·计算机工程与应用·

基于Xen的自下而上调用的设计与实现

陈兴蜀，李 辉，张 磊，任 益

(四川大学计算机学院 成都 610064)

【摘要】针对虚拟机监视器(virtual machine monitor, VMM)与上层客户虚拟机(Guest-VM)之间的语义鸿沟(semantic gap)问题，该文提出了一种自下而上的调用方式，该方法使得VMM能够同步调用客户机的已有功能来获取客户机语义信息，为在客户虚拟机地址空间之外的监控机制带来便利。在Xen半虚拟化环境下，实现了自下而上的同步调用方式，有效地解决了语义重构所需的重复定义和实现问题。实验表明，该方法能使VMM有效地调用客户机的已有功能为自己服务，使VMM能准确地获取上层虚拟机操作系统的信息。

关 键 词 内核通信； 语意鸿沟； 虚拟机； 虚拟机监视器； Xen

中图分类号 TP316 **文献标志码** A doi:10.3969/j.issn.1001-0548.2014.06.015

The Design and Implement of Xen-Based Upcall

CHEN Xing-shu, LI Hui, ZHANG Lei, and REN Yi

(School of Computer Science, Sichuan University Chengdu 610064)

Abstract In order to solve the semantic gap problem between virtual machine monitor (VMM) and guest virtual machine (VM), an up-call mechanism is proposed, with which the service request is launched by VMM, and the guest-VM is responsible to provide response to the request from VMM. This makes it possible for VMM to synchronously call guest-VM services to get guest semantic information, bringing convenience for monitor agents constructed out of guest-VM address space to get guest information exactly. The up-call mechanism implemented as a synchronous communication channel is able to make up the problem of duplicate definitions and implementations inside VMM brought by semantic reconstruction. A prototype system on the para-virtualization platform of Xen is implemented. The result shows that the method proposed in this paper is able to help VMM to call guest functions to get guest services and semantic information instantly.

Key words kernel communication; semantic gap; virtual machine; virtual machine monitor; Xen

现今，虚拟化技术已被广泛用于解决系统安全问题，相对于客户机操作系统，虚拟机监视器具有更小的可信计算基(TCB)[1]，运行在更高的特权级时，不易被攻击。因此，很多研究都选择将监控代码置于VMM中[2-3]。文献[4]则利用隔离技术将监控代码与客户机隔离，该方法能够有效地防止来自恶意客户机的攻击。虽然这种外部(out-of-VM)监控方式可以提高监控代码的抗干扰能力，但却面临着虚拟机监视器与客户机之间的语义鸿沟问题[5]，使VMM获取客户机信息变得困难。

当前研究倾向于使用语义重构技术(semantic reconstruction)[6-7]，在VMM中重新构造客户机内核数据对象，并实现代码来维护这些数据结构，如重构客户机文件系统和内存信息映射[7]。文献[8]采用追踪客户机在实现某一功能时所执行的机器指令，根据这些指令重新构造可被VMM执行的机器代码，VMM通过调用这些重构的代码来实现同样的功能。这种基于重构的解决方式需要在VMM中重新定义数据结构并实现代码来维护它们，并且客户机操作系统已经具备同样的定义且功能更加完善，若能在VMM中直接调用客户机的已有功能，就可省去在VMM中重新定义类似数据结构以及用以维护和管理的代码实现。

为此，SymCall[9]实现了一种基于硬件虚拟化[10]的自下而上[11]调用方式，通过重用客户机的已有功能，让VMM可根据需要调用上层客户机函数为自己服务。然而，该方法需要CPU对虚拟化的支持，并且SymCall只支持在退出VMM时调用客户机功能，

收稿日期：2013– 05 – 27；修回日期：2014 – 01 – 30
基金项目：国家科技支撑计划(2012BAH18B05)
作者简介：陈兴蜀(1968 –)，女，博士，教授，主要从事信息安全、计算机网络方面的研究.

致使VMM不能根据需要随意返回到客户机中获取客户机服务。

本文针对VMM与客户机之间的语义鸿沟问题，实现了基于Xen半虚拟化的自下而上调用方式，使得Xen可以利用客户机的已有功能为自己服务。相比于语义重构，本文提出的方法省去了在Xen中重新定义客户机数据结构以及用于维护这些数据结构的代码实现，同时无需担心Xen所维护的数据与客户机真实数据之间的同步问题。这种基于客户机API的调用方式可用于客户机在内部功能实现和数据结构方面的变更，具备很强的可扩展性。

1 问题描述

由于VMM和客户机操作系统在设计上的独立性，使VMM只能看到低级语义信息(如内存、磁盘中的二进制数据)，而较少知道隐藏在客户机内的深层含义[12](操作系统级语义)。例如，VMM调度的对象是客户虚拟机，而不关心此时客户机处于哪个进程，VMM能读取内存中客户机的二进制数据，但却无法理解这些数据的语义信息。通常，VMM中用于监控客户机运行的监控工具，需要理解和解析客户机的运行环境。而VMM本身缺乏定义所需数据结构以及用于管理这些信息的代码实现，使得信息的获取和维护变得困难。语义重构技术针对特定的客户机数据对象，在VMM层重新构造该数据对象并实时同步它们与客户机之间的数值。如重构客户机的进程控制块信息，则需要在VMM中给出对应于客户机进程控制块的数据结构定义，并读取客户机进程信息为它们赋值。由于特权级差异以及客户机内核实现的复杂性，该方法在实现上较困难。本文的解决思路是：由于客户机本身已存在这些信息，只要实现由VMM到客户机的自下而上调用过程，监控工具就可直接调用客户机的已有功能来获取监控所需要的信息。

半虚拟化下，Xen提供超级调用与事件通道两种通信机制。超级调用是一种自上而下的同步调用方式，事件通道可用于实现Xen与客户机之间的通信，但作为一种异步通信机制，无法满足实时性需求。基于此，本文提出了一种自下而上的同步通信方式，用于实现由Xen到客户机的同步调用，使Xen可以实时获取上层服务。

2 自下而上调用的设计与实现

2.1 过程概述

自下而上调用的总体过程如图1所示，在实现自下而上调用的过程中，本文解决了如下3个问题：

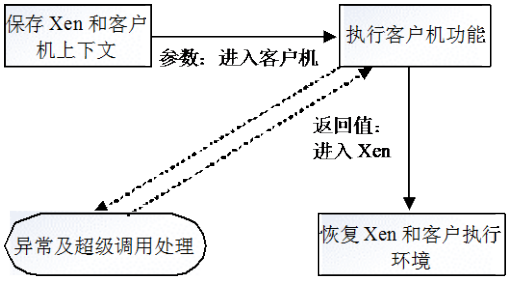

图1 自下而上调用的总体过程

1) 特权级切换：由于CPU不允许高特权级直接对低特权级发起调用，这里通过手动构造iret堆栈结构的方式返回到客户机指定地址中。当客户机执行完Xen所需操作后，仍需返回到Xen中以便Xen处理返回值并恢复Xen的执行环境，这里通过添加软中断的方式来帮助客户机返回到Xen中。

2) 环境保存：在自上而下调用过程中，低特权级priA通过中断或异常陷入高特权级priB后，priA的返回地址将会被CPU自动保存到priB的堆栈中。在高特权级priB执行过程中，无需担心priA的堆栈数据会被破坏。而在自下而上调用过程中，环境的保存与恢复都需要手动进行，且因为所调用的低特权级服务并不具备所有的操作权限，在系统通过自下而上调用进入到低特权级priA中时，priA所引发的中断或异常仍会引起系统再次陷入到priB中，造成对priB执行环境的破坏。因而，除了要保存客户机以及Xen的寄存器上下文之外，Xen和客户机的运行时堆栈也需要被保护。32位下的Xen使用4个内存页作为运行时堆栈，为避免因保存Xen堆栈内容而引起过多的内存拷贝，本文采用了堆栈切换的方式(切换后的堆栈记为$stack_{new}$，Xen原来的堆栈记为$stack_{old}$)，让系统在陷入Xen后使用不同的堆栈，而客户机堆栈的保护则可通过设置正确的客户机esp数值来实现。

3) 异常及超级调用处理：系统处于自下而上调用过程中时，中断处于屏蔽状态。屏蔽中断会影响Xen对客户机的异常处理，并且Xen的异常和超级调用处理流程都会更改中断标志位，所以需要对该过程做出修改。同时，为缩短自下而上处理时间，本文省略了Xen在返回客户机前对客户机事件的处理。

2.2 过程实现

CPU为每个进程都设置了TSS(task state segment)结构，用来保存当前进程的基本状态信息。x86的分段机制为程序运行设置了4个特权级，每个特权级都有自己特定的堆栈，TSS中保存着运行在特权级0、1、2上的程序的堆栈初始地址(TSS.ESPn)。当系统从低特权级向高特权级切换时，CPU会从TSS中取出TSS.ESPn并加载到ESP寄存器中，完成堆栈

切换，而系统原来使用的堆栈地址则被压入到切换后的堆栈中以便在返回时恢复。同样，Xen为每个CPU都维护着一个tss_struct结构，保存在全局变量init_tss数组中，init_tss[processor_id].esp0便指向Xen堆栈的初始地址。当系统从低特权级切换到Xen时，CPU便把init_tss[processor_id].esp0加载进esp寄存器中，实现切换到Xen堆栈的过程。通过修改该值，便可实现系统在陷入Xen后使用不同堆栈的目的。类似的，init_tss[processor_id].esp1则指向客户机的堆栈初始地址。

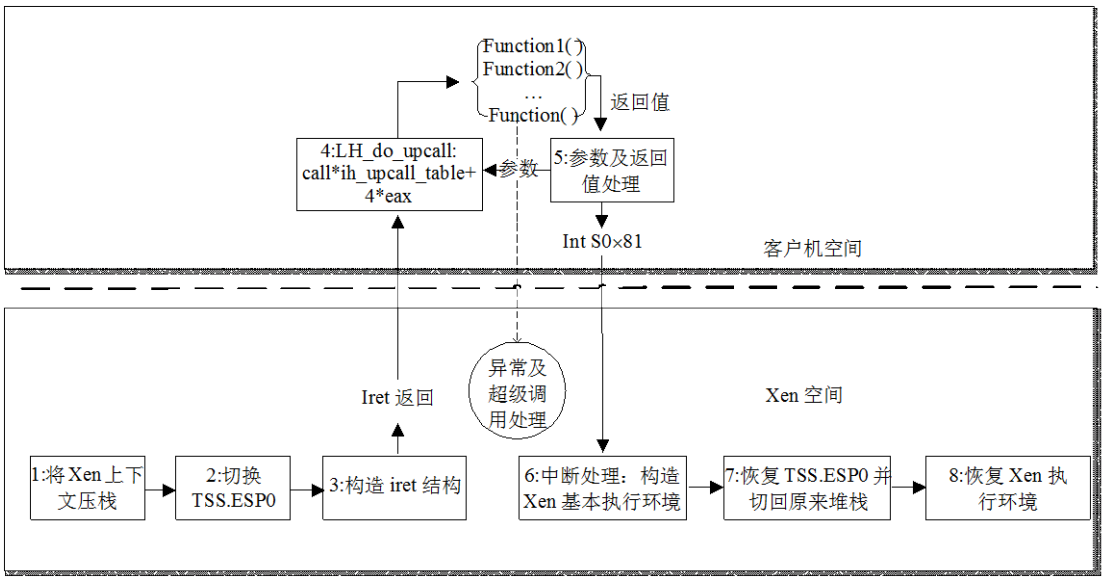

图2　自下而上调用的实现过程

基于Xen的半虚拟化环境下，自下而上调用的实现过程如图2所示，各个步骤的实现及原理介绍如下：

1) Xen在通过iret指令返回客户机地址空间之前，各个寄存器的值需要被保存，以便随后对Xen执行环境的恢复。该部分数据被压入到Xen的$stack_{old}$栈中，其中，esp和eip被保存在全局变量中。

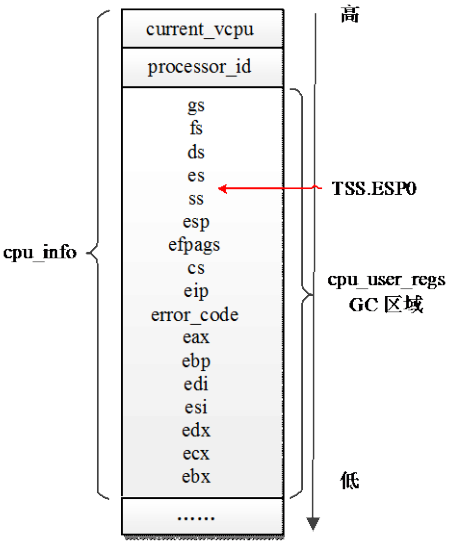

图3　Xen堆栈结构

2) 在切换TSS.ESP0(Xen的初始esp指针)指向的堆栈前，需从$stack_{old}$栈中拷贝数据到$stack_{new}$栈中。Xen堆栈结构如图3所示，客户机上下文被Xen定义为cpu_user_regs结构，与Linux操作系统一样，Xen也在堆栈顶端保存了指向当前VCPU结构的current_vcpu指针以及多处理器下的processor_id，一起构成cpu_info结构。

Xen将TSS.ESP0指向到cpu_user_regs结构的es寄存器所在地址，在特权级切换到Xen时，Xen将会把客户机寄存器上下文保存到cpu_user_regs区域，随后CPU需要从堆栈顶部读取current_vcpu和processor_id的数值。因而，在修改TSS.ESP0所指向的Xen堆栈前，current_vcpu和processor_id数值需要被复制到$stack_{new}$对应的位置中。同样，申请4个连续内存页作为新的堆栈$stack_{new}$，在$stack_{new}$上构造cpu_info结构，将current_vcpu和processor_id复制到$stack_{new}$的顶端，同时将$stack_{new}$上的es寄存器所在地址赋值给TSS.ESP0。

表1　iret的构造过程

语句	伪代码
1	IF(RPL(GC.cs) == 3)　//由应用层陷入Xen？
2	THEN iret.esp = init_tss[processor_id].esp1 /*获取客户机内核堆栈初始地址*/
3	ELSE iret.esp = GC.esp　//否则，则由客户机内核陷入
4	iret.eip = LH_do_upcall
5	iret.ss = current->arch.guest_context.kernel_ss
6	iret.cs = __KERNEL_CS

3) 通过执行iret指令来发起对客户机指定函数的调用，iret的构造过程如表1所示。如图3所示，客户机在陷入Xen后，寄存器上下文被保存在GC区域。为避免发起的自下而上调用对客户机堆栈环境造成破坏，在构造iret指令所需esp数值时需考虑两种情况：若当前系统是由应用层陷入到Xen中(如通过异常)，此时客户机内核堆栈处于初始状态，客户机堆栈指针数值需从init_tss[processor_id].esp1中获取；若系统是由客户机内核陷入到Xen中，表明客户机内核堆栈已包含运行时数据，此时客户机堆栈数值应从GC.esp成员中获取，避免了从TSS.ESP1中获取客户机堆栈数值，导致随后产生的运行时数据对客户机运行环境造成破坏。本文通过添加超级调用的方式，在客户机启动时将客户机处理自下而上调用的函数地址(LH_do_upcall)传递给Xen，此地址被用作iret结构的eip数值。同时借鉴超级调用的参数传递方式，以eax作为事件号用来标记不同的事件，ebx、ecx、edx、esi、edi被用来传递函数调用所需的参数。

4) 通过以上构造的iret堆栈结构，Xen便可执行iret指令返回到客户机LH_do_upcall函数中，该函数在设置好客户机运行环境后，根据eax寄存器的值来获得Xen要求客户机所执行的操作，LH_do_upcall部分代码如表2所示。类似于超级调用表，定义一张自下而上调用表(lh_upcall_table)用来保存处理Xen对应操作的各函数的入口地址，自下而上调用表如表3所示。

表2 LH_do_upcall部分代码

语句	代码
1	ENTRY(LH_do_upcall)
2	1: pushl %eax
3	pushl %ebp
4	pushl %edi
5	pushl %esi
6	pushl %edx
7	pushl %ecx
8	2: pushl %ebx
9	cld
10	……
11	movl %eax,%ebx
12	movl %esp,%eax
13	call *lh_upcall_table(,%ebx,4)
14	……

5) 如上所述，通过寄存器传递所需参数。Xen设置好各参数数值并返回到LH_do_upcall中后，各个寄存器的数值需要进一步传递到对应处理函数中。如表2所示，标号1和2之间的压栈操作将动态构建如下结构体实例：

表3 自下而上调用表

语句	汇编代码
1	ENTRY(lh_upcall_table)
2	.long 0
3	.long LH_xen_test
4	.long LH_xen_hypercall_test
5	.long LH_xen_exception_test
6	.long LH_xen_getpid
7	.long LH_xen_prepare_proc #5
8	.long LH_xen_write_proc

```
struct LH_xen_upcall
{
unsigned long ebx,ecx,edx,esi,edi,ebp,eax;
} regs;
```

lh_upcall_table表中定义的各个函数均以struct LH_xen_upcall *regs作为参数，regs指向堆栈中ebx寄存器所在地址，使得各个处理函数均可方便访问寄存器数值。同时，regs指针作为各个处理函数的输出参数，也保存各个处理函数的返回值和其他需要反馈给Xen的信息。

6) 客户机在完成Xen所需操作后，便可通过执行int $0x81指令进入到添加的中断处理程序中，通过特权级切换，先前设定的TSS.ESP0已被加载到esp寄存器中，堆栈切换到了$stack_{new}$。该中断处理程序所完成的操作如下：

① 设置Xen运行所需的基本执行环境；

② 把TSS.ESP0修改为原来的数值，以便随后系统在陷入Xen后使用原来的堆栈；

③ 保存客户机的返回值；

④ 将堆栈切换回$stack_{old}$，释放先前用做临时堆栈的内存页；

⑤ 恢复Xen的寄存器上下文；

⑥ 将保存在全局变量中的eip数值压入到堆栈中，执行ret指令弹出栈上eip数值到指令寄存器当中，如此便完成了对Xen执行环境的恢复；

⑦ 处理客户机的返回信息。

3 异常及超级调用处理

当客户机处于自下而上执行环境中时，由于客户机不具备最高特权级，致使客户机在执行过程中仍会陷入到Xen中，由客户机主动发起的超级调用和在执行过程中CPU所抛出的异常都会引发客户机的

陷入。本文所实现的自下而上调用过程中的中断处于屏蔽状态，该过程会影响Xen对异常和超级调用的处理。Xen异常处理流程如图4所示，若检测到系统触发异常，中断处于屏蔽状态，便会跳转到exception_with_ints_disabled处理该异常。若检测到该异常来自客户机，Xen则认为客户机在非常时期触发了异常，便会调用fatal_trap函数打印出错信息。

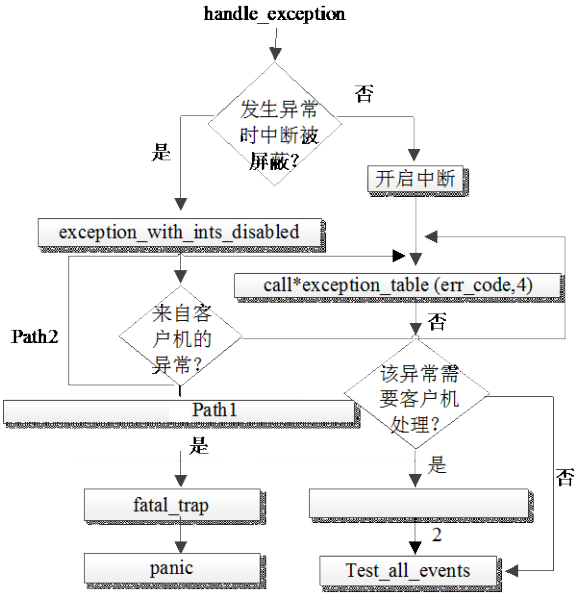

图4　Xen异常处理流程

为使Xen在自下而上调用环境中仍能处理客户机异常，在domain结构体中增加字段lh_upcall_flags，在Xen发起自下而上调用时将该字段赋值为11，自下而上检测代码如表4所示，用于检测系统是否处于自下而上调用环境中。若检查到系统处于该环境，异常处理流程不经过Path1而通过Path2，Xen按照正常异常处理路径处理该异常。同时，在异常处理程序和超级调用响应例程中，在所有开启中断的指令sti以及对中断开启的断言ASSERT_INTERRUPTS_ENABLED语句前都插入上述UPCALL_CHECK代码，若系统处于自下而上环境，上述开启中断的指令和断言将被忽略跳过而直接执行。在自下而上的调用过程中，屏蔽中断可能会导致系统不能及时响应甚至丢失系统所产生的中断，所以该过程所执行的操作应尽量简短。无论是异常还是返回超级调用，Xen都会跳转到test_all_events中去处理客户机事件，若客户机有大量事件需要处理，该过程会大量消耗CPU时间。为减少时间消耗，UPCALL_CHECK同样被插入到test_all_events中，Xen将跳过对客户机事件的处理，将客户机事件推迟到自下而上结束后再处理。

表4　自下而上检测代码

语句	汇编代码
1	#define UPCALL_CHECK　\
2	pushl %ebx;　\
3	pushl %eax;　\
4	GET_CURRENT(%ebx);　\
5	movl VCPU_lh_upcall_flags(%ebx),%eax;\
6	cmpl $11,%eax;　\
7	popl %eax;　\
8	popl %ebx;　\

4　实验测试

本文在Xen 3.4.4-Linux 3.1.10半虚拟化环境下实现了自下而上调用的原型系统，并对系统的基本功能进行了测试，自下而上调用测试表如表5所示。其中，调用号1、2、3为基本功能测试，通常情况下，发生异常和超级调用时自下而上调用功能都能正常运转。调用号4、5、6为应用测试，使Xen可通过自下而上调用获取客户机语义信息以及调用客户机的相应功能为自己服务。

表5　自下而上调用测试表

调用号	描述
1	基本功能测试，测试整个自下而上调用流程；
2	超级调用测试，通过自下而上调用返回到客户机后，在客户机中发起超级调用陷入到Xen中；
3	异常测试，通过自下而上调用返回到客户机后，在客户机中引发异常再陷入到Xen中；
4	在Xen中调用客户机服务获得当前进程ID；
5	在proc文件系统下创建目录和数据项；
6	向proc文件系统写入数据。

对异常进行测试时，需在内核中主动引发异常。但在内核中引发异常，如除法错误、非法内存引用引起的缺页异常(如修改只读数据)，大多都会使内核产生panic错误而崩溃，不利于测试的顺利进行。为避免触发不恰当的异常引起系统崩溃，实验中选择对系统调用进行测试。在Xen虚拟机监视器中，通过注释init_int80_direct_trap函数，系统调用首先将会以异常的方式陷入到Xen中，Xen按照通用异常处理流程处理系统调用，可利用系统调用则对Xen的异常处理进行测试，测试代码如表6所示，系统调用引发的异常测试结果如图5所示。同时，系统调用陷入Xen后，可在Xen中监控与拦截客户机的系统调用。测试过程中，利用虚拟机监视器层拦截客户机的sys_unlink系统调用来监控客户机删除的文件名，并通过自下而上调用方式调用客户机函数向Linux的proc文件系统写入文件名，make内核模块时所删除

的文件名如图6所示。

表6 测试代码

语句	代码
1	int num = 20;
2	_asm_volatile(“int $0x80”::”a”(num));
3	_asm_volatile(“”:”=a”(num):);
4	printk(KERN_ERR”lh_xen_exception_test %d”,num);

```
[root@localhost tornado]# cat /var/log/messages| grep lh_xen_exception_test
May  3 17:03:32 localhost kernel: lh xen exception test 1779
```

图5 系统调用引发的异常测试结果

```
[root@localhost upcall]# cat /proc/upcall/xen_upcall
/dev/.udev/db/\x2fmodule\x2fupcall
/home/tornado/.gconfd/saved_state.orig
/tmp/ccVh96nc.s
/tmp/cc97fUQq.s
/tmp/ccmZOx8s.s
/tmp/ccRZhqDE.s
/tmp/cci8FxUG.s
/tmp/ccHvHWfJ.s
/tmp/ccc01gzL.s
/tmp/cc3Bt9UN.s
/home/tornado/sample/upcall/.1847.tmp
/home/tornado/sample/upcall/.1863.tmp
/tmp/ccR2iiYj.s
/tmp/cc5ubaTh.ld
/tmp/ccOAMYsk.le
/tmp/ccrZheJc.c
/tmp/ccChgfjf.o
/tmp/ccekqI2d.o
/tmp/ccM3UCaa.s
/tmp/ccmRf8ov.ld
/tmp/ccw5XguC.le
/tmp/cc4p3zeh.c
/tmp/ccyE2Tjo.o
/tmp/ccgqOMWo.o
/tmp/ccTZFBCf.s
```

图6 make内核模块时所删除的文件名

在上述测试中，本文所实现的原型系统均能正常运行，整个过程未对客户机的运行环境造成破坏，同时Xen的执行环境在自下而上调用前后均被正常保存和恢复。自下而上调用的目的是帮助Xen快速调用上层客户机服务、屏蔽中断，使得自下而上调用被串行化，不会出现嵌套过程，使得客户机可集中处理来自Xen的服务请求。然而，这也限制了所调用的客户机函数不能引起系统睡眠或阻塞以及进程的上下文切换。

5 结 束 语

本文提出了一种基于Xen半虚拟化的自下而上的调用方式，并实现了原型系统。解决了在自下而上调用环境中的环境保护、参数传递、异常处理和超级调用问题，并对所实现的原型系统进行了功能测试。实验结果表明，本文提出的自下而上调用方法能够保证在不破坏Xen和客户机的执行环境前提下，调用客户机服务并成功返回到Xen中，方便Xen同步调用客户机函数来获取客户机服务和语义信息。然而，在基于虚拟化的解决方案中，如何保证所调用的客户机服务是可信的还需要进一步研究。

参 考 文 献

[1] RUSHBY J. A trusted computing base for embedded systems[C]//Proceedings 7th DoD/NBS Computer Security Conference, [S.l.]: [s.n.], 1984.

[2] AZAB A M, NING P, SEZER E C, et al. HIMA: a hypervisor-based integrity measurement agent[C]// Computer Security Applications Conference. Hawaii, USA: IEEE, 2009.

[3] ABHINAV SRIVASTAVA, JONATHON GIFFIN. Efficient monitoring of untrusted kernel-mode execution[C]// Proceedings of Network & Distributed System Security Symposium. California, USA: [s.n.], 2011.

[4] SHARIF M I, LEE W, CUI W, et al. Secure in-vm monitoring using hardware virtualization[C]//Proceedings of the 16th ACM Conference on Computer and Communications Security. Chicago, USA: ACM, 2009.

[5] CHEN P M, NOBLE B D. When virtual is better than real[C]//Proceedings of the Eighth Workshop on Hot Topics in Operating Systems, Elmau,GERMANY: IEEE, 2001.

[6] DOLAN-GAVITT B, LEEK T, ZHIVICH M, et al. Virtuoso: Narrowing the semantic gap in virtual machine introspection [C]//2011 IEEE Symposium on Security and Privacy (SP). [S.l.]: IEEE, 2011.

[7] JIANG X, WANG X, XU D. Stealthy malware detection through vmm-based out-of-the-box semantic view reconstruction[C]//Proceedings of the 14th ACM Conference on Computer and Communications Security. Alexandria: ACM, 2007.

[8] IBRAHIM A S, HAMLYN-HARRIS J, GRUNDY J, et al. Supporting virtualization-aware security solutions using a systematic approach to overcome the semantic gap[C]// 2012 IEEE 5th International Conference on Cloud Computing (CLOUD). Chicago, USA: IEEE, 2012.

[9] LANGE J R, DINDA P. Symcall: Symbiotic virtualization through vmm-to-guest upcalls[C]//ACM SIGPLAN Notices. [S.l.]: ACM, 2011.

[10] Intel Technical Articles. Intel virtualization technology: Hardware support for efficient processor virtualization [EB/OL]. [2013-05-16]. http://noggin.intel.com/content/intel-virtualization-technology-hardware-support-for-efficient-processor-virtualization.

[11] CLARK D D. The structuring of systems using upcalls [J]//ACM SIGOPS Operating Systems Review, 1985, 19(5): 171-180.

[12] XIONG Hai-quan, LIU Zhi-yong, XU Wei-zhi, et al. Libvmi: a library for bridging the semantic gap between guest OS and VMM[C]//The 12th International Conference on Computer and Information Technology. Chengdu: IEEE , 2012.

编 辑 叶 芳