小于网卡驱动,故性能损失较小.

综上所述,VM 内驱动隔离架构虽然对驱动的性能造成了一定的影响,但是都在可接受的范围内,小于 16%.其中,除了网卡驱动的性能损失相对较大外,对其他驱动程序的性能影响很小.在当前利用 VM 实例进行隔离的 Xen[6]和 L4Ka[7]中,Xen 隔离的网卡驱动和硬盘驱动的性能损失分别为 3%~18%和 1%,L4Ka 隔离的网卡驱动和硬盘驱动的性能损失分别为 8%~25%和 2.4%.相对来说,本架构对于隔离这两类驱动带来的性能损失也在类似

范围内(网卡驱动为 1%~16%,硬盘驱动为 2.3%~5.4%).另外,由于本架构为了充分验证其隔离的有效性,在实验评估中所测试的驱动程序也比这两种方法要多(甚至包括它们所未测试的声卡驱动),因此可以充分说明,本架构不会对被隔离驱动带来严重的性能损失.

## 6 结束语

本文介绍了一种通过隔离驱动程序提高 VM 可靠性的架构,本架构能兼容已有的驱动程序,在 VM 内部实现对驱动程序的有效隔离,从而提高 VM 的可靠性.本架构通过监控驱动程序使用的内存资源来及时更新驱动授权表,基于监控驱动运行状态设置 VM 对应的影子页表写保护,从而构建驱动的隔离运行环境,保证驱动程序写操作的正确性,避免驱动错误扩散到 VM 内核.

目前,本架构的实现仍然依赖于对 VM 内核代码的少量修改,从而限制了架构的移植性.下一步,我们将研究在不修改 VM 内核的情况下实现本架构,并完善驱动程序的错误检查和恢复功能.

**致谢** 在此,我们向浪潮公司对本文的工作所给予的支持和建议表示由衷的感谢.

**References**:

[1] Andy C, Junfeng Y, Benjamin C, Seth H, Dawson E. An empirical study of operating systems errors. In: Proc. of the 18th ACM Symp. on Operating Systems Principles (SOSP). New York, 2001. 73−88. [doi: 10.1145/502034.502042]

[2] Ganapathi A, Ganapathi V, Patterson D. Windows XP kernel crash analysis. In: Proc. of the 20th Conf. on Large Installation System Administration (LISA). Berkeley: USENIX Association, 2006. 101−112. http://dl.acm.org/citation.cfm?id=1267805

[3] Nicolas P, Gaël T, Suman S, Christophe C, Julia L, Gilles M. Faults in Linux: Ten years later. In: Proc. of the 16th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS). New York, 2011. 305−318. [doi: 10.1145/1950365.1950401]

[4] Jim G. Why do computers stop and what can be done about it. In: Proc. of the 5th Symp. on Reliability in Distributed Software and Database Systems (SRDS). Los Angeles: IEEE Computer Society Press, 1986. 3−12. http://dl.acm.org/citation.cfm?id=21794

[5] Peter C, Brian N. When virtual is better than real. In: Proc. of the 8th USENIX Workshop on Hot Topics in Operating Systems (HOTOS). Washington: IEEE Computer Society, 2001. 133−138. http://dl.acm.org/citation.cfm?id=876409

[6] Keir F, Steven H, Rolf N, Ian P, Andrew W, Mark W. Safe hardware access with the Xen virtual machine monitor. In: Proc. of the 1st Workshop on Operating System and Architectural Support for the on demand IT InfraStructure (OASIS). New York: ACM, 2004. 1−10. http://www.sigops.org/announce/Aug04Sigops.htm

[7] Joshua L, Volkmar U, Jan S, Stefan G. Unmodified device driver reuse and improved system dependability via virtual machines. In: Proc. of the 6th USENIX Symp. on Operating Systems Design & Implementation (OSDI). Berkeley: USENIX Association, 2004. 1−14. http://dl.acm.org/citation.cfm?id=1251256

[8] Lin T, Ellick C, Reza F, Nevedita M, Jeffrey C, Francis D, Roy C. iKernel: Isolating buggy and malicious device drivers using hardware virtualization support. In: Proc. of the 3rd IEEE Int'l Symp. on Dependable, Autonomic and Secure Computing (DASC). Washington, 2007. 134−144. [doi: 10.1109/DASC.2007.16]

[9] Jorrit H, Herbert B, Ben G, Philip H, Andrew T. MINIX 3: A highly reliable, self-repairing operating system. SIGOPS Operating Systems Review, 2006,40(3):80−89. [doi: 10.1145/1151374.1151391]

[10] Dan W, Patrick R, Kevin W, Emin S, Fred S. Device driver safety through a reference validation mechanism. In: Proc. of the 8th USENIX Symp. on Operating Systems Design & Implementation (OSDI). Berkeley: USENIX Association, 2008. 241−254. http://dl.acm.org/citation.cfm?id=1855758

[11] Peter C. Get more device drivers out of the kernel. In: Proc. of the Linux Symp. 2004. Ottawa, 2004. 149−161. http://www.linuxsymposium.org/2004/

[12] Vinod G, Arini B, Michael SW, Somesh J. Microdrivers: A new architecture for device drivers. In: Proc. of the 11th USENIX Workshop on Hot Topics in Operating Systems. Berkeley: USENIX Association, 2007. 151−156. http://dl.acm.org/citation.cfm?id=1361412

[13] Ben L, Peter C, Nicholas F, Stefan G, Charles G, Luke M, Daniel P, Yueting S, Kevin E, Gernot H. User-Level device drivers: Achieved performance. Journal of Computer Science and Technology, 2005,20(5):654−664. [doi: 10.1007/s11390-005-0654-4]

[14] Emmett W, Junghwan R, Krste A. Mondrix: Memory isolation for Linux using mondriaan memory protection. In: Proc. of the 20th ACM Symp. on Operating Systems Principles (SOSP). New York, 2005. 31−44. [doi: 10.1145/1095810.1095814]

[15] Michael S, Henry L, Brian B. Improving the reliability of commodity operating systems. In: Proc. of the 19th ACM Symp. on Operating Systems Principles (SOSP). New York, 2003. 207−222. [doi: 10.1145/945445.945466]

[16] Zheng H, Zhang XJ, Wang ED, Wu N, Dong XS. Achieving high reliability on Linux for k2 system. In: Proc. of the 2012 IEEE/ACIS 11th Int'l Conf. on Computer and Information Science (ICIS). Washington, 2012. 107−112. [doi: 10.1109/ICIS.2012.25]

[17] Brian B, Stefan S, Przemysław P, Emin S, Marc F, David B, Craig C, Susan E. Extensibility safety and performance in the spin operating system. ACM SIGOPS Operating Systems Review, 1995,29(5):267−283. [doi: 10.1145/224057.224077]

[18] Galen H, James L. Singularity: Rethinking the software stack. ACM SIGOPS Operating Systems Review, 2007,41(2):37−49. [doi: 10.1145/1243418.1243424]

[19] Ruslan N, Godmar B. VirtuOS: An operating system with kernel virtualization. In: Proc. of the 24th ACM Symp. on Operating Systems Principles (SOSP). New York, 2013. 116−132. [doi: 10.1145/2517349.2522719]

[20] Tzi-cker C, Ganesh V, Prashant P. Integrating segmentation and paging protection for safe, efficient and transparent software extensions. In: Proc. of the 17th ACM Symp. on Operating Systems Principles (SOSP). New York, 1999. 140−153. [doi: 10.1145/319151.319161]

[21] Takahiro S, Kenji K, Takashi M. A hierarchical protection model for protecting against executable content. Technical Report, 2003.

[22] Úlfar E, Martín A, Michael V, Mihai B, George CN. XFI: Software guards for system address spaces. In: Proc. of the 7th USENIX Symp. on Operating Systems Design & Implementation (OSDI). Berkeley: USENIX Association, 2006. 75−88. http://dl.acm.org/citation.cfm?id=1298455.1298463&coll=DL&dl=ACM&CFID=551835625&CFTOKEN=44516859

[23] Miguel C, Manuel C, Jean-Philippe M, Marcus P, Periklis A, Austin D, Paul B, Richard B. Fast byte-granularity software fault isolation. In: Proc. of the 22nd ACM Symp. on Operating Systems Principles (SOSP). New York, 2009. 45−58. [doi: 10.1145/1629575.1629581]

[24] Mao YD, Chen HG, Zhou D, Wang X, Zeldovich N, Kaashock MF. Software fault isolation with API integrity and multi-principal modules. In: Proc. of the 23rd ACM Symp. on Operating Systems Principles (SOSP). New York, 2011. 115−128. [doi: 10.1145/2043556.2043568]

[25] Asim K, Matthew R, Michael S. Fine-Grained fault tolerance using device checkpoints. In: Proc. of the 18th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS). New York, 2013. 473−484. [doi: 10.1145/2451116.2451168]

[26] Feng Z, Jeremy C, Zachary A, Ilya B, Rob E, Matthew H, George N, Eric B. SafeDrive: Safe and recoverable extensions using language-based techniques. In: Proc. of the 7th USENIX Symp. on Operating Systems Design & Implementation (OSDI). Berkeley: USENIX Association, 2006. 45−60. http://dl.acm.org/citation.cfm?id=1298455.1298461&coll=DL&dl=ACM&CFID=551835625&CFTOKEN=44516859

[27] Leonid R, Peter C, Ihor K, Etienne S, Gernot H. Automatic device driver synthesis with termite. In: Proc. of the 22nd ACM Symp. on Operating Systems Principles (SOSP). New York, 2009. 73−86. [doi: 10.1145/1629575.1629583]

[28] Peter C, Wee N, Subhachandra C, Christopher A, Gurushankar R, David L. The Rio file cache: Surviving operating system crashes. ACM SIGOPS Operating Systems Review, 1996,30(5):74−83. [doi: 10.1145/248208.237154]

**郑豪**(1986－),男,福建仙游人,博士生,主要研究领域为操作系统,虚拟化技术的可靠性.
E-mail: mowendugu@gmail.com

**董小社**(1963－),男,博士,教授,博士生导师,CCF 高级会员,主要研究领域为高性能计算机体系结构,网格计算.
E-mail: xsdong@mail.xjtu.edu.cn

**王恩东**(1966－),男,研究员,主要研究领域为高端商用服务器,存储技术.
E-mail: wangend@inspur.com

**陈宝可**(1988－),男,硕士生,主要研究领域为虚拟化技术的可靠性.
E-mail: baoke.71@163.com

**朱正东**(1963－),男,博士,高级工程师,CCF 会员,主要研究领域为高性能计算机体系结构,网格计算.
E-mail: zdzhu@mail.xjtu.edu.cn