计算机系统应用 ISSN 1003-3254, CODEN CSAOBN
Computer Systems & Applications,2019,28(3):73−79 [doi: 10.15888/j.cnki.csa.006792]

E-mail: csa@iscas.ac.cn
http://www.c-s-a.org.cn
Tel: +86-10-62661041

系统虚拟化环境下客户机系统调用信息捕获与分析①

宁　强[1,2], 崔超远[1], 李勇钢[1,2]

[1](中国科学院 合肥物质科学研究院 智能机械研究所, 合肥 230031)
[2](中国科学技术大学, 合肥 230026)
通讯作者: 崔超远, E-mail: cycui@iim.ac.cn

摘　要: 针对当前方法无法对系统调用参数和返回值等信息进行捕获和分析的问题, 在 Nitro 的基础上建立了一个实时监视客户机内系统调用的系统. 该系统通过修改硬件规范和指令重写, 实现对快速系统调用进入和退出指令的捕捉和分析. 之后, 结合 VCPU 的上下文信息和系统调用的语义模板解析各参数; 捕获到系统调用退出指令后, 则根据 VCPU 寄存器信息解析返回值. 实验证明, 与同类捕获系统调用的方法相比, 该系统可以实时捕获客户机内的系统调用序列, 解析得到完整的系统调用信息, 包括系统调用名、系统调用号、参数和返回值. 该系统还能区分不同进程产生的系统调用, 并在宿主机中引入了不超过 15% 的性能开销.

关键词: 系统调用序列; 系统调用参数; 系统调用返回值; KVM; 指令重写

引用格式: 宁强,崔超远,李勇钢.系统虚拟化环境下客户机系统调用信息捕获与分析.计算机系统应用,2019,28(3):73–79. http://www.c-s-a.org.cn/1003-3254/6792.html

Capture and Analysis of Guest System Calls' Information in System Virtualization Environment

NING Qiang[1,2], CUI Chao-Yuan[1], LI Yong-Gang[1,2]

[1](Institute of Intelligent Machines, Hefei Institutes of Physical Science, Chinese Academy of Sciences, Hefei 230031, China)
[2](University of Science and Technology of China, Hefei 230026, China)

Abstract: For the problem that current methods unable to capture and analyze the system call parameters and return values, a system for real-time monitoring of system calls in the guest was established based on Nitro. The system capture and analyze fast system call entry and exit instructions by modifying hardware specifications and rewriting instructions. After capturing the system call entry instruction, the parameters are parsed according to the context information of the VCPU and the semantic template of the system call; after the system call exit instruction is captured, the return value is parsed according to the VCPU register information. Compared with the similar capture system call method, experiments show that the system can capture the system call sequence in the guest in real time, and obtain complete system call information including system call name, system call number, parameters, and return value. The system can also distinguish between system calls generated by different processes and brings no more than 15% performance overhead to the host.

Key words: system call sequence; system call parameters; system call return value; KVM; instruction rewriting

系统调用被广泛应用于进程行为分析和恶意软件入侵检测[1]等方面. 系统调用信息的捕获和分析是开展这些工作的关键. Ether[2]是一个基于硬件虚拟化的客户机自省工具, 其用来进行恶意软件分析. Ether 利用

① 收稿时间: 2018-09-05; 修改时间: 2018-09-27; 采用时间: 2018-09-30; csa 在线出版时间: 2019-02-22

x86 快速系统调用条目机制的特性, 在系统调用执行期间, 其系统调用拦截机制使用 x86 处理器上的特殊寄存器, 在一个已有地址上产生页错误. Ether 通过接收该地址上产生页错误的信号判断一个系统调用的产生. Nitro[3]同样也是一个使用硬件虚拟化扩展的用于客户机自省的开源项目, 它在 KVM[4](Kernel Virtual Machine, 内核虚拟机) 平台上实现. KVM 由两个部分组成, 一个是基于 QEMU[5]的用户空间应用程序, 一个是提供实际虚拟化功能的 Linux 内核模块. Nitro 将这两部分进行了扩展, 实现了系统调用的跟踪和监控. 目前捕获系统调用的同类方法由于不能预知每个客户机操作系统的类型及其运行的应用, 无法制定统一的系统调用信息捕获和解析规则, 故仅能获取系统调用原始信息, 无法解析参数和返回值等重要信息. 此外, 现有方法没有考虑到捕获系统调用的性能开销问题, 没有对系统调用以进程为划分界限进行分类. 由于不同系统调用的参数类型不同, 参数解析面临很大的困难. 针对以上问题, 本文在 KVM 虚拟化平台上建立了一个实时捕获和分析客户机内系统调用的系统. 该系统根据不同系统调用建立相应的分析规则, 实现了对系统调用序列的捕获以及对参数和返回值的解析. 通过 exit() 系统调用判断出进程的终止操作, 并将进程从监视目标中移除, 从而区分系统调用序列所属的不同进程.

1 系统设计

1.1 设计目标

系统调用监视系统旨在利用虚拟化技术[6]在客户机外部实现对其内部系统调用序列进行实时捕获, 并对参数和返回值进行解析. 同时, 不对宿主机和客户机带来太大的性能开销. 因此设计目标包括以下三个方面:

(1) 实现对系统调用参数和返回值的快速捕获. 基于此点要求, 需要在处理系统调用引发异常时, 对系统调用进入和退出指令进行模拟; 然后在模拟过程中获取客户机 VCPU 信息, 从中提取参数和返回值的原始信息 (即二进制信息);

(2) 将低层次的系统调用信息还原成高层次语义[7]. 即在捕获到系统调用的二进制信息后, 使用内核自省工具 libvmi[8]将其解析为高层次语义信息;

(3) 引入较小的性能开销. 在确保快速捕获系统调用参数和返回值的前提下, 使用线程池技术快速获取并解析多个 VCPU 的二进制信息, 尽量减小对宿主机和客户机带来的性能开销.

1.2 系统总体架构

系统总体架构如图 1 所示, 主要由 KVM 内核功能扩展模块、监听模块、信息处理模块和参数解析模块四部分组成. 系统运行过程中, 监听模块将修改硬件规范使系统调用陷入的指令发送给 KVM 内核功能扩展模块, 然后 KVM 内核功能扩展模块将捕获到的系统调用低层次语义信息发送给监听模块. 信息处理模块将低层次语义信息转化为系统调用名和系统调用号这些高层次语义信息. 参数解析模块完成系统调用参数的解析.

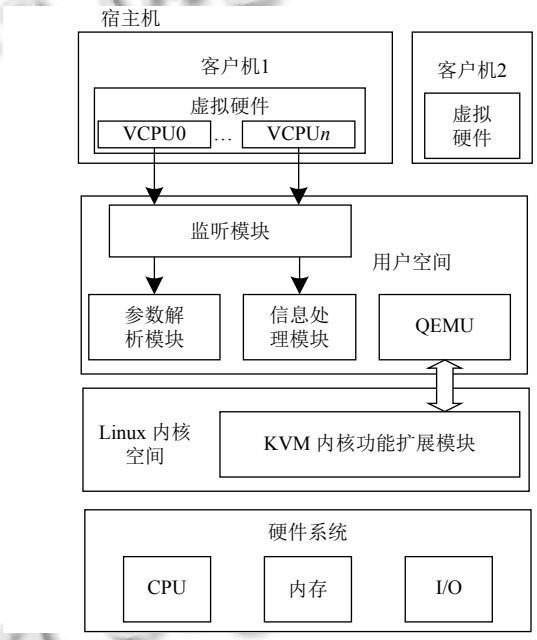

图 1 系统架构

实现该系统需要解决模拟指令的重写、语义的转换和进程的区分三个关键问题. 在由快速系统调用指令引起的异常发生时, VMM[9](Virtual Machine Monitor, 虚拟机监控器) 捕获该异常进行处理后对指令进行模拟. 模拟指令需要解决的首要问题就是对指令进行重写, 重写后的指令一方面要维持原有指令的功能, 另一方面要满足捕获系统调用的需求, 即得到 VCPU 寄存器信息. 解决模拟指令重写的方法是在 KVM 模拟指令的函数 (如 em_syscall()) 中加入 kvm_arch_vcpu_ioctl_get_regs() 函数和 kvm_arch_vcpu_ioctl_get_sregs() 函数来获取通用寄存器信息和特殊寄存器信息, 这些信息对应了系统调用的参数和返回值. 语义的转换包含两方面, 一个是解析系统调用名, 方法

是获取其在内核符号表中的虚拟地址, 然后使用 libvmi 的 translate_v2ksym() 进行解析, 另一个是解析系统调用参数, 针对字符串类型的参数, 需使用 libvmi 的 read_str_va() 进行解析. 捕获系统调用时需要区分已终止的进程和正在运行的进程, 其目的在于防止重复捕获同名同号新旧进程的系统调用造成新进程系统调用序列不再准确, 提高捕获效率, 解决进程区分问题的方法是识别进程是否调用了 exit() 或 exit_group() 函数.

2 系统实现

2.1 系统调用的捕获

当前基于 x86 架构的操作系统大多使用基于 syscall 指令和基于 sysenter 指令的快速系统调用. 针对这两种类型的系统调用需要采用不同的捕获方法.

客户机正常执行时 syscall 指令不会陷入到 VMM 中, 因此无法实现对基于 syscall 指令系统调用的捕获. 根据 Intel 软件开发手册上的硬件规范可知, 在将 EFER 寄存器的 SCE 标志位置 0 时执行 syscall 指令, 系统会发生 UD 未定义的无效操作码异常陷入, 这样就成功实现了对基于 syscall 指令系统调用的捕获.

基于 sysenter 指令的系统调用同样依赖于一组 MSR 特殊寄存器, 分别是 SYSENTER_CS_MSR, SYSENTER_ESP_MSR 和 SYSENTER_EIP_MSR. 为了使客户机系统在执行 sysenter 指令时能陷入到 VMM 中, 需要修改相关的硬件规范. 基于 sysenter 指令的系统调用入口地址保存在上述 MSR 寄存器中, 因此可以先将 MSR 寄存器原始值先保存起来, 然后装入一组 NULL 非法值, 系统在执行 sysenter 指令时由于找不到系统调用入口地址会产生 GP 通用保护错异常陷入, 这样就成功实现了对基于 sysenter 指令系统调用的捕获.

在客户机运行时, 用户模式下的监听模块首先将设置系统调用捕获的指令通过/dev/kvm 设备文件的 iotcl 调用发送给 KVM 内核模块, 当客户机内的程序执行系统调用时发生异常陷入到 VMM, KVM 内核模块捕获到系统调用并将信息返回给监听模块, 监听模块收到事件信息后停止监听并关闭/dev/kvm 设备文件的 ioctl 调用.

2.2 模拟指令的重写

对于 VM 中发生的异常, 必须要识别出该异常是由于客户机正常执行过程中引起的还是由于改变了系统调用指令正常执行条件引起的. 为此本文在内核处理异常的代码中加入了对客户机异常的判断, 其原理是判断发生异常时 VCPU 的状态, 即: 使用 is_invalid_opcode() 识别无效操作码异常, 若异常在客户模式下产生, 则是正常情况下产生的异常, 就使用 kvm_queue_exception() 将异常注入到 Guest OS 中处理; 否则, 使用 emulate_instruction() 对引起异常的系统调用指令进行模拟, 最后返回 Guest OS. 使用 is_general_protection() 识别通用保护异常, 若异常发生时 VCPU 处于 sysenter_sysexit 状态, 则模拟系统调用指令, 最后返回 Guest OS; 否则是正常情况下产生的异常, 将异常注入到 Guest OS 中处理.

异常识别和处理的方法如算法 1 所示. 第 1–6 行用于识别和处理无效操作码异常. 第 7–12 行用于识别和处理通用保护异常.

算法 1. 异常识别和处理算法

输入: 异常信息 intr_info, 客户机虚拟 CPU 信息 vcpu
输出: 顺利处理完异常, 返回 1

```
1.  IF(is_invalid_opcode(intr_info))
2.    IF(is_guest_mode(vcpu))
3.      kvm_queue_exception(vcpu);
4.      RETURN 1;
5.    ELSE emulate_instruction(vcpu);
6.      RETURN 1;
7.  ELSE IF(is_general_protection(intr_info))
8.    IF(is_sysenter_sysexit(vcpu))
9.      emulate_instruction(vcpu);
10.     RETURN 1;
11.   ELSE kvm_queue_exception(vcpu);
12.     RETURN 1;
```

模拟指令是处理异常中的关键环节, 对该指令的重写是获取系统调用信息的关键因素. 客户机在执行完若干条传递系统调用参数的传送指令后执行 sysenter 快速系统调用指令从用户态快速进入内核态, 进入内核态后执行 wrmsr 特权指令实现 MSRs 寄存器的初始化工作. 由于修改了硬件规范, 因此在执行 wrmsr 特权指令时客户机会产生异常陷入到 VMM 中, 为了恢复客户机系统调用的正常执行, VMM 需要对系统调用指令进行模拟. 对模拟指令进行重写有两个目的, 一个是重写模拟 sysenter 进入指令获取系统调用的参数, 另一个是重写模拟 sysexit 退出指令获取系统调用的返回值, 重写的方法是在内核代码 arch/x86/

kvm/emulate.c 下的 em_sysenter() 和 em_sysexit() 函数中添加 kvm_arch_vcpu_ioctl_get_regs() 函数和 kvm_arch_vcpu_ioctl_get_sregs() 函数.

2.3 语义的转换

语义的转换是将内核扩展模块捕获到的系统调用低层次语义信息转换为高层次语义信息, 信息处理模块完成对系统调用名和系统调用号的解析, 参数解析模块完成对系统调用参数的解析. 实现语义的转换必须要不断向 KVM 内核模块发送获取 VCPU 事件信息的请求, 在客户机中有多个 VCPU 的情况下, 如果只创建一个线程来监听多个 VCPU 的事件信息, 那么每捕获到一个 VCPU 上的系统调用就需要暂停一次客户机, 客户机暂停时间相应地增长了; 如果针对每个 VCPU 创建一个监听线程, 那么客户机暂停时间将缩短为多个监听线程中耗时最长的一个, 相比于单监听线程客户机性能得到了提升. 本文在监听模块中使用了线程池技术, 其原理是引入 concurrent.futures 模块, 该模块可实现并行计算, 即使用 ThreadPoolExecutor 类把监听多个 VCPU 事件信息的工作分配给多个 Python 线程处理, 进而实现系统调用二进制信息的快速获取和解析, 降低了客户机暂停时间, 提升了客户机的性能.

2.3.1 系统调用名和系统调用号解析

在内核中维护着一张符号表 System.map 被称作内核符号表, 该表记录了内核中所有符号 (函数, 全局变量等) 的地址及名称, 这其中就包括系统调用表和系统调用名的地址和名称. 系统调用表 sys_call_table 中各表项是指向实现各种系统调用的内核函数的函数指针, 系统调用执行时, 系统调用处理程序会读取 rax 寄存器来获取系统调用号, 将其乘以 4(32 位下为 4, 64 位下为 8) 生成偏移地址, 然后以 sys_call_table 为基址, 基址加上偏移地址可得系统调用表项地址, 使用 libvmi 对表项地址进行解析可得系统调用服务例程地址, 最后使用 libvmi 解析服务例程地址可得系统调用名.

解析系统调用名的方法如算法 2 所示. 第 1 行定义了 64 位下系统调用表项大小. 第 2 行根据系统调用号 rax 计算系统调用表项的地址. 第 3 行使用 libvmi 的 read_addr_va() 解析表项地址得到系统调用服务例程地址. 第 4 行使用 libvmi 的 translate_v2ksm() 解析系统调用服务例程地址得到系统调用名. 第 5 行返回系统调用名.

算法 2. 系统调用名解析算法

输入: 系统调用表地址 sys_call_table_addr, 系统调用号 rax
输出: 系统调用名 sys_call_name

```
1. #define VOID_P_SIEZ 8
2. p_addr = sys_call_table+rax*VOID_P_SIZE
3. addr = libvmi.read_addr_va(paddr)
4. sys_call_name = libvmi.translate_v2ksym(addr)
5. RETURN sys_call_name
```

2.3.2 系统调用参数解析

基于 syscall 指令的系统调用其参数存放在 rdi、rsi、rdx、r10、r8 和 r9 寄存器中, 而基于 sysenter 指令的系统调用其参数存放在 rbx、rcx、rdx、rsi、rdi、rbp 中. 不同的系统调用参数个数和类型也不同, 为了解决系统调用参数解析困难的问题, 设计了参数解析模块. 该模块根据不同类型的系统调用设置了相应的系统调用参数处理规则. 例如, 针对文件进行操作的 open 和 access 系统调用, 其第一个参数是所要操作文件的文件名, 对该参数的处理规则就是将存放参数的寄存器中的地址信息转换为文件名对应的字符串, 转换方法为使用 libvmi 的 read_str_va() 函数.

2.4 进程区分

现有的系统调用捕获方法在运行系统调用监控程序时, 没有考虑到进程终结的问题, 即不能判断已经退出的进程, 并将属于该进程的系统调用序列从整个系统调用的序列中剔除. 这样就会存在两个弊端: 第一, 存在与已退出旧进程同名同号的新进程, 由于监控程序无法判断旧进程的退出, 故会重新扫描旧进程的系统调用序列并加入新进程的系统调用序列, 造成新进程系统调用序列不再准确. 第二, 对于监控程序捕获到的多个进程的系统调用序列, 在上述情况下, 每个进程的系统调用序列都不再准确, 监控程序所捕获到的以一个进程系统调用序列为单位的总系统调用序列相应地也不再准确. 以上存在的弊端会严重降低系统调用序列捕获的准确率, 对宿主机的性能也会产生影响.

针对以上问题, 我们需要在捕获系统调用时及时判断进程是否退出, 如果进程退出, 就将属于该进程的系统调用序列从总的系统调用序列中移除. 在操作系统中, 进程退出一般会显式或隐式地调用 exit() 或 exit_group() 函数, 两者的区别在于前者只退出该进程, 而后者是退出属于该进程组的所有进程, 两者最后都会调用内核中的 do_exit() 函数 (位于 kernel/exit.c 函数

中). 在捕获系统调用时, 判断进程是否会调用此函数, 如果调用了, 说明进程已经退出, 将其从总的系统调用序列中移除并写到文件中.

3 实验与分析

3.1 实验环境

本实验是在 Linux 操作系统下进行的, 以 KVM 作为虚拟化平台, 设计并实现了系统调用监控系统, Qemu 版本为 2.8; 宿主机操作系统版本为 ubuntu-16.04-desktop-amd64,内核版本为 4.9, CPU 采用 24 核 Intel(R) Xeon(R) CPU E5-2620 v2 @ 2.10 GHz; 客户机操作系统版本为 ubuntu-12.04-server-amd64, 内核版本为 3.2.0-23-generic, VCPU 为单核, 内存为 1 GB.

3.2 实验结果与分析

3.2.1 系统调用捕获分析

从宿主机中捕获的客户机系统调用序列如图 2 所示 (左侧是客户机外部视图, 右侧是客户机内部视图), 通过交叉视图对比发现, 客户机内外系统调用序列完全一致, 说明在客户机外部可以捕获到客户机内完整的系统调用序列.

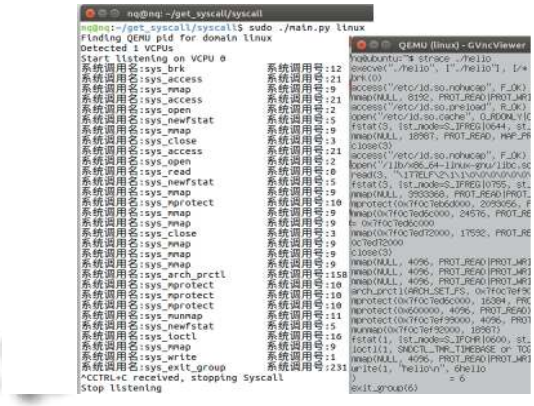

图 2 系统调用序列的捕获

由于解析的系统调用序列较长, 所以此处仅选择部分解析的系统调用序列进行分析, 客户机内外系统调用参数和返回值对比如图 3 所示.

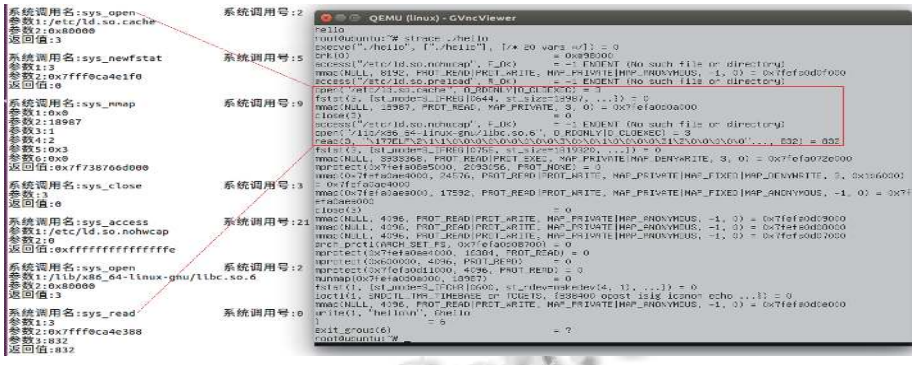

图 3 系统调用参数和返回值

Open 系统调用第一个参数为打开文件的路径名. 由内外视图对比可以发现, open 分别打开了/etc/ld.so.cache 和/lib/x86_64-linux-gnu/libc.so.6 文件, 这两个文件分别表示 Linux 动态共享库缓存文件和 Glibc 动态库文件. O_RDONLY 表示以只读方式打开文件, O_CLOEXEC 表示在进程执行 exec 系统调用时关闭此打开的文件描述符, 两者进行或操作后值为 0x80000. 客户机内外部得到的返回值都为 3. 对比发现, 客户机外部捕获到的 open 系统调用参数和返回值是正确的.

Access 系统调用是检查调用进程是否可以对指定的文件执行某种操作. 由图可见, 客户机进程调用了 access 来确定/etc/ld.so.nohwcap 文件是否存在, F_OK 在头文件 unistd.h 中的预定义为 0. 对比发现, 客户机外部捕获到的 access 系统调用参数和返回值是正确的.

Mmap 系统调用是将一个文件或者其它对象映射进内存. 其第 3 个参数 PROT_READ 和第 4 个参数 MAP_PRIVATE 在 sys/types.h 中分别定义为 1 和 2, 其余参数也一致. 但是得到的返回值有差异, 这是由于 strace 改变了原来系统调用的走向, 使得返回值发生了改变. 为了验证客户机外部得到的 mmap 返回值是否正确, 本文选择在测试程序运行过程中输出 mmap 的返回值, 如图 4 所示. 对比发现, 客户机外部捕获到的 mmap、open 和 write 系统调用参数和返回值是正确的.

由于本系统对模拟系统调用进入的指令进行重写,

捕获到了参数, 并对参数进行解析将低层次语义信息转换为高层次语义信息. 对模拟系统调用退出的指令进行重写, 捕获到了返回值. 以上实验数据充分说明本系统在捕获系统调用方面具有高效性和准确性的特点.

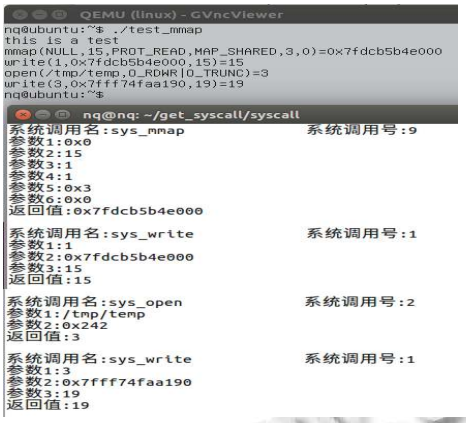

图 4 mmap、open 和 write 参数返回值

3.2.2 性能分析

本文实验使用了 Nbench 工具分别对宿主机和客户机的性能进行了分析. Nbench 是一个简单的用于测试处理器和存储器性能的基准测试工具, 它的测试结果主要分为 MEM、INT 和 FP, MEM 指数主要体现处理器总线、CACHE 和存储器性能, INT 指整数处理性能, FP 指双精度浮点性能. 测试所得数据能够反映宿主机和客户机在不同状态下的性能, 即原始状态的性能①、运行监控程序获取系统调用原始信息的性能②和对信息进行处理时的性能③, 宿主机性能比较如图 5 所示, 客户机性能比较如图 6 所示.

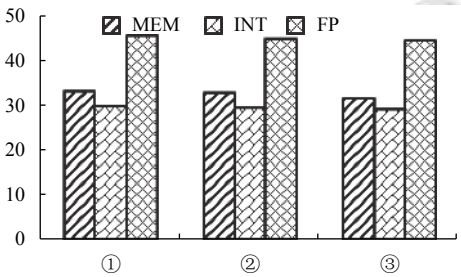

图 5 宿主机性能比较

根据图 5, 可以发现运行系统调用监控程序对宿主机并没有什么影响, 主要是由于宿主机只负责和客户机进行交互操作以及系统调用信息的处理等操作, 而这些操作主要涉及处理器的计算和内存的读写, 所以性能下降不是很明显.

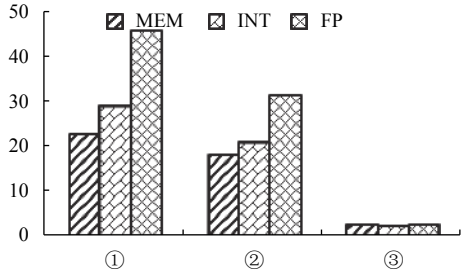

图 6 客户机性能比较

根据图 6, 可以发现监控程序在只捕获客户机系统调用原始信息的情况下, 即运行 Nitro 的情况下, 客户机的性能下降较小. 当监控程序需要对系统调用信息进行处理, 也就是需要解析系统调用参数和返回值、进程信息的时候, 客户机的性能下降较多, 主要是由于客户机在调用系统调用退出命令时陷入到 VMM 中, 陷入期间宿主机获取并解析系统调用参数, 客户机调用系统调用退出命令时再次陷入 VMM 中, 陷入期间宿主机获取系统调用返回值, 这两步造成客户机陷入时间过长. 针对 MEM、INT 和 FP 测试结果, 客户机原始性能指数分别为 22.7、29.1 和 45.7, 开启信息处理之后客户机的性能指数分别为 2.8、2.5 和 2.7, 性能开销比例分别为 8.1=22.7/2.8、11.64=29.1/2.5 和 16.9=45.7/2.7.

综上分析可知, 性能开销在可接受的范围内, 并且在只获取系统调用原始信息的情况下, 性能开销比例只有 1 到 1.5 左右, 性能要优于熊海泉等人提出的非陷入系统调用指令捕获方法[10], 其方法首先在 VMM 中通过监测 CR3 的更新来识别客户机内的当前进程, 随后通过修改硬件规范实现对基于 sysenter 和基于 syscall 指令系统调用的捕获, 最后基于 netlink 机制输出系统调用信息. VMM 在创建 netlink 连接向用户空间发送系统调用信息时客户机处于暂停状态, 额外增加了客户机的性能开销, 其方法对基于 sysenter 指令系统调用捕获的开销比例为 2.608=1238.8/475.0, 对基于 syscall 指令系统调用捕获的开销比例为 2.195=1213.3/552.8. 本监控系统性能优于其他方法的原因是在实现仿真指令的重写过程中即获取到了系统调用信息并实时传递给用户空间, 同时考虑到客户机存在多 VCPU 的情况, 引入了线程池技术实现系统调用信息的快速捕获和解析, 大大提升了客户机的性能.

4 结束语

为了解决捕获客户机系统调用时无法解析系统调用参数和返回值的问题, 本文在 KVM 虚拟化平台上设计并实现了一个系统调用监控系统. 系统采用了模块化的设计, 内核功能扩展模块和用户空间模块相互配合. 该系统不仅可以实时捕获客户机内的系统调用序列, 同时还可以解析系统调用的参数和返回值, 将系统调用的原始信息转换为高层次语义信息.

目前该系统在解析系统调用原始信息时对客户机带来的性能开销较大, 对用户的使用有些影响. 此外, 系统未涉及针对基于系统调用的恶意软件的分析, 这些将是我们未来的研究工作.

参考文献

1 吴瀛, 江建慧, 张蕊. 基于系统调用的入侵检测研究进展. 计算机科学, 2011, 38(1): 20–25, 47. [doi: 10.3969/j.issn.1002-137X.2011.01.004]

2 Dinaburg A, Royal P, Sharif M, *et al*. Ether: Malware analysis via hardware virtualization extensions. Proceedings of the 15th ACM Conference on Computer and Communications Security. Alexandria, VA, USA. 2008. 51–62.

3 Pfoh J, Schneider C, Eckert C. Nitro: Hardware-based system call tracing for virtual machines. Proceedings of the 6th International Workshop Advances in Information and Computer Security. Tokyo, Japan. 2011. 96–112.

4 Kivity A, Kamay Y, Laor D, *et al*. KVM: The Linux virtual machine monitor. Proceedings of Ottawa Linux Symposium. Ottawa. 2007. 1–8.

5 Bartholomew D. QEMU: A multihost, multitarget emulator. Linux Journal, 2006, 145: 3.

6 广小明, 胡杰, 陈龙, 等. 虚拟化技术原理与实现. 北京: 电子工业出版社, 2012. 43–45.

7 李勇钢, 崔超远, 李平. 基于虚拟机自省的客户机进程内容获取. 计算机工程与设计, 2016, 37(6): 1697–1701.

8 Xiong HQ, Liu ZY, Xu WZ, *et al*. Libvmi: A library for bridging the semantic gap between guest OS and VMM. Proceedings of the 2012 IEEE 12th International Conference on Computer and information Technology. Chengdu, China. 2012. 549–556.

9 Agesen O, Garthwaite A, Sheldon J, *et al*. The evolution of an x86 virtual machine monitor. ACM SIGOPS Operating Systems Review, 2010, 44(4): 3–18. [doi: 10.1145/1899928]

10 熊海泉, 刘志勇, 徐卫志, 等. VMM 中 Guest OS 非陷入系统调用指令截获与识别. 计算机研究与发展, 2014, 51(10): 2348–2359. [doi: 10.7544/issn1000-1239.2014.20130612]