计算机研究与发展
Journal of Computer Research and Development
DOI:10.7544/issn1000-1239.2014.20130612
51(10):2348-2359,2014

VMM 中 Guest OS 非陷入系统调用指令截获与识别

熊海泉[1,2] 刘志勇[1] 徐卫志[3] 唐士斌[1,2] 范东睿[1]
[1](计算机体系结构国家重点实验室(中国科学院计算技术研究所) 北京 100190)
[2](中国科学院大学 北京 100049)
[3](清华大学微电子所 北京 100084)
(xionghaiquan@ict.ac.cn)

Interception and Identification of Guest OS Non-trapping System Call Instruction within VMM

Xiong Haiquan[1,2], Liu Zhiyong[1], Xu Weizhi[3], Tang Shibin[1,2], and Fan Dongrui[1]
[1](*State Key Laboratory of Computer Architecture*, *Institute of Computing Technology*, *Chinese Academy of Sciences*, *Beijing* 100190)
[2](*University of Chinese Academy of Sciences*, *Beijing* 100049)
[3](*Institute of Microelectronics*, *Tsinghua University*, *Beijing* 100084)

Abstract To solve the problem that VMM can not monitor and control some Guest OS specific behavior due to its non-trapping feature in virtualized computing environment, an idea has been proposed to make those non-trapping instructions trap into VMM through modifying their normal execution conditions so as to cause system exception. According to the idea, special methods have been explored on how to intercept and identify the three different non-trapping system call instructions of x86 architecture from Guest OS within VMM. The int and sysenter instructions trap into VMM through causing GP system exception, while syscall instruction trap into VMM through causing UD system exception. They are identified with the virtual CPU context information within VMM. The Qemu&Kvm based prototype indicates that VMM can successfully intercept and identify all the three system call behaviors from Guest OS, and the performance overhead is within an accepted range for normal applications. For example, in unixbench shell test case, the performance overhead ratio is range 1.900 to 2.608. Compared with existing methods, they are all based on the architecture specification, so the advantage is that they are transparent to Guest OS and need not any modifications to Guest OS.

Key words Guest OS; virtual machine monitor (VMM); virtualization; non-trapping instruction; system call

摘 要 针对虚拟化环境下 Guest OS 某些特定指令行为不会产生陷入从而在虚拟机管理器(virtual machine monitor, VMM)中无法对其进行监控处理的问题,提出通过改变非陷入指令正常运行条件,使其执行非法产生系统异常陷入 VMM 的思想;据此就 x86 架构下 Guest OS 中 3 种非陷入系统调用指令在 VMM 中的截获与识别进行研究:其中基于 int 和 sysenter 指令的系统调用通过使其产生通用保护(general protection, GP)错系统异常而陷入,基于 syscall 指令的系统调用则通过使其产生 UD(undefined)未定义指令系统异常而陷入,之后 VMM 依据虚拟处理器上下文现场信息对其进行识别;基于 Qemu&Kvm 实现的原型系统表明:上述方法能成功截获并识别出 Guest OS 中所有 3 种系统调用行为,正常情况下其性能开销也在可接受的范围之内,如在 unixbench 的 shell 测试用例中,其性能开销

收稿日期:2013-04-25;**修回日期**:2013-12-16
基金项目:国家"九七三"重点基础研究发展计划基金项目(2011CB302501);国家自然科学基金项目(61332009);计算机体系结构国家重点实验室开放课题(CARCH201203);北京市教委科技计划面上项目(KM201210028004)

比在 1.900～2.608 之间.与现有方法相比,它们都是以体系结构自身规范为基础,因此具有无需修改 Guest OS、跨平台透明的优势.

关键词 客户操作系统;虚拟机管理器;虚拟化;非陷入指令;系统调用

中图法分类号 TP316

虚拟化技术在 IT 安全领域得到了广泛的应用[1],它们往往要求虚拟机管理器(virtual machine monitor, VMM)能够对运行其上的 Guest OS 行为及状态进行有效监控.通常情况下,Guest OS 在其执行到敏感指令时会与 VMM 发生交互,之后利用硬件体系结构或 Guest OS 的相关软件规范知识,VMM 可以推演挖掘出 Guest OS 中更多有用的监控信息[2-3].然而出于性能考虑或硬件自身设计等原因,Guest OS 中总会存在某些与特定行为事件相关的指令因不属于敏感指令[4]而无法直接陷入 VMM 的情形,例如 x86 架构下系统调用指令就是其中之一[2],这时要在 VMM 中对其进行监控处理就非常困难.

为了解决 Guest OS 非陷入指令在 VMM 中的截获监控问题,现有研究的基本思想是用可陷入敏感指令替换非陷入指令[5].例如 Guest OS 代码注入技术,它要求在 Guest OS 特定位置(通常是特定行为执行总入口处)替换注入一段可陷入敏感代码,使 Guest OS 在执行特定行为之前先执行这段代码从而产生陷入,之后再执行之前保存的原始代码;基于双重指令交换技术,它需要进行 2 次可陷入敏感指令与非陷入指令交换:其中第 1 次用其覆盖 Guest OS 特定行为代码开头的非陷入指令并保存它,这样执行时会产生陷入,然后在可陷入敏感指令处理中恢复之前被覆盖的非陷入指令原始代码,并在特定行为代码的尾部第 2 次用可陷入敏感指令替换非陷入指令,这样第 2 次陷入时,将可陷入敏感指令再次覆盖 Guest OS 特定行为代码开头部分并恢复第 2 次覆盖的非陷入指令,为下一次执行作准备.以上 2 种方法的共同特点是需要对 Guest OS 二进制代码进行修补或插入,具体位置与特定的 Guest OS 有关,要准确定位非常困难,因此可移植性差,且对 Guest OS 不透明;另外对于双重指令交换会因增加虚拟机陷入次数而产生更大的性能开销.

为此,本文提出通过改变非陷入指令运行条件使其产生系统异常陷入的思想.据此思想并考虑到系统调用行为监控在系统安全中的重要性[6-11],本文专门就 x86 架构下基于 int,sysenter 以及 syscall 3 种非陷入系统调用指令的截获与识别技术方法进行研究,其中基于 int 和 sysenter 指令的系统调用行为使它们产生通用保护(general protection, GP)错系统异常而陷入;而基于 syscall 指令的系统调用则通过关闭 EFER(extension feature enable register)的 SCE(system call enable)标志位使其产生 UD(undefined)未定义指令系统异常而陷入;之后 VMM 再依据虚拟处理器上下文现场信息进行识别.基于 Qemu&Kvm 实现的原型系统表明:上述方法能成功截获并识别出 Guest OS 中所有 3 种系统调用行为,正常情况下其性能开销也在可接受的范围之内,如在 unixbench 的 shell 测试用例中,其性能开销比在 1.900～2.608 之间.相比已有方法,本文方法都是以体系结构自身规范为基础的,具有无需修改 Guest OS、跨平台透明的优势.

1 Guest OS 非陷入指令的截获与识别

要在 VMM 中监控并处理 Guest OS 内部发生的特定行为事件,其前提就是在它们发生时 VMM 能够自动获得告知.大多数情况下,Guest OS 中需要由 VMM 控制管理的行为会因其为敏感指令(sensitive instruction)而自动产生陷入并进入 VMM 由其处理,具体如图 1(a)所示.

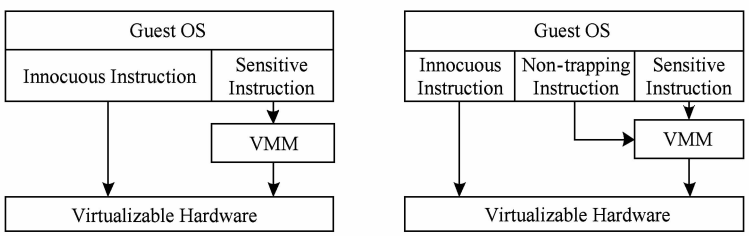


Fig. 1 The interaction between Guest OS and VMM.

图 1 Guest OS 与 VMM 交互

然而在实践中也会存在与 Guest OS 特定行为事件相关的非陷入指令(non-trapping instruction)而无法陷入 VMM 的情形. 这首先是因为硬件设计之初不可能考虑到所有应用需求;其次从软硬件协同优化设计的角度看,硬件上尽可能地将那些应用广泛的特定事件行为指令设计为可陷入敏感指令,而那些用得少则让它不直接产生陷入,需要时可利用软件方法来实现非陷入指令的陷入,如图 1(b)所示.

1.1 Guest OS 非陷入指令在 VMM 中的截获

对于与 Guest OS 内部特定行为相关的非陷入指令的截获,首先根据硬件体系结构所遵从的软件规范,确认此指令是否支持动态开启与关闭. 如果支持,则通过软件将其设置为关闭,之后执行它必然产生系统异常,进而达到陷入 VMM 目的;否则进一步根据硬件体系结构的软件规范分析此指令正常执行所需满足的条件并进行改变打破它,使其产生系统异常从而陷入 VMM,其基本流程如图 2 所示. 第 1 种情形通常是因为系统要向前兼容而产生的,从而使某些特性可以动态开启与关闭;第 2 种则在大部分情况下可以找到相应的处理方法,当然也存在找不到的情形,这时要实现陷入就需要借助于前面介绍的用可陷入敏感指令替换非陷入指令的方法来解决,本文不讨论这种少数情况.

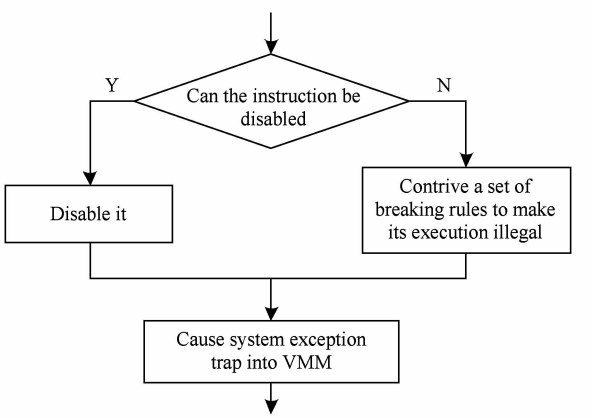


Fig. 2 The interception of Guest OS non-trapping instruction.

图 2 Guest OS 非陷入指令截获

1.2 Guest OS 非陷入指令在 VMM 中的识别

Guest OS 的应用程序在执行到经过上述处理后的非陷入指令时会产生系统异常陷入,进而将控制权转移到 VMM. VMM 中存在一个虚拟机陷入总入口(VM exit),根据 Guest OS 敏感指令陷入原因派发到相应的模拟仿真例程进行处理. 因此对于 Guest OS 非陷入指令的识别首先要在 VMM 中定位到虚拟机陷入总入口,进一步查找因非陷入指令运行条件变化而产生的系统异常处理入口;然后利用虚拟处理器上下文现场信息来识别区分是因非陷入指令产生还是正常情况下产生的;如果是正常情况,则将异常直接转交给 Guest OS 处理;否则,需要根据具体应用进行相关处理并模拟仿真,最后再返回 Guest OS. 图 3 描述了 VMM 中非陷入指令识别处理基本流程:

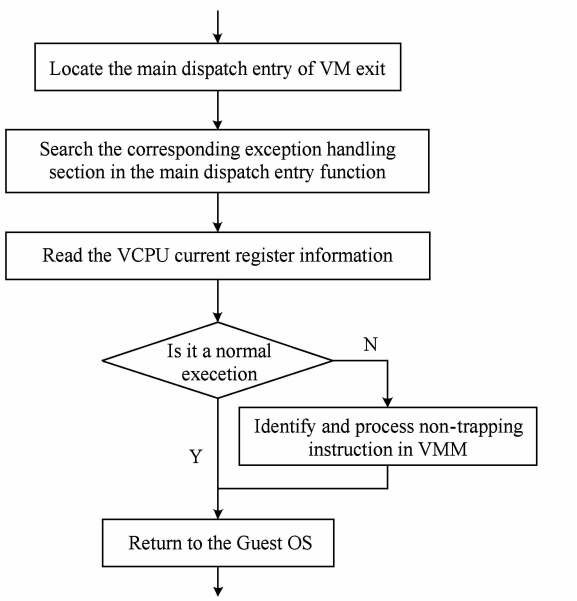


Fig. 3 The identification of Guest OS non-trapping instruction.

图 3 Guest OS 非陷入指令识别

2 Guest OS 非陷入系统调用指令的截获与识别

为了能够在 VMM 中对 Guest OS 进程发出的系统调用行为进行截获与识别跟踪处理,逻辑上需要完成 3 件事情:1)Guest OS 中当前进程的识别;2)Guest OS 中系统调用行为的截获与识别;3)Guest OS 系统调用参数信息获取与处理.

2.1 Guest OS 中当前进程的识别

Guest OS 中的系统调用是由进程发出的,在一些特定场合如安全领域,往往需要确切知道哪些应用可以访问哪些资源以便实施相应的安全策略. 在技术上这就要求能够将系统调用行为与具体进程建立起关联,即在 VMM 中能够识别出 Guest OS 的当前进程. 在现代操作系统中,无论是 Windows 还是 Linux,或是其他操作系统,一般来讲它们都会为每个进程提供一个单独的地址空间,并通过一组页

表设施来实现内存资源的动态映射,而这组页表是由页目录基地址唯一确定的.每当进程上下文发生切换时,都需要将新进程的页目录基地址加载到物理机器特定系统寄存器当中,如 x86 架构下的 CR3 控制寄存器[2,12].这一操作属于敏感指令,执行时会产生陷入,因此在 VMM 中可以通过监测 CR3 的更新来识别 Guest OS 的当前进程.考虑到新进程创建可能会重用之前撤销的进程页目录基地址,实现时可以同时考虑顶层页表中第 1 有效项从而创建唯一的标识符.

2.2 Guest OS 系统调用截获与识别

在 x86 架构的发展演变过程中共出现了 3 种系统调用实现机制:基于 int 软中断指令、基于 sysenter 指令以及基于 syscall 指令.Guest OS 应用程序执行系统调用时通常不会直接陷入 VMM.因此要在虚拟机之外监控虚拟机内部应用程序行为就比较困难.为了处理这个问题,首先需要使它们执行时陷入到 VMM,然后在 VMM 中进行识别处理.以下具体论述 x86 架构下 3 种非陷入系统调用指令的截获与识别技术方法.

2.2.1 基于 int 指令实现的系统调用截获与识别

x86 架构下中断处理是通过中断描述符表实现的.中断描述符表最大可以有 256 项,其大小界限由中断描述符表指针寄存器(interrupt description table register, IDTR)指明.当中断产生时,硬件通过 IDTR 找到中断描述符表(interrupt description table, IDT),判断中断向量号是否通过越界等权限检查并最终跳转到对应的中断处理例程开始执行.虚拟化环境下,Intel VT-x 只允许 Guest OS 中的系统级中断(号码在 0～31 之间)自动陷入 VMM,而用户级中断(号码在 32～255 之间)则不会自动陷入 VMM.

为了使 Guest OS 中基于 int 软中断实现的系统调用产生自动陷入(其中断向量号位于 31 以上),这里通过修改 IDTR 寄存器界限值来实现,即将 IDTR 寄存器 *Limit* 界限字段设置为 32,如图 4 所示.这样所有系统中断不受影响,而用户级中断(大于 31)会因为 IDT 表访问越界而触发 GP 通用保护错异常陷入.之后 VMM 根据 GP 通用保护错异常陷入进行识别处理,确认它是因 int x 指令的系统调用还是正常情形下产生的.主要识别依据是检查当前指令(如是否为 int x 指令以及当前中断向量号 x 是否大于 31),如果是正常情况则将其转发给 Guest OS 正常处理,否则识别出其是因为 int x 系统调用中断所致,则需收集相应的参数并依据事先给定的规则处理.

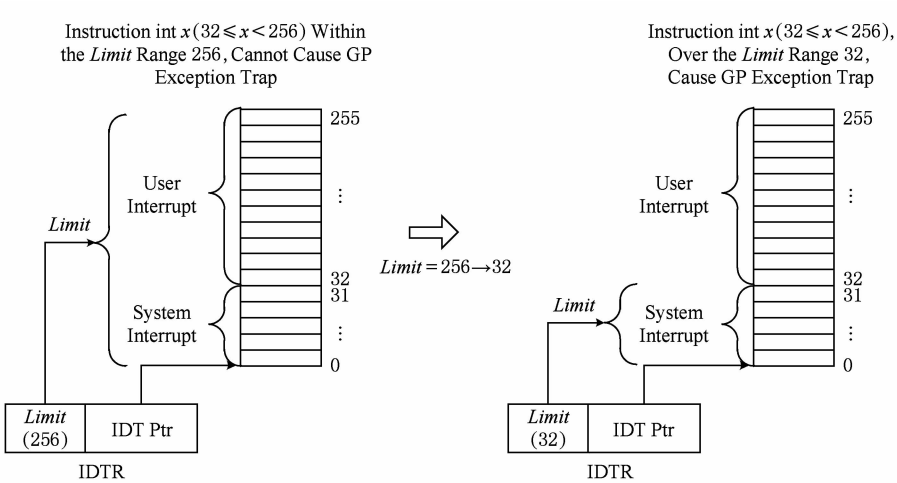


Fig. 4 Interception and identification of int-based system call mechanism.

图 4 基于 int 指令的系统调用截获

2.2.2 基于 sysenter 系统调用截获与识别

基于 sysenter 指令实现的系统调用依赖于一组 MSRs,即 SYSENTER_CS_MSR, SYSENTER_ESP_MSR 以及 SYSENTER_EIP_MSR,它们代表了系统调用处理的入口地址.当 Guest OS 应用程序执行 sysenter 时,系统会将代表 Guest OS 系统调用处理总入口地址的信息加载到这组 MSRs 中,这样 sysenter 执行就不会经由 VMM,如图 5(a)所示.为了截获这种系统调用机制需要对其执行条件进行调整,使其产生异常陷入.依据 Intel 体系结构软件规范,当 sysenter 依赖的这组 MSRs 寄存器装入非法值时,sysenter 会产生 GP 通用保护错异常.为此将 Guest OS 中代表系统调用入口的 MSRs 寄存器原始值先保存起来,然后给它们装入一组 NULL

非法值,这样 sysenter 执行时会因 SYSENTER_CS_MSR 的 NULL 值加载到 CS 段寄存器,产生 GP 通用保护错异常陷入 VMM,如图 5(b)所示. VMM 利用虚拟化处理器现场信息来区分是因为 sysenter 还是通常情形下导致的 GP 异常. 如果是通常情形的 GP 陷入,直接注入给 Guest OS;否则因为 sysenter 导致,则执行相关数据信息收集并处理,再利用先前保存好的 SYSENTER_CS_MSR 值进行仿真并将控制权返回给 Guest OS.

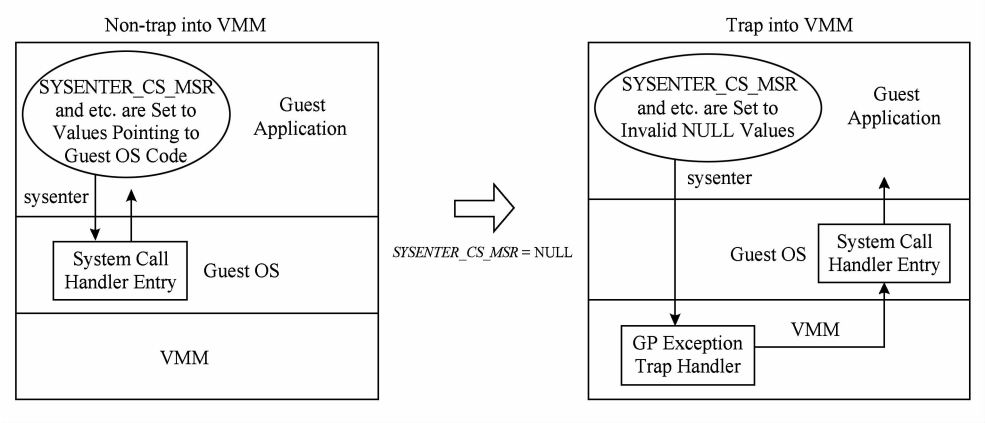


Fig. 5 Interception and identification of sysenter-based system call mechanism.

图 5 基于 sysenter 指令的系统调用截获与识别

2.2.3 基于 syscall 的系统调用截获与识别

syscall 指令也依赖于一组特征寄存器 MSRs,即 STAR_MSR, CSTAR_MSR 以及 LSTAR_MSR. 正常情况下 Guest OS 中 syscall 指令不会陷入 VMM,因此无法直接对其截获控制,如图 6(a)所示. 然而基于硬件体系结构规范,可以通过系统设置 EFER 寄存器的 *SCE* 位来开启或关闭此指令. 即在 *SCE*=0 时执行 syscall 指令,系统会产生 UD 未定义指令异常陷入,如图 6(b)所示. VMM 在探测到 UD 异常陷入后,借助虚拟处理器上下文现场异常信息来识别是否由 syscall 指令引起. 如果是通常情况,直接转发给 Guest OS,否则进行数据信息收集与处理之后再转交给 Guest OS.

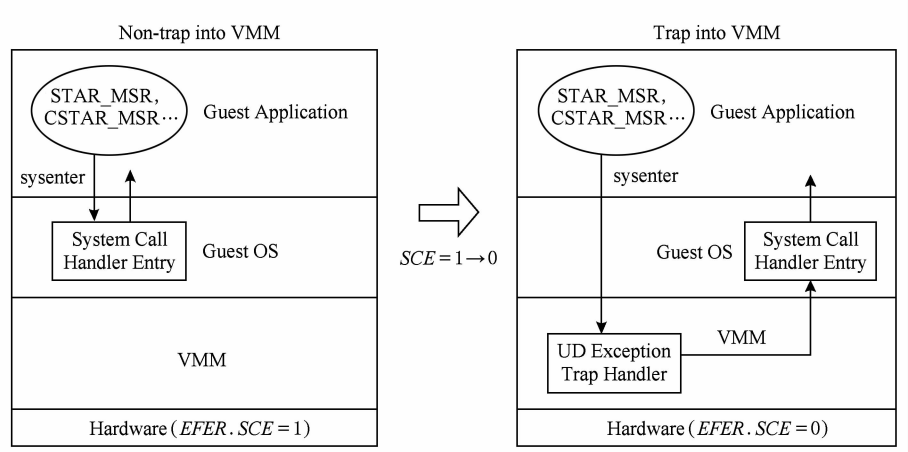


Fig. 6 Interception and identification of syscall-based system call mechanism.

图 6 基于 syscall 指令的系统调用截获与识别

2.3 Guest OS 系统调用参数信息获取及处理

不同应用对于系统调用跟踪所需的信息是不同的:有些只需记录系统调用的执行序列而不需要参数;另一些则需要更具体的信息,如寄存器值、基于堆栈的参数以及系统调用的返回值. 由于很难事先预知 Guest OS 的类型以及可能的系统调用跟踪应用,因此这里并不为每个系统调用列出一组固定的数据格式,而是创建一种机制让用户自己能够细粒

度地去定义描述系统调用处理规则.例如,用户可以规定使用哪一个通用寄存器来存放 Guest OS 系统调用号,这样原型系统就可以提取这个系统调用号.在其他情况下,系统调用参数甚至是内存指针引用的变量.为了满足这些要求,系统通过一组规则使它能够描述基于堆栈的和寄存器传递的参数.以下处理规则参考了文献[6]的文法定义,具体如图 7 所示:

```
rule→add_system_call_monitor_rule〈condition〉〈location〉〈action〉
condition→〈register〉〈value〉
location→〈register〉〈offset〉
register→ rax|rbx |rcx|rdx|rsp|rbp|rsi|rdi
value→[0,4294967295]
offset→[−2147483648,2147483647]
action→hex|int|uint
```

Fig. 7 The definition of system call processing rule.

图 7 系统调用处理规则定义

一条系统调用处理规则包括监控的条件 condition:其中包括寄存器 register 及其条件值 value;需要读取数据的地址信息 location:包括基地址寄存器 register 以及偏移 offset;同时也包括简单的信息输出格式描述 action,可以是十六进制 hex 以及十进制 int 和 uint.

3 系统实现

为了验证上述方法的正确有效性,本节描述基于 Qemu&Kvm 平台的原型实现,并在第 4 节中对其功能与性能开销进行评估.

3.1 系统逻辑结构

Qemu&Kvm 是一种宿主型虚拟计算机系统,逻辑上由用户态 Qemu-Kvm[13] 和内核态 KVM[14] 两部分组成.其中 KVM 负责内存和处理器的虚拟化,Qemu-Kvm 则负责 I/O 设备的虚拟化.之所以选择 Qemu&Kvm,不仅因为它开源,更为重要的是它有一个非常灵活方便的监控器 Qemu Monitor.它不仅可动态改变 Qemu&Kvm 的系统设置,也可与 Qemu&Kvm 虚拟机进行动态交互,例如可以控制虚拟机暂停或恢复、获取运行状态、更改运行时配置等等.因此利用 Qemu Monitor 作为原型系统的操作控制界面,实现简单.

图 8 是原型系统逻辑结构,包括 3 部分:Qemu Monitor 操作控制模块(qemu monitor user control module)、基于 KVM 的功能扩展模块(KVM extension)以及基于 netlink 方式的系统调用处理信息输出模块(netlinkclient).

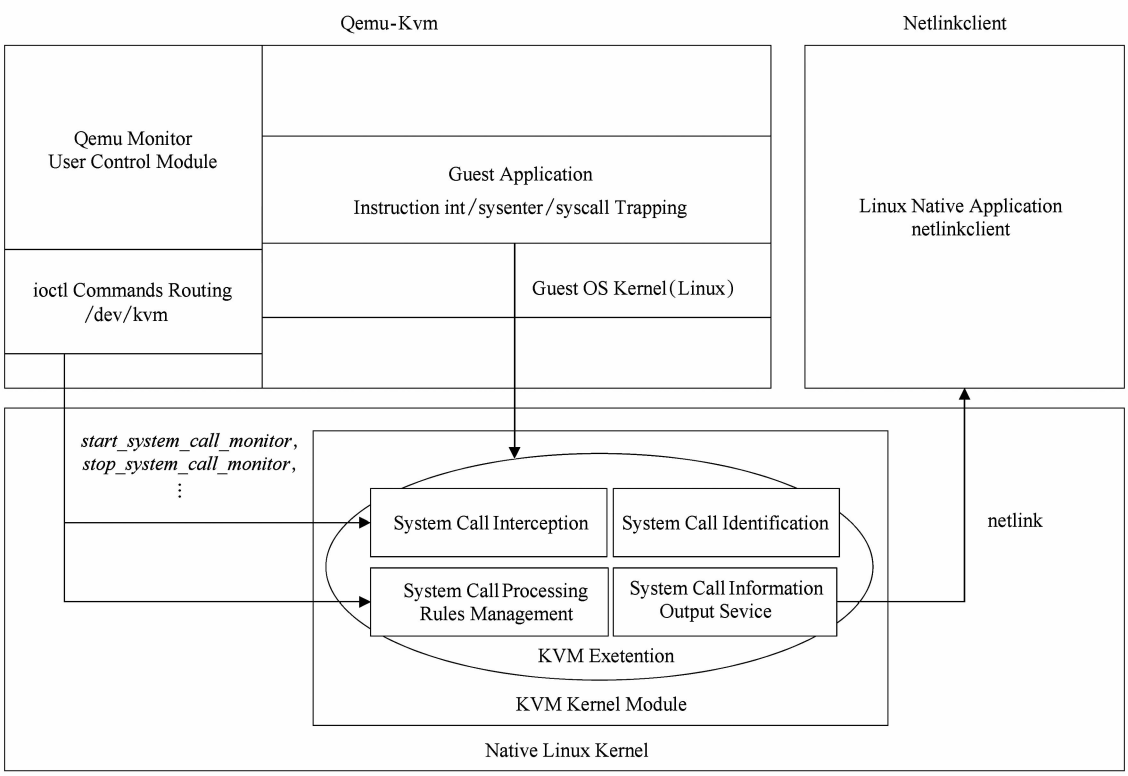


Fig. 8 Logic structure of the prototype.

图 8 原型系统逻辑结构

Qemu Monitor 操作控制模块实现用户接口，通过它的命令行操作窗口可动态开启与关闭 VMM 的系统调用截获机制以及增加删除系统调用处理规则等，内部通过向/dev/kvm 发送 ioctl 命令来实现的.

基于 KVM 的功能扩展模块是实现上节当中关于 3 种系统调用截获、识别与系统调用处理，直接响应来自 Qemu Monitor 所发出的命令，同时为了便于查看系统调用处理信息，它还创建一个 netlink 连接，将所有截获系统调用需要输出的信息都转发给它，并由 netlinkclient 读取.

基于 netlink 的输出模块 netlinkclient 负责读取输出 KVM 功能扩展模块中关于系统调用的处理信息.

3.2 系统具体实现

本原型基于 Qemu-Kvm 0.13 和 KVM 2.6.38.rc7. 其中 Qemu-Kvm 修改部分涉及 monitor.c, qemu-monitor.hx，而 KVM 模块则需要增加系统调用截获、识别以及处理规则管理等功能，下面描述它们的具体实现.

3.2.1 Qemu Monitor 用户操作控制模块

在 Qemu-Kvm 中增加处理 Guest OS 系统调用截获控制，处理规则定义等用户操作接口. 其中在 monitor.c 添加对 KVM 功能扩展模块的 ioctl 命令，它们都通过 *kvm_vm_ioctl* 与/dev/kvm 交互，最终由 KVM 功能扩展模块具体执行. 这些接口包括系统调用截获的开启与关闭，如 *start_system_call_monitor*, *stop_system_call_monitor*；系统调用处理规则的增加、删除等，如 *add_system_call_monitor_rule*.

由于所有命令实现方法类似，这里仅以系统调用截获开启命令 do_start_system_call_monitor 为例进行说明，如图 9 所示：

```
static void do_start_system_call_monitor(…) {
⋮
  if (kvm_vm_ioctl(…, KVM_START_SYSTEM_CALL_
    MONITOR,…))
⋮
}
```

Fig. 9 The *do_start_system_call_monitor* control command.

图 9 *do_start_system_call_monitor* 控制命令

对参数预处理之后，通过 *kvm_vm_ioctl* 向/dev/kvm 发送 KVM_START_SYSTEM_CALL_MONITOR 命令，告诉 KVM 功能扩展模块开启对 Guest OS 系统调用的截获，之后 Guest OS 中应用程序执行的系统调用都会陷入到 VMM 当中.

为了操作执行 *do_start_system_call_monitor*，需要在 qemu-monitor.hx 代码文件中添加对应的命令接口，如 *start_system_call_monitor*，其模板代码如图 10 所示，其中包括参数类型与对应的命令处理函数描述. 例如 *start_system_call_monitor*，其参数为 *idt_index* 中断索引号以及系统调用号使用的寄存器(可以是 rax,rbx,rcx 以及 rdx)，其所对应的操作则为 *do_start_system_call_monitor*.

```
{
.name="start_system_call_monitor",
.args_type="idt_index:i,syscall_reg:s",
.params="idt_index [rax|rbx|rcx|rdx]",
.help="start system call monitoring",
.mhandler.info=do_ start_system_call_monitor,
}
```

Fig. 10 The *do_start_system_call_monitor* command template.

图 10 *do_start_system_call_monitor* 命令模板

3.2.2 KVM 功能扩展模块

在 KVM 功能扩展模块(KVM extension)中需要增加 Guest OS 三种系统调用机制的截获(system call interception)、识别(system call identification)、监控规则管理(system call processing rules management)以及输出(system call information output service)等功能服务.

1) 系统调用截获控制模块

此模块负责完成 3 种系统调用机制截获开启与关闭. 其中开启包括：①设置 GP 异常陷入；②设置 int 陷入截获；③设置 sysenter 陷入；④设置 syscall 陷入截获.

① 设置 GP 异常陷入

对于每个虚拟处理器，设置其 VMCS 控制结构，使其在出现 GP 通用保护错时产生虚拟机陷入 VM exit，为基于 int 和 sysenter 指令的系统调用截获作准备，代码片段如图 11 所示：

```
kvm_for_each_vcpu(i, vcpu, kvm) {
⋮
/* Setting GP to cause VM exit */
  kvm_x86_ops->set_gp_trap(vcpu);
⋮
}
```

Fig. 11 Setting GP fault to VM exit.

图 11 设置机器 GP 通用保护错 VM exit

② 设置 int 陷入截获

读取每个虚拟处理器的中断描述符表并保存,重设中断描述符表的界限为 32,这样在虚拟处理器执行到大于 32 的 int 软中断(如 Linux 系统调用使用 128 号软中断,Windows 使用 46 号软中断)时会导致 GP 通用保护错异常陷入,代码片段如图 12 所示:

```
kvm_for_each_vcpu(i, vcpu, kvm) {
 ⋮
/* Save the current values of IDT for every VCPU */
kvm_arch_vcpu_ioctl_get_sregs(vcpu,&sregs);
/* Reset IDT Limit to 32 */
  sregs.idt.limit=32×kvm-> data.idt_entry_size-1;
  kvm_arch_vcpu_ioctl_set_sregs(vcpu,&sregs);
 ⋮
}
```

Fig. 12 Setting interception for int trap.

图 12 设置 int 陷入截获

③ 设置 sysenter 陷入

保存每个虚拟 VCPU 的 MSR_IA32_SYSENTER_CS 的当前值,然后将其设置为 0. 这样当 Guest OS 执行基于 sysenter 方式的系统调用时,会因为加载 MSR_IA32_SYSENTER_CS 的 0(NULL)值到 CS 寄存器产生 GP 通用保护错异常出现陷入,代码片段如图 13 所示:

```
kvm_for_each_vcpu(i, vcpu, kvm) {
 ⋮
/* Save the current values of MSR_IA32_SYSENTER_CS for
   every VCPU */
  kvm_x86_ops->get_msr(vcpu, MSR_IA32_SYSENTER_
  CS, &(kvm-> data.sysenter_cs_val));
/* Reset MSR_IA32_SYSENTER_CS to 0(NULL) */
  kvm_x86_ops->set_msr(vcpu, MSR_IA32_SYSENTER_CS,
  0);
 ⋮
}
```

Fig. 13 Setting interception for sysenter trap.

图 13 设置 sysenter 陷入截获

④ 设置 syscall 陷入截获

对于每个虚拟 VCPU,关闭其 EFER 的 SCE 标志,使其产生 UD 未定义指令异常陷入,代码片段如图 14 所示.

```
kvm_for_each_vcpu(i, vcpu, kvm) {
 ⋮
/* Set MSR_EFER.SCE to 1,make it cause UD exception */
  kvm_get_msr_common(vcpu, MSR_EFER, &(kvm-> data.
  efer_val));
  kvm_set_msr_common(vcpu, MSR_EFER, kvm->data.efer_
  val & ~EFER_SCE);
 ⋮
}
```

Fig. 14 Setting interception for syscall trap.

图 14 设置 syscall 陷入截获

2) 系统调用识别处理

无论是哪一种系统调用实现机制,在开启截获后它们都会产生系统异常陷入,然后在 VMM 中根据虚拟处理器上下文现场信息进行识别处理. 基本步骤如下:首先要在 VMM 中定位到虚拟机陷入总入口,对于 KVM 即为 *vmx_handle_exit*;其次根据非陷入指令因运行条件改变而产生的异常处理入口 *handle_exception*,其中基于 int 和 sysenter 指令方式的会进入通用保护错 *handle_gp* 分支,而 syscall 则会进入 *handle_ud*;最后利用虚拟处理器上下文现场信息来识别是因非陷入指令产生还是通常情况下产生的,如果是通常情况,则将异常直接转交给 Guest OS 处理,否则,需要根据应用需求进行特定处理后再返回 Guest OS. 图 15 描述了 VMM 中非陷入指令识别基本流程:

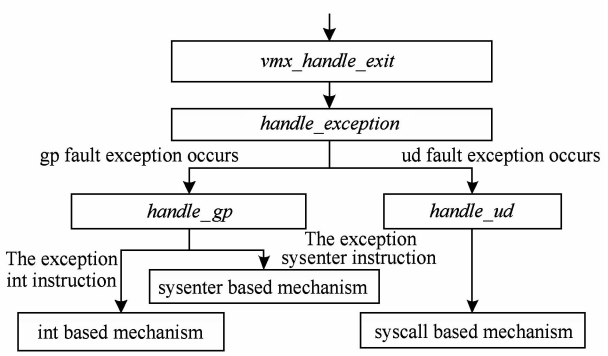


Fig. 15 Identification of three Guest OS system call mechanisms.

图 15 Guest OS 3 种系统调用机制识别处理

3) 系统调用监控规则管理

在识别出 Guest OS 系统调用后需要根据规则进行处理,此模块实现系统调用规则的添加、删除、查看、清除处理,它们在系统调用截获时会逐条加以处理. 最简单的处理包括系统调用上下文信息的输出(VCPU 现场寄存器),这一组操作以/dev/kvm 的 ioctl 接口提供给 Qemu Monitor 进行操作,从而可通过命令来动态增加修改新的系统调用处理规则.

4) 系统调用监控结果输出管理

为了将系统调用处理结果信息进行输出,本文原型在 KVM 内核模块初始化时创建了一个 netlink 链接,然后将所有需要输出的信息通过它转发给下面的 netlinkclient 输出模块,最终由其反馈给用户查看.

3.2.3 netlinkclient

netlinkclient 是一个普通的 Linux 应用程序,它只是打开 KVM 功能扩展模块创建的 netlink 网络链接,读取并输出其中的信息.

4 功能与性能开销实验评估

本节对原型系统中 VMM 截获识别 Guest OS 中 3 个不同系统调用机制的功能及其所带来的性能开销进行验证评估.

所有测试基于的硬件平台为 Intel i7 930 2.8 GHz,内存为 12 GB,其中 int 和 sysenter 方式的宿主操作系统采用 32 b 的 Fedora 13,而 syscall 的宿主操作系统采用 64 b 的 Fedora 13,VMM 采用的版本为 Qemu-Kvm 0.13 和 KVM 2.6.38-rc7,Guest OS 分别采用 32 b 的 Fedora 13(用于测试 int 与 sysenter 方式),64 b 的 Fedora 13(用于测试 syscall 方式).

4.1 功能验证

表 1 给出了测试 Guest OS 三种不同系统调用机制的 Qemu&Kvm 虚拟机启动方法,其中 sysenter 方式需要在启动时添加-cpu qemu64,model=3 命令行选项,以使 Guest OS 认为其运行在具有 sysenter 特性的机器上,从而选取 sysenter 指令实现的系统调用机制.

Table 1 The VM Startup Methods of Three System Call Mechanisms

表 1 虚拟机 3 种系统调用机制的启动方法

System Call Mechanism	The Startup Methods of Qemu&Kvm VM
int	#qemu -hda fedora13-x86-32.img -m 1024 -monitor stdio
sysenter	#qemu -cpu qemu64,model=3 -hda fedora13-x86-32.img -m 1024 -monitor stdio
syscall	#qemu-system-x86_64 fedora13-x86-64.img -m 1024 -monitor stdio

这里仅以 int 为例来说明 Guest OS 系统调用截获与识别功能:

1) 在 Linux 的 shell 命令行中敲入如下命令,启动 Qemu&Kvm 虚拟机,此时 Guest OS 默认采用基于 int 系统调用机制:

#qemu -hda fedora13-x86-32.img -m 1024 -monitor stdio,

其中,选项-hda fedora13-x86-32.img 代表 32 位 Guest OS-Fedora 13 的磁盘镜像,-m 1024 表示创建的虚拟机逻辑物理内存为 1 024 MB,-monitor stdio 代表启动时同时运行 Qemu Monitor,通过它来开启或关闭 VMM 中的系统调用截获与识别机制.

2) 在虚拟机启动之后,切换到 Qemu Monitor 命令行窗口,使用命令 add_system_call_monitor_rule 添加系统调用处理规则,操作如下:

(qemu)add_system_call_monitor_rule rax 3 any 0 hex.

在这条规则中,条件寄存器为 *rax*(在 Linux 操作系统中,它存放系统调用号),当 *rax*=3 时,代表截获 read 系统调用,any 0 表示输出所有通用寄存器现场信息,而 hex 表示以 16 进制输出.注意,本原型验证系统并不真正对截获的系统调用作特定的处理,只是将截获到系统调用时的处理器通用寄存器现场信息简单输出,只验证 3 种系统调用截获识别方法的正确性.实际应用中应该结合具体要求,尽可能地挖掘抽取信息并根据结果进一步处理,比如阻止某个特定系统调用的执行,杀死取消发出系统调用的 Guest OS 进程等.

3) 完成系统调用处理规则初始化之后,通过 Qemu Monitor 命令 start_system_call_monitor 正式开启 VMM 中的系统调用截获机制,具体如下:

(qemu)start_system_call_monitor 128 rax,

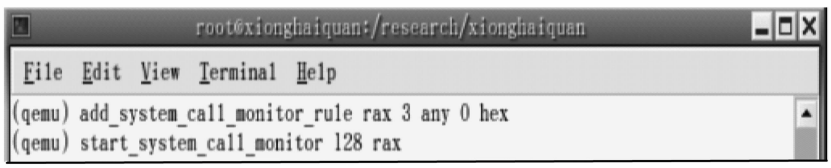
其中 128 是 Linux 系统调用使用的软中断号,表示要对其进行截获,而在 Windows 下为 46,系统调用号用 rax 寄存器存放.这条命令是通过 ioctl 方式向 KVM 功能扩展模块发出的,由其来实际开启所有 3 种系统调用截获机制,也即 Guest OS 中不管以哪种方式启用的系统调用机制之后都会被 VMM 截获.

4) 系统调用机制开启截获后,VMM 会对截获到的系统调用依据第 2 步的处理规则进行处理,并将结果不断地通过 netlink 连接发送,之后由 netlinkclient 连续读取并输出,命令如下:

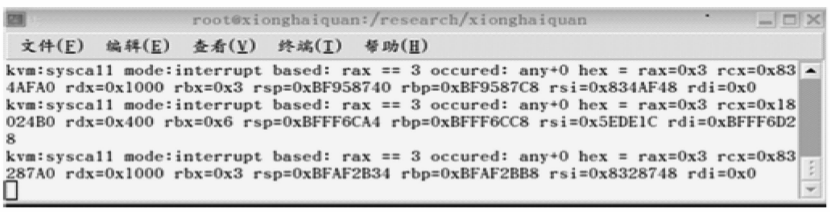
#netlinkclient.

除了步骤 1)启动方法有些不同,sysenter 与 syscall 截获验证操作步骤和 int 方式一样.图 16 显示的是 3 种不同虚拟机启动方法下 Guest OS 3 种

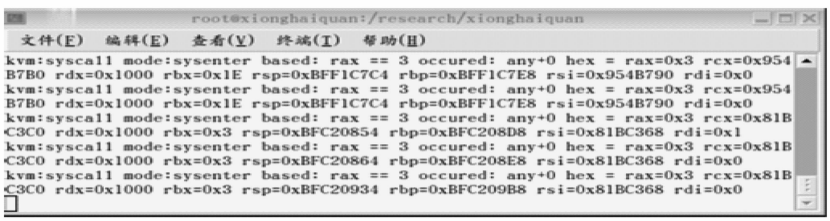
系统调用机制截获与识别运行截图,其中包括 Qemu Monitor 命令窗口和 netlinkclient 在 3 种不同系统调用方式下的信息输出. 3 种输出结果显示为 sycall mode:interrupt based,sysenter based 以及 syscall based,由此可见,以本文第 2 节为基础的技术方法在 Qemu&Kvm VMM 平台上实现的原型能成功截获与识别 Guest OS 中所有 3 种不同系统调用行为,即 int 方式、sysenter 方式以及 syscall 方式.



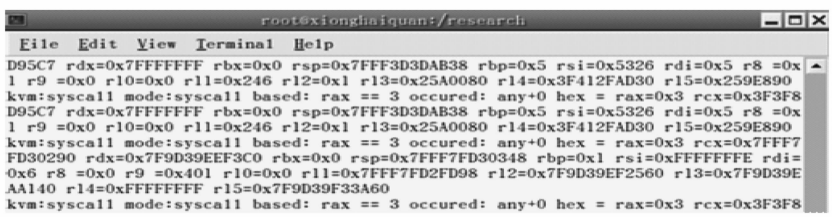
(a) Qemu monitor console



(b) Interrupt based system call



(c) Sysenter based system call



(d) Syscall based system call

Fig. 16 Qemu monitor and the functional verification of three system call mechanisms interception and identification.

图 16 Qemu 监控操作及 3 种系统调用机制截获与识别功能验证

4.2 性能开销实验评估

为了评估 Guest OS 三种不同系统调用在 VMM 中截获与识别所带来的性能开销,本节从 unixbench 中挑选了 3 种测试用例:syscall,arithmetic 以及 shell. 其中 syscall 测试用例执行期间会循环调用系统调用,它是一种极限测试场景,用于测试系统调用最大性能开销;arithmetic 则属于计算型测试用例,其运行期间以运算为主,即 Guest OS 几乎不发出系统调用请求,这可看作是与 syscall 相反的另一种极限测试场景,用于测试在没有系统调用时带来的性能开销;而 shell 则介于两者之间,即运行期间会发出系统调用,也含有不少的计算,因此可看作是正常应用的代表.

以下 3 个表格测试数据都是在 Guest OS 中获得的. 每个表格都包含 3 种不同系统调用在关闭与开启截获前后的性能指数,其中性能开销比反映的是关闭和开启截获的比值,比值越大代表开销越大. 如表 2 的 syscall 测试用例中,基于 int 方式在关闭截获时性能指数为 538,开启之后为 38.9,故其性能开销比例为 13.830=538/38.9.

Table 2 syscall Test Case

表 2 syscall 测试用例

System Call Mechanism	Disabled	Enabled	Performance Overhead Ratio
int	538	38.9	13.830
sysenter	1 245.3	12.4	100.427
syscall	1 229.9	20.3	60.586

在表 2 syscall 测试用例中,基于 int,sysenter 以及 syscall 三种系统调用截获开启前后的性能开销比例分别是 18.830,100.427 以及 60.586,这种结果表明在 VMM 中截获 Guest OS 系统调用行为会产生很大性能开销,不过如前所述这是一种极限情形,即其运行期间始终是循环执行系统调用.另外 3 种方式中性能开销最大的是 sysenter 方式,主要原因是其不仅增加了陷入截获识别开销,同时它还需要加上仿真模拟开销;表 3 可以看作是正常应用情形,性能开销比分别为 1.900,2.608,2.195,这种结果表明在实际应用中,系统调用截获并没有出现数量级的下降,其性能开销在可接受范围之内;表 4 则是另一种极限情形,即运行期间几乎不发生任何系统调用,此时在 VMM 中无论是开启还是关闭 Guest OS 中的系统调用截获都不会产生性能损失,与预期目标吻合,即如果 Guest OS 不执行系统调用就不应该产生额外的性能开销.

Table 3 shell Test Case

表 3 shell 测试用例

System Call Mechanism	Disabled	Enabled	Performance Overhead Ratio
int	1 203.3	633.3	1.900
sysenter	1 238.8	475.0	2.608
syscall	1 213.3	552.8	2.195

Table 4 arithmetic Test Case

表 4 arithmetic 测试用例

System Call Mechanism	Disabled	Enabled	Performance Overhead Ratio
int	545.2	545.3	0.999 8
sysenter	548.1	548.0	1.000 2
syscall	635.2	634.2	1.001 6

5 结 论

本文提出的基于硬件体系软件规范来改变非陷入指令运行条件使其产生陷入的思想,相比已有方法它具有跨平台透明、无需修改 Guest OS 的优点;据此思想所提出的 x86 架构下 Guest OS 中 3 种非陷入系统调用指令截获与识别的技术方法在 Qemu&Kvm 平台实现的原型系统上得到了验证,并在正常情况下所产生的性能开销也在实际应用可接受的范围之内.该思想方法不仅对于系统调用是适用的,其他与之类似场景都可进行对应的处理;目前本原型系统只实现了 VMM 中 Guest OS 系统调用截获与识别处理基本框架机制;在进一步的研究中,需要与实际应用需求结合起来,定义系统调用的具体处理规则,这样就可以在虚拟机之外实现 Guest OS 中进程级行为细粒度的监控.

参 考 文 献

[1] Garfinkel T, Rosenblum M. A Virtual machine introspection based architecture for intrusion detection [C] // Proc of the 10th Annual Network and Distributed System Security Symp (NDSS'2003). Washington: ISOC, 2003: 191-200

[2] Pfoh J, Schneider C, Eckert C. Exploiting the x86 architecture to derive virtual machine state information [C] // Proc of the 4th Int Conf on Emerging Security Information, Systems and Technologies (SECURWARE'2010). Piscataway, NJ: IEEE, 2010: 166-175

[3] Pfoh J, Schneider C, Eckert C. A formal model for virtual machine introspection [C] //Proc of the 1st ACM Workshop on Virtual Machine Security. New York: ACM, 2009: 1-10

[4] Popek G J, Goldberg R P. Formal requirements for virtualizable third generation architectures [J]. Communications of the ACM, 1974, 17(7): 412-421

[5] Prosnitz B. Blackbox no more: Reconstruction of internal virtual machine state [OL]. (2007-03-26) [2013-03-21]. http://virtuoso.cs.northwestern.edu/NWU-EECS-07-01.pdf

[6] Onoue K, Oyama Y, Yonezawa A. Control of system calls from outside of virtual machines [C] //Proc of the 2008 ACM Symp on Applied Computing (SAC'2008). New York: ACM, 2008: 2116-1221

[7] Forrest S, Hofmeyr S, Somayaji A. The evolution of system-call monitoring [C] // Proc of the 2008 Annual Computer Security Applications(ACSAC'2008). Piscataway, NJ: IEEE, 2008: 418-430

[8] Li Bo, Li Jianxin, Wo Tianyu, et al. A vmm-based system call interposition framework for program monitoring [C] // Proc of the 16th IEEE Int Conf on Parallel and Distributed Systems (ICPADS'2010). Piscataway, NJ: IEEE, 2010: 706-711

[9] Jiang Xuxian, Wang Xinyuan, Xu Dongyan. Stealthy malware detection and monitoring through vmm-based "out-of-the-box" semantic view reconstruction [J]. ACM Trans on Information and System Security (TISSEC), 2010, 13(2): 1-28

[10] Li Bo, Li Jianxin, Hu Chunming, et al. Software integrity verification based on vmm-level system call analysis [J]. Journal of Computer Research and Development, 2011, 48(8): 1438-1446 (in Chinese)

(李博,李建欣,胡春明,等. 基于 VMM 层系统调用分析的软件完整性验证[J]. 计算机研究与发展,2011,48(8):1438-1446)

[11] Li Zhen, Tian Junfeng, Yang Xiaohui. Program behavior monitoring based on system call attributes [J]. Journal of Computer Research and Development, 2012, 49(8): 1676-1684 (in Chinese)
(李珍,田俊峰,杨晓晖. 基于系统调用属性的程序行为监控[J]. 计算机研究与发展,2012,49(8):1676-1684)

[12] Jones S T, Arpaci-Dusseau A C, Arpaci-Dusseau R H. Antfarm: Tracking processes in a virtual machine environment [C] //Proc of USENIX Annual Technical Conf, General Track (USENIX'2006). Berkeley: USENIX Association, 2006: 1-14

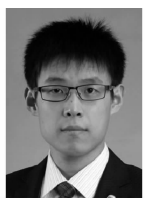
[13] Bellard F. QEMU, a fast and portable dynamic translator [C] //Proc of USENIX Annual Technical Conf, FREENIX Track (USENIX'2005). Berkeley: USENIX Association, 2005: 41-46

[14] Kivity A, Kamay Y, Laor D, et al. KVM: The Linux virtual machine monitor [C/OL] //Proc of the Linux Symp. (2007-07-26) [2013-03-21]. https://www.kernel.org/doc/mirror/ols2007v1.pdf#page=225

Xiong Haiquan, born in 1976. PhD. Student member of China Computer Federation. His main research interests include operating system and system virtualization technology.

Liu Zhiyong, born in 1946. Professor and PhD supervisor. Senior member of China Computer Federation. His research interests include high performance algorithm and architecture, parallel processing and memory system.

Xu Weizhi, born in 1982. PhD. Postdoctoral researcher of Institute of Microelectronics, Tsinghua University. His research interests include high performance algorithm and architecture.



Tang Shibin, born in 1986. PhD. Student member of China Computer Federation. His research interests include high performance computer architecture, debugging parallel program and on chip memory system.

Fan Dongrui, born in 1979. Professor and PhD supervisor, His main research interests include many-core processor design.