DOI：10.7652/xjtuxb201310002

一种采用驱动隔离的宿主机可靠性方法

郑豪[1]，张兴军[1]，王恩东[2]，董小社[1]，公维峰[2]
（1.西安交通大学电子与信息工程学院，710049，西安；2.高效能服务器和存储技术国家重点实验室，250013，济南）

摘要：针对虚拟化环境下宿主机中的驱动程序故障容易造成宿主机及其虚拟机崩溃的问题，提出了一种采用驱动隔离的宿主机可靠性方法。该方法基于 StyleBox 架构的保护域功能，将其扩展用于复杂接口的驱动程序的隔离，包括：为驱动和内核的复杂交互接口建立包装函数，从而限定驱动程序运行在保护域中；为驱动程序提供私有内存，以保证驱动程序能够在保护域内正常运行。实验结果表明：在利用该方法修改的 Linux2.6.28.10 宿主机内核中，对于随机注入网卡驱动并造成内核破坏的错误，该方法可以有效检测 90.63％的注入错误，并成功隔离 67.5％的注入错误，防止了驱动故障传播到宿主机内核，提高了宿主机的可靠性。

关键词：可靠性；虚拟化；驱动隔离；宿主机
中图分类号：TP316 **文献标志码**：A **文章编号**：0253-987X(2013)10-0007-06

Improving the Reliability of Host Machine with Driver Isolation

ZHENG Hao[1], ZHANG Xingjun[1], WANG Endong[2], DONG Xiaoshe[1], GONG Weifeng[2]
(1. School of Electronics and Information Engineering, Xi'an Jiaotong University, Xi'an 710049, China;
2. State Key Laboratory of High-End Server & Storage Technology, Ji'nan 250013, China)

Abstract: An approach to improve the reliability of the host machines with driver isolation is proposed to solve the problem that the driver fault crashes the host machines and its virtual machines. The approach is based on the protection domain of the pre-developed architecture StyleBox and extends it to isolate the driver with complex interfaces. For detail, it creates wrapper functions for the complex interactive interfaces between the driver and the kernel, limits the driver running in the protection domain, and provides private memories for isolated drivers to support the driver's normal operation in the protection domain. Experimental results on a Linux2.6.28.10 host machine kernel that is modified using the proposed approach show that the approach successfully detects 90.63% of the faults, which are randomly injected to the network driver and damage the host machines kernel, and isolates 67.5% of them, and that the reliability of host machines is improved.

Keywords: reliability; virtualization; driver isolation; host machine

当前，虚拟化技术已成实现服务器资源整合和优化的重要手段。虚拟化技术的隔离性使得单个虚拟机故障并不会影响其他虚拟机。然而，这种隔离性对于虚拟机所依赖的底层宿主机并不起作用，如果底层宿主机（特别是其中的驱动程序）出现故障，就可能造成整个服务器及其所有虚拟机的崩溃。由于驱动程序占操作系统内核代码量的比例巨大，一般由第三方开发缺乏完善测试，且和操作系统运行

收稿日期：2013-04-17。 作者简介：郑豪（1986—），男，博士生；张兴军（通信作者），男，副教授。 基金项目：国家“863 计划”资助项目（2008AA01A202，2011AA01A204）；国家科技攻关计划资助项目（2011BAH04B03）。
网络出版时间：2013-06-28 网络出版地址：http://www.cnki.net/kcms/detail/61.1069.T.20130628.1714.001.html

于相同地址空间和特权级，使得驱动故障很容易扩散到内核中。相关研究表明，Linux 系统中驱动程序故障是内核其他部分故障的 3 到 7 倍[1]。因此，实现宿主机内部驱动故障的隔离对提高服务器的可靠性具有重要意义。

目前，针对驱动引起的操作系统可靠性问题存在多种尝试的解决方法。虚拟机架构[2-3]利用虚拟机来承受驱动崩溃以换取整机可靠性的提高，但对宿主机的可靠性并未提高。微内核架构[4-5]将驱动隔离到独立的用户态进程中来避免对内核的破坏，但需重写驱动，破坏了兼容性，且性能损失严重。类型安全语言[6-7]方法通过编译器自动生成运行时检查，其需要修改驱动源代码，同样造成兼容性问题，且该方法隔离性较差。硬件架构[1,8]利用硬件的段保护或页保护来限制驱动写权限，大多在早期内核版本上开发不能被直接使用，有些还需要特殊硬件支持，其中 Zheng 等人研制的 StyleBox[9] 是一种较新的具有兼容性的解决驱动可靠性问题的硬件架构，然而它只在简单模拟驱动上进行验证。

本文针对运行虚拟机环境中的宿主机操作系统的可靠性问题，提出了一种采用驱动隔离的宿主机可靠性方法。该方法将 StyleBox 架构扩展到了 Linux 复杂接口驱动的隔离，通过修改内核为驱动建立保护域(包括限制驱动写权限的私有页表和允许驱动访问的最小内存资源)，从而使驱动被限制在保护域内运行，以便及时捕捉和处理驱动错误，防止驱动错误扩散到操作系统的内核中。

1 宿主机驱动隔离架构

StyleBox 是在 Linux 2.6.28.10 内核的 x86 平台下开发的一种提高操作系统可靠性的架构，包括隔离环境、错误检测和错误恢复 3 大功能，如图 1 所示。本文将重点讨论如何将 StyleBox 架构扩展用于复杂接口驱动的隔离。

隔离环境功能提供驱动隔离运行的相对封闭的环境(即保护域)，以防止驱动错误的扩散，通过为每个被隔离驱动程序建立一套私有页表来控制驱动的读写权限。私有页表允许被隔离的驱动对所有内核空间的读操作，但限制其只能对有限的地址空间进行写操作。另外，该功能还提供驱动运行所需的各种内存资源，并记录驱动所使用的内核数据。同时，该功能还提供进出保护域的调用，并通过包装函数自动实现这种调用。

错误检测功能负责检测驱动和内核交互过程中

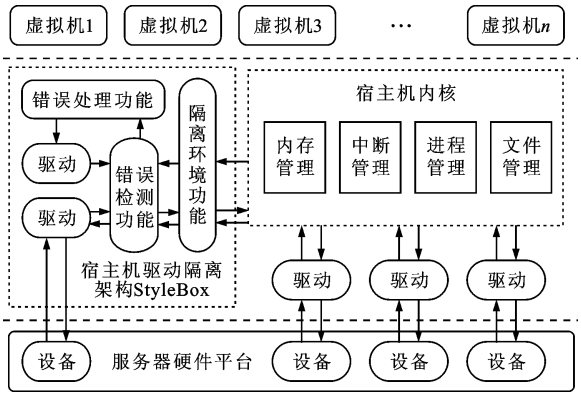
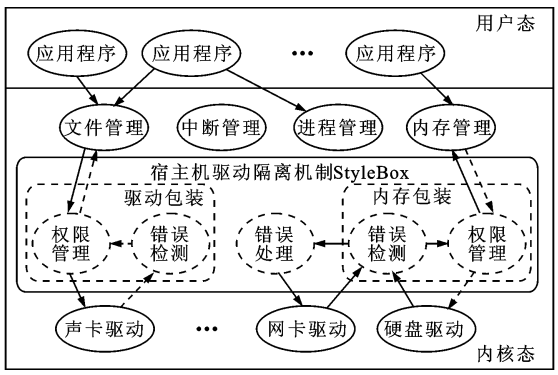


图 1 StyleBox 宿主机驱动隔离架构的结构图

发生的各种错误。例如，缓冲器指针或文件指针是否为空指针，是否使用已经删除的数据，是否发生死锁、无限循环，以及是否写保护域以外的内存等。为了保证与原驱动程序的兼容性，避免修改驱动程序，所有的错误检测都是通过在包装函数中添加运行时检测来完成，如图 2 所示。



实线表示调用开始；虚线表示调用返回 StyleBox 的调用过程

图 2 StyleBox 宿主机驱动隔离架构的调用关系示意图

错误处理功能负责预测和恢复驱动程序错误，能提供多种恢复策略。其中：时间冗余恢复策略通过卸载异常的驱动程序，清除驱动所使用的内核资源，并重新加载和初始化驱动程序来实现；空间冗余恢复策略利用冗余的驱动程序来更快速地恢复出错的驱动，该策略在驱动程序出错后直接切换到备用的驱动程序，利用备用驱动来接管底层硬件设备，然后再恢复出错驱动。

从图 2 可以看出，内核和驱动间的交互通过包装程序间接完成。内核在调用包装过的驱动函数时，会先切换到保护域，然后在保护域中执行原驱动函数，并检测返回值的正确性；驱动在调用内核函数时也有类似过程。由于驱动的功能复杂，使得驱动与内核的交互过程相当复杂，因此 StyleBox 只在简单的模拟字符设备驱动上实现了隔离验证。本文将

StyleBox 的保护域扩展用于复杂接口驱动程序的隔离，并通过实验证明该方法可以提高宿主机的可靠性。

2　宿主机网卡驱动隔离方法

在 StyleBox 架构中，被隔离驱动需要在限制写权限的保护域内运行，由保护域提供驱动正常运行所需的必要资源。为了完成对一个新驱动的隔离，本文完成了以下 2 项工作：①为待隔离驱动程序添加对应的包装函数；②适当地调整包装函数涉及内存的分配策略。下面以 e1000 网卡驱动为例，具体说明如何利用 StyleBox 架构实现对驱动程序的隔离，从而提高宿主机的可靠性。

2.1　函数包装

为保证兼容性，StyleBox 通过包装函数来实现内核和驱动间的调用，并在包装函数中包含保护域切换和错误检测，从而避免修改驱动程序。驱动的包装包括内核函数包装和驱动函数包装。

2.1.1　*内核函数包装*　驱动所使用的内核函数会以未定义符号的方式出现在编译后的驱动模块中。为了确定 e1000 网卡驱动待包装的内核函数，需完成以下工作：①在 StyleBox 架构中编译 e1000 网卡驱动，获得模块文件 e1000.ko（由于 StyleBox 对内核的导出函数进行了修改，因此其编译出驱动模块的未定义符号与原始 Linux 系统稍有差别）；②通过 linux 的 nm e1000.ko| grep U 命令查看该文件的未定义符号。图 3 为通过以上步骤所获取的驱动模块文件的未定义符号片段示例，这些符号对应的函数就是实现隔离 e1000 网卡驱动需要包装的主要内核函数。根据包装策略的不同，符号大致可分为以下 4 类。

（1）无需包装的未定义符号。例如，用来记录系统启动以来节拍总数的全局变量 jiffies，以及用于模块参数处理的函数 param_get_int 等。

（2）已包装的通用内存管理函数符号。为了便于管理驱动所使用的内存资源，StyleBox 对内核的内存管理函数进行了扩展，并将扩展的包装函数提供给被隔离驱动使用，因此这部分函数无需再包装，例如 slab 的释放函数 kfree 等。

U param_get_int	U param_set_int
U _kmalloc	U kfree
U _alloc_skb	U kfree_skb
U _pci_register_driver	U register_netdev
U dma_alloc_coherent	U dma_free_coherent

图 3　驱动模块文件中的未定义符号片段

（3）需修改 inline 函数类型并导出符号的函数。例如，为保证 dma 的分配/释放操作在内核而不是 StyleBox 的保护域中完成，而将 inline 类型函数 dma_alloc_coherent 和 dma_free_coherent 修改为导出符号函数，并进行包装。

（4）剩余需包装的未定义符号。这类符号是待包装内核函数的主要部分，例如 pci 设备注册函数 pci_register_driver 等。

未定义符号对应的内核包装函数名为加前缀的原函数名，如 register_netdev 函数对应的内核包装函数名为 stylebox_k_register_netdev。在内核包装函数中，需先进行切换出保护域的操作，然后再执行原内核函数，执行完毕后切换回保护域。至此，利用 StyleBox 提供的驱动加载方式加载 e1000.ko 模块文件，就可以将文件中的未定义符号自动链接到这些内核包装函数。

2.1.2　*驱动函数包装*　内核通过一系列的标准接口函数来操作驱动程序，这些标准接口函数以函数指针字段的形式定义在相关的接口结构体中。驱动在注册时，将驱动内部相应函数地址赋值于结构体的对应函数指针字段，这样内核就可通过调用函数指针字段来调用驱动函数功能。例如，内核可以通过调用网卡设备的接口结构体 struct net_device 中的 hard_start_xmit 字段来实现网络数据包发送。

通过将函数指针字段记录的函数地址替换为驱动包装函数的地址，而不是原驱动函数地址，就可以保证对驱动的调用进入 StyleBox 保护域运行。如图 4 所示，hard_start_xmit 字段将被赋予包装函数 stylebox_d_hard_start_xmit 的地址，而不是原 e1000_xmit_frame 函数的地址。这些标准接口对应的驱动包装函数，将增加进入保护域的切换操作，然后再调用原驱动函数，执行完毕再退出保护域，如图 2 所示。

由于 Linux 内核中记录接口函数的结构体比较复杂，有些驱动接口函数位于接口结构体的下一层结构体中。以网卡接口结构体 struct net_device 为例，其主要函数指针字段的替换示例如图 4 所示。由图 4 可见，内核除了调用本层结构体中的 open 和 hard_start_xmit 等接口函数指针外，还会调用下一层结构体，如 ethtool_ops 和 header_ops 等字段所指结构体中的函数指针，因此这些下层结构体中的接口函数指针也需要包装。网卡驱动所涉及的接口结构体如表 1 所示，其中最后一列函数所对应的内

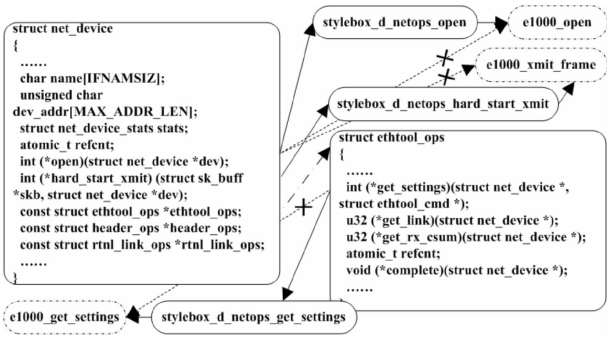


图 4 驱动接口结构体的指针替换示意图

核包装函数，就是进行原驱动接口函数指针替换所在的位置。例如，网卡结构体的驱动包装函数替换位置在网卡注册函数 register_netdev 对应的内核包装函数 stylebox_k_register_netdev 中进行。

表 1 e1000 网卡驱动需包装的接口结构体

接口结构体	接口函数数量	函数替换位置
pci_driver	31	_pci_register_driver
net_device	83	register_netdev
napi_struct	1	netif_napi_add
timer_list	1	mod_timer
irq_handler_t	1	request_irq

由表 1 可知，网卡驱动的接口不只是网卡接口本身，还包括 PCI 总线、定时器和中断等其他操作。选择包装这些接口结构体的具体原因如下：

(1)网卡驱动被作为一个 PCI 总线设备，并遵守相应的 PCI 总线规范，网卡驱动的注册和卸载等操作都以 PCI 设备的形式进行，因此需要包装结构体 struct pci_driver 中的驱动接口函数；

(2)网卡驱动的核心操作为结构体 struct net_device 中定义的一系列通信接口，网卡驱动通过这些接口同内核的 TCP/IP 协议栈进行交互，为此需要包装该结构体中的各种接口函数；

(3)为了提高网络处理效率，Linux 常采用 NAPI 机制来接收数据，为了保证被隔离驱动的接收过程也在保护域内运行，需要包装其涉及到的结构体 struct napi_struct 中的 poll 字段；

(4)网卡驱动程序通过定时器来周期性读取物理设备的相关寄存器，从而获得载波状态并更新设备的连接状态，因此需要包装相应定时器结构体 struct timer_list 中的 function 字段；

(5)在接收到或发送完数据包等操作时，网卡设备会产生中断，为了让网卡的中断调用过程在保护域内完成，还需包装网卡的中断处理函数。

2.2 数据分配

在 StyleBox 架构中，控制读写权限的最小粒度为一个页面，根据内存数据大小的不同，对保护域内内存的分配策略如下：①对于整页对齐的内存资源，先通过内核的内存管理机制申请，然后授予这些页面写权限；②对于非整页对齐的内存资源，先通过内核申请整个页面内存，然后在保护域内将其拆分成较小的资源对象来管理。

2.2.1 *共享数据分配* 对于接口数据结构，除了记录各种功能的驱动函数地址的函数指针字段外，还有记录驱动信息的各种其他字段，如硬件设备的信息、驱动运行状态和引用计数等。如果驱动和内核对于这些字段都具有写权限，驱动就有可能破坏到操作系统内核。而且驱动在内核中常常以多线程的形式运行，如果一个线程错误修改了接口结构体的内容，就有可能造成另一个线程的执行出错。

为了避免这种错误修改，需要在隔离驱动的保护域和内核中，分别分配一份相同的接口结构体。驱动程序只能写位于保护域内的接口结构体副本，并在确保修改正确无误的情况下，同步给内核中对应的结构体。由于使用 2 份接口结构体，因此需要确保数据同步工作的正确性和及时性。以 struct net_device 接口的同步过程为例，其内核和保护域版本分别称为 kdev 和 sdev，如图 5 所示，主要包括以下步骤。

(1)内核到驱动调用的同步：在所有使用到 net_device结构体的驱动包装函数中，在切换进保护域前将 kdev 所有内容复制到 sdev，在切换出保护域后将 sdev 所有内容复制到 kdev。

(2)驱动到内核调用的同步：在所有使用到 net_device结构体的内核包装函数中，在切换出保护域后且执行内核函数前将 sdev 所有内容复制到 kdev，在执行内核函数后切换回保护域前将 kdev 所有内容复制到 sdev。

(3)渐进精简：由于不是所有使用结构体 net_

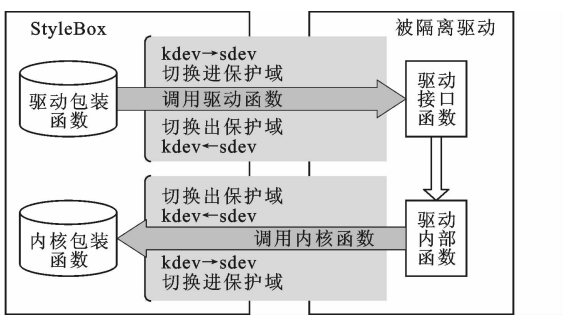


图 5 驱动接口结构体的同步过程示意图

device 的函数都进行写操作，且进行写操作的函数也只会修改其部分字段，若每次保护域切换都进行全部字段的同步操作，存在很多冗余操作。因此，可以通过分析函数的功能和具体实现，并结合实际运行，来逐步剔除不必要的同步操作。

需注意的是，类似于 struct net_device 接口结构体内的字段往往都很复杂，且内核和驱动间的调用关系也很复杂，若跳过步骤 1、2 直接进行步骤 3 的精简操作，往往容易出错。因此，需要在步骤 1、2 确保数据同步正确性的基础上，结合代码分析和实际测试，来渐进实现同步精简。

2.2.2　本地数据分配　对于被隔离网卡驱动，其涉及到的数据应该尽量在保护域内部。对于直接调用通用内存管理函数进行分配/释放的数据，可保证位于保护域内部。例如，调用 kmalloc 函数进行数据分配，由于被隔离驱动实际会调用包装函数 stylebox_k_kmalloc 来完成数据分配，且分配的数据就会位于保护域的内部。但是，驱动程序还会调用其他特定数据结构的分配/释放函数。图 6 为保护域内网卡驱动内存分配示意图，图中虚线所示的特定函数，其内部也会调用到通用的内存管理函数。例如，alloc_etherdev_mq 函数分配的数据，虽然驱动实际会调用包装函数 stylebox_k_alloc_etherdev_mq 来完成数据分配，但该函数实际上调用 kmalloc 函数而不是 stylebox_k_kmalloc 函数来完成数据分配，使得分配的数据位于保护域外，造成驱动对其没有写权限。为此，需对内核包装函数进行特殊处理，使这些特定数据的分配位于保护域内，具体如下。

(1)net_device 结构体分配。由上分析可知，网卡驱动 net_device 结构体的原分配函数为 alloc_etherdev_mq，直接调用其对应的包装函数会出现内存分配在保护域外的情况。为此，需在其包装函数中复制一份原函数的代码，并将其中 slab 分配函数替换为包装过的分配函数 stylebox_k_kmalloc，这样就可保证分配的数据在保护域内。

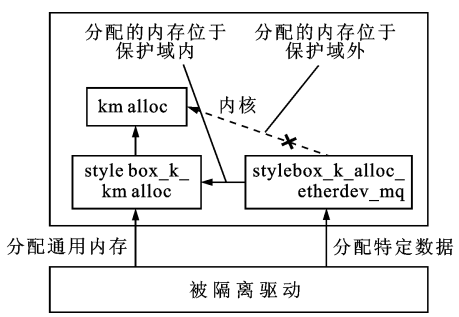


图 6　保护域内网卡驱动内存分配示意图

(2)sk_buff 结构体分配。内核中套接字缓冲区 sk_buff 都是从 2 个全局的专用内存高速缓存 skbuff_head_cache 和 skbuff_fclone_cache 中分配，这 2 个专用缓存在内核启动时创建，且位于保护域外部，隔离驱动程序不具有写权限。为了分配隔离驱动程序具有写权限的 sk_buff 结构体，需在保护域内部建立 2 个类似缓存 stylebox_skbuff_head_cache 和 stylebox_skbuff_fclone_cache，如图 7 所示。

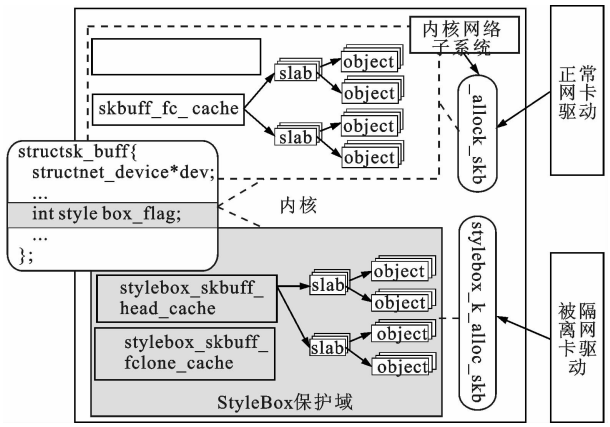


图 7　保护域内套接字缓冲区的内存分配示意图

由图 7 可见，保护域内的被隔离驱动函数会通过内核包装函数在这 2 个新建的高速缓存中分配 sk_buff 结构体。考虑到内核网络子系统也会分配 sk_buff 结构体，但是两者分配的 sk_buff 结构体在释放时，需释放到不同的专用内存高速缓存。为了区分被隔离驱动和内核分配的 sk_buff 结构体，需要为其增加一个字段 stylebox_flag 进行标示，如果是内核分配，该字段置 0，否则，置 1。这样根据该标志，就可将 sk_buff 释放到不同的专用内存高速缓存中。

虽然本文是以 e1000 网卡驱动为例来说明具体的隔离方法，但本文方法也同样适用于其他驱动程序的隔离。综上所述，基于 StyleBox 架构，本文对驱动程序的隔离方法主要需要完成以下 2 个步骤：①分析驱动和内核的交互接口，为这些接口建立包装函数(包装函数需进行权限切换和错误检查)，从而限定驱动程序在保护域内运行；②分析驱动程序的内存使用情况，并改变这些驱动程序所使用内存的分配方式，使得保护域内可提供驱动程序正常运行所需的内存。另外，虽然本文的驱动隔离方法是针对 Linux 2.6.28.10 的内核进行分析的，但对于其他类型和版本的操作系统环境中的驱动程序同样可以使用本文的方法进行隔离。

3 实验测试评估

本节将利用从 Rio File Cache 移植的自动错误注入工具向 e1000 驱动随机注入故障，以验证本方法对驱动错误的隔离性，并分析对驱动性能的影响。分别定义以下 2 种测试环境：①HOST_N：运行未修改的 Linux 内核和正常驱动的测试环境；②HOST_S：运行本文方法修改过的 Linux 内核和被隔离驱动的测试环境。

3.1 隔离性

HOST_S 的目标是防止 VM 内驱动错误造成整个 VM 的崩溃。为此，本文实验利用自动错误注入工具进行了 160 次错误注入测试，这些错误来自 8 种不同类型错误，每类进行 20 次测试。为提高注入错误被激活的概率，每次测试同时注入 10 个同类型的错误，测试项目为静默失效、功能丧失、内核报错和系统崩溃。静默失效表明系统暂时未表现出不正常；功能丧失表明系统部分功能已处于不正常状态，但未崩溃；内核报错表明 VM 内核捕捉到了非法操作，并报告 oops 错误；系统崩溃表明整个内核崩溃，需要重启系统。

表 2 为对注入故障的破坏后果测试结果。在 HOST_N 测试中，有 78 次注入的错误处于潜伏状态，由于其破坏的内核对象没有被使用到，因而没有表现出明显的错误，而在 HOST_S 测试中共检测到了 73 次，并成功隔离了 69 次这种错误。HOST_N 的 VM 内核报告了 66 次非法操作，HOST_S 都成功地检测到了这些错误，且隔离了 34 次错误。HOST_N 测试中注入的错误引起 10 次 VM 内核崩溃；HOST_S 检测到了 5 次将引起内核崩溃的错误，并成功隔离了 4 次内核崩溃错误，避免了部分宿主机的灾难性崩溃。HOST_N 有 6 次虽然没有造成系统崩溃但引起了功能损坏，都是由于注入的错误破坏了路由造成的；HOST_S 隔离 1 次这种错误。

表 2 注入故障的破坏后果测试结果

测试环境	静默失效数	功能丧失数	内核报错数	系统崩溃数
HOST_N	78	6	66	10
HOST_S(隔离)	69	1	34	4
HOST_S(检测)	4	0	32	1

注：HOST_N 表示注入故障对 HOST_N 环境产生的不同破坏后果；HOST_S(隔离)和 HOST_S(检测)分别表示造成 HOST_N 环境各种破坏的相同故障被 HOST_S 环境隔离、以及检测到但未隔离的情况。

实验结果表明，StyleBox 捕捉到了 90.63% 的注入错误，并隔离了很大一部分错误，对于所有 160 次错误，隔离率为 67.5%，具有很好的隔离效果，提高了宿主机的可靠性。

3.2 性能测试

为测试本文方法对驱动性能的影响，使用 netperf 工具进行 TCP 和 UDP 网络收发测试。用于收发测试的计算机配置为 Intel Core2 Quad CPU，2.83 GB 主频和 2 GB 内存，操作系统为 Ubuntu 10.04。

表 3 为利用 netperf 工具进行 TCP 或 UDP 批量发送或接收时，网卡驱动的收发性能。其中，HOST_S 中隔离的 e1000 驱动的吞吐率严重下降。由于网卡驱动的接口较多，因此内核会频繁通过不同接口调用网卡驱动，尤其是定时器接口和中断接口会被经常调用。利用 StyleBox 隔离的网卡驱动，每次内核与驱动之间的调用都需要切换保护域，即需要进行页表切换等操作，于是系统的 TLB 会被不停地刷新，从而导致读写内存时间的增加，使得被隔离网卡驱动的性能下降较多。这是由于网卡驱动在所有的驱动中是最复杂的，如果采用接口相对简单、内核与驱动间的交互频率也较适中的其他驱动，则被隔离驱动的性能损失会减少很多。虽然保护域的频繁切换导致了性能的部分损失，但是换来了系统驱动故障隔离性能的提高。因此，本文利用 StyleBox 在宿主机进行驱动隔离的方法，能够通过部分性能损失来换取驱动故障的有效隔离，对于提高宿主机操作系统的可靠性具有重要意义。

表 3 HOST_N 和 HOST_S 环境中网卡驱动的收发性能

测试基准	吞吐率/Mb·s^{-1}	
	HOST_N	HOST_S
TCP 发送	84.76	16.03
UDP 发送	86.60	12.50
TCP 接收	83.77	14.86
UCP 接收	87.50	17.10

4 结 语

本文将 StyleBox 架构扩展用于复杂接口驱动程序的隔离的方法，具体包括如何对驱动和内核之间的调用接口进行包装，以及如何确保被隔离驱动使用的内存在保护域内分配和释放。实验表明，利

(下转第 24 页)