The Architectural Based Interception and Identification of System Call Instruction within VMM

Haiquan Xiong[1,2]
[1]State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences
[2]University of Chinese Academy of Sciences
Beijing, China
xionghaiquan@ict.ac.cn

Zhiyong Liu[1]
[1]State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences
Beijing, China
zyliu@ict.ac.cn

***Abstract*—To solve the problem that VMM cannot monitor and control Guest OS system call instructions due to their non-trapping property, this paper propose an idea that make these instructions trap into VMM through breaking their normal execution conditions so as to cause exception. As to the three different system call mechanisms in the x86 architecture, we make software interrupt and sysenter based system calls trap into VMM through causing GP exception trap, while syscall trap into VMM through causing UD exception trap, and then identify them with the vcpu context information corresponding to the exception trap. The Qemu&Kvm based prototype indicates that VMM can successfully intercept and identify all the three system call behaviors coming 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 ranges from 1.900 to 2.608.**

Keywords- Guest OS;VMM;Virtualization; System Call

I. INTRODUCTION

Virtualization technology has been widely used in the IT security field[1], most often, they need effective capabilities to monitor Guest OS behaviors and its state within VMM[6-9]. Generally, when a sensitive instruction is executing in Guest OS, it will be trapped into VMM, then with the knowledge of hardware architecture and Guest OS software convention, VMM can distill more useful information from these events[2,3,10]. However, there always exist some instructions related to specific behaviors that cannot directly trap into VMM because of their non-sensitive property[4]. For example, the x86 system call instruction is just a case in point[2]. As a result, it is difficult to monitor these behaviors within VMM.

In order to solve the above problems, the basic idea of current research is to replace the non-trapping instruction with the sensitive trapping instruction [5]. For example, the Guest OS code injection method requires a sensitive trappable code segment be injected into the specific location of Guest OS; Double instruction swapping method needs two traps - two VMCALLs were performed and the guest OS's default handler was modified frequently. The common requirement of these two methods is that they both need modifying Guest OS; therefore the portability is pool and also is not transparent to Guest OS.

In this paper, we explored how to intercept and identify the three different non-trapping system call instructions of x86 architecture from Guest OS within VMM without any modifications to Guest OS, making software interrupt and sysenter based system calls trap into VMM through causing GP exception, while syscall instruction trap into VMM through causing UD exception, and then identify them with the VCPU context information corresponding to the exception trap.

The evaluation of the Qemu&Kvm based prototype demonstrates that VMM can successfully intercept and identify all the three system call mechanisms, and in normal condition, the performance overhead is within the acceptable range. For example, in unixbench shell test case, the performance overhead ratio ranges from 1.900 to 2.608. This provides a new method for VMM to monitor the Guest OS process level behavior.

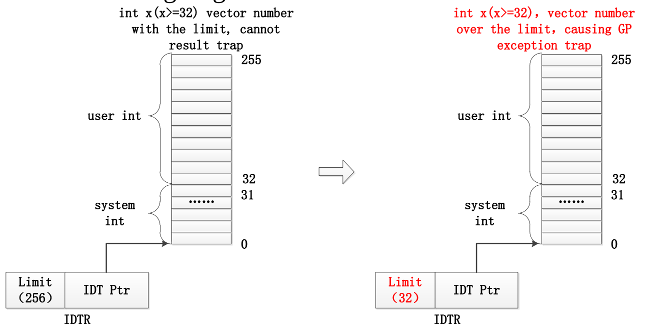
II. THE INTERCEPTION AND IDENTIFICATION OF SYSTEM CALL INSTRCTION WITHIN VMM

In the evolution of x86 architecture, there have been three kinds of system call mechanisms: software interrupt based, sysenter instruction based and syscall instruction based methods. Generally, without special treatment, these system calls cannot be automatically trapped into VMM.

A. Mechanism 1: Software Interrupt Based Method

In the x86 architecture, interrupt handling is through the interrupt description table (IDT). When an interrupt occurs, the hardware will find the handler from IDT indexed by the vector number. In virtualized environment, Intel VT-x only allow the system interrupts (vector number from 0 to 31) to be automatically trapped into VMM, and the user interrupts (vector number from 32 to 255) that can be used to implement system call cannot be trapped directly.

To make the software interrupt based mechanism trap into VMM, the solution relies on setting the limit field. When the limit is set to 32, all system interrupts are not influenced, but all the user interrupts (>31) will lead to GP exception. Thereafter, VMM needs to discern this kind of GP exception from the normal one. If belonging to the normal case, it will be forwarded to Guest OS directly; otherwise, it will be a user interrupt and need to collect more information for further handling. Figure 1 demonstrates the mechanism.

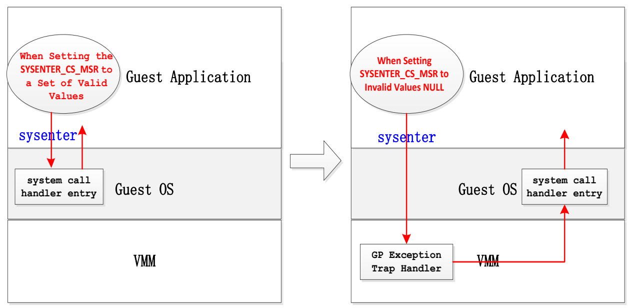


int x(x>=32) vector number with the limit, cannot result trap
255
user int
32
31
system int
......
0
Limit (256)
IDT Ptr
IDTR
int x(x>=32), vector number over the limit, causing GP exception trap
255
user int
32
31
system int
......
0
Limit (32)
IDT Ptr
IDTR

Figure 1 Software Interrupt Based System Call Interception

B. *Mechanism 2: Sysenter Instruction Based Method*

Sysenter instruction based system call mechanism depends on a set of MSRs，that is SYSENTER_CS_MSR, SYSENTER_ESP_MSR and SYSENTER_EIP_MSR, they all together represent the entry address information of system calls. When the application in the Guest OS executes sysenter instruction, the system will load the values stored in these MSRs which represent the system call entry in Guest OS. So it needs no intervention from VMM and VMM cannot realize the required controls directly. To make sysenter instruction trap, it should create a contrived environment. The solution is to write a set of invalid values (such as null) to MSRs, and then it will cause GP exception. Therefore, after saving the old MSRs values and loading a set of null values into these MSRs will cause the sysenter trap into VMM, then VMM will discern the contrived case from the normal case with vcpu context information. If it is a normal GP exception, VMM will direct it to the Guest OS, otherwise it will be regarded as sysenter system call, so it needs collecting further information to process and use the previous saved MSRs to return to Guest OS. Figure 2 illustrates this idea.



When Setting the SYSENTER_CS_MSR to a Set of Valid Values
Guest Application
sysenter
system call handler entry
Guest OS
VMM
When Setting SYSENTER_CS_MSR to Invalid Values NULL
Guest Application
sysenter
Guest OS
system call handler entry
GP Exception Trap Handler
VMM

Figure 2 Sysenter based Interception and Identification

C. *Mechanism 3: Syscall Instruction Based*

Syscall instruction based system call mechanism also depends a set of MSRs， that is STAR_MSR, CSTAR_MSR and LSTAR_MSR, which registers are used depends on what mode of Guest OS is running in. Normally syscall cannot directly trap into VMM, so it is difficulty for VMM to implement controls. The solution here is to set the SCE flag value in EFER register. When it is set to 0, the syscall will UD exception trap, while set to 1 it will not. When detecting an UD exception trap in VMM, with the current vcpu context information, the VMM can identify the syscall instruction. If in normal case, the UD exception will direct to Guest OS, otherwise, it will collect related data and simulate the required effects, then return to Guest OS.

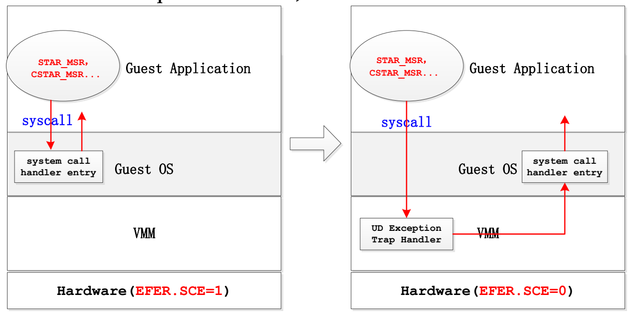


Figure 3 syscall based instruction system call interception and identification

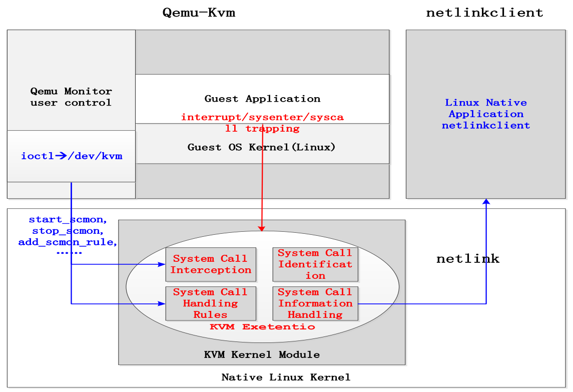
III. The Prototype System

To verify and evaluate the purposed methods, we have implemented a prototype on Qemu&Kvm VMM. Figure 4 illustrates the prototype system architecture. It includes three main modules: Qemu Monitor user control module, KVM extension module and user mode application netlinkclient.

The Qemu Monitor user control module provides the user with an interface to issue commands to the KVM extension module. These commands include enabling and disabling system call interception mechanisms, adding or deleting system call processing rules and etc.

The KVM extension module implements the system call interception, identification and processing functionality in response to commands from the Qemu Monitor user control module. Besides, it also includes an output service for displaying system call processing results through a netlink, which is then read by the user mode application netlinkclient.

The netlink-based user mode application netlinkclient is responsible for reading the information produced by the KVM extension output service.



Qemu-Kvm
netlinkclient
Qemu Monitor user control
Guest Application
interrupt/sysenter/syscall trapping
Guest OS Kernel(Linux)
ioctl→/dev/kvm
Linux Native Application netlinkclient
start_scmon, stop_scmon, add_scmon_rule, ...
System Call Interception
System Call Identification
System Call Handling Rules
System Call Information Handling
KVM Exetentio
netlink
KVM Kernel Module
Native Linux Kernel

Figure 4 The Prototype System Architecture

IV. EVALUATION

We evaluated the functional correctness and performance overheads of the three system call mechanisms. All experiments are done on a platform with Intel i7 930 and 12 GB memory. The host and guest operating system for software interrupt and sysenter based methods adopt the 32 bit Fedora 13, while the syscall selects 64 bit Fedora 13. VMM chosen are Qemu-Kvm 0.13 and KVM 2.6.38-rc7.

A. Functional Verification

Table 1 gives different Qemu&Kvm virtual machine startup methods for testing the three system call mechanisms. Among them, the sysenter method needs adding –cpu qemu64, model=3 option parameter so as to make Guest OS think that it is running on a hardware which choose sysenter syscall mechanism.

System Call Mode	Qemu&Kvm Virtual Machine Startup Methods
Interrupt Based	#qemu –hda fedora13-x86-32.img –m 1024 –monitor stdio
Sysenter Based	#qemu –cpu qemu64,model=3 –hda fedora13-x86-32.img –m 1024 –monitor stdio
Syscall Based	#qemu-system-x86_64 fedora13-x86-64.img –m 1024 –monitor stdio

Table 1 Three System Call Start Methods

For simplicity, here we only select the software interrupt based type as an example to illustrate the basic operation steps and demonstrate the functional correctness of the above discussed methods:

1， In the Linux shell command line, input the following command, startup the Qemu&Kvm virtual machine. In this way, the Guest OS will automatically select the software interrupt based mechanism.

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

2， After the virtual machine startup, switch to the Qemu Monitor and input the command add_scmon_rule to add the system call handling rule, for example:

```
(qemu)add_scmon_rule rax 3 any 0 hex
```

When rax is set to 3, it represents the read system call in Linux; any 0 refers to output all general register s values, and hex means output format in hex style.

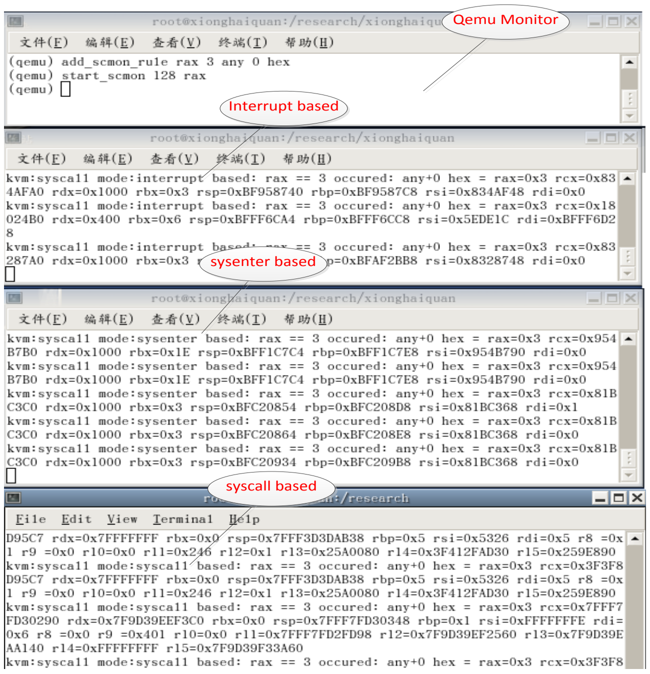
3， After step 2, the next is to enable the mechanism with command start_scmon, 128 is the vector number used by system call in Linux:

```
(qemu)start_scmon 128 rax
```

4， Finally, VMM can intercept the system call behaviors occurred in Guest OS and process them according to the rules set in step 2 and continue sending the results to the netlink from which the user mode application netlinkclient can read and output to the user.

```
#netlinkclient
```

Besides the differences in Step 1, Sysenter instruction and Syscall instruction based methods have similar steps. Figure 5 shows the snapshot of all the three mechanisms; It includes the Qemu Monitor command window and the outputs of the three mechanisms from netlinkclient. The outputs are: “syscall mode: interrupt based”, “syscall mode: sysenter based” and “syscall mode: syscall based”. The results demonstrate that all the three mechanisms can be successfully intercepted and identified within VMM, the methods described in Section III have been verified.


Qemu Monitor
Interrupt based
sysenter based
syscall based
root@xionghaiquan:/research/xionghaiquan
文件(F) 編輯(E) 查看(V) 終端(T) 帮助(H)
File Edit View Terminal Help

```
(qemu) add_scmon_rule rax 3 any 0 hex
(qemu) start_scmon 128 rax
(qemu)
```

```
kvm:syscall mode:interrupt based: rax == 3 occured: any+0 hex = rax=0x3 rcx=0x83
4AFA0 rdx=0x1000 rbx=0x3 rsp=0xBF958740 rbp=0xBF9587C8 rsi=0x834AF48 rdi=0x0
kvm:syscall mode:interrupt based: rax == 3 occured: any+0 hex = rax=0x3 rcx=0x18
024B0 rdx=0x400 rbx=0x6 rsp=0xBFFF6CA4 rbp=0xBFFF6CC8 rsi=0x5EDE1C rdi=0xBFFF6D2
8
kvm:syscall mode:interrupt based: rax == 3 occured: any+0 hex = rax=0x3 rcx=0x83
287A0 rdx=0x1000 rbx=0x3 r                  p=0xBFAF2BB8 rsi=0x8328748 rdi=0x0
```

```
kvm:syscall mode:sysenter based: rax == 3 occured: any+0 hex = rax=0x3 rcx=0x954
B7B0 rdx=0x1000 rbx=0x1E rsp=0xBFF1C7C4 rbp=0xBFF1C7E8 rsi=0x954B790 rdi=0x0
kvm:syscall mode:sysenter based: rax == 3 occured: any+0 hex = rax=0x3 rcx=0x954
B7B0 rdx=0x1000 rbx=0x1E rsp=0xBFF1C7C4 rbp=0xBFF1C7E8 rsi=0x954B790 rdi=0x0
kvm:syscall mode:sysenter based: rax == 3 occured: any+0 hex = rax=0x3 rcx=0x81B
C3C0 rdx=0x1000 rbx=0x3 rsp=0xBFC20854 rbp=0xBFC208D8 rsi=0x81BC368 rdi=0x1
kvm:syscall mode:sysenter based: rax == 3 occured: any+0 hex = rax=0x3 rcx=0x81B
C3C0 rdx=0x1000 rbx=0x3 rsp=0xBFC20864 rbp=0xBFC208E8 rsi=0x81BC368 rdi=0x0
kvm:syscall mode:sysenter based: rax == 3 occured: any+0 hex = rax=0x3 rcx=0x81B
C3C0 rdx=0x1000 rbx=0x3 rsp=0xBFC20934 rbp=0xBFC209B8 rsi=0x81BC368 rdi=0x0
```

```
D95C7 rdx=0x7FFFFFFF rbx=0x0 rsp=0x7FFF3D3DAB38 rbp=0x5 rsi=0x5326 rdi=0x5 r8 =0x
1 r9 =0x0 r10=0x0 r11=0x246 r12=0x1 r13=0x25A0080 r14=0x3F412FAD30 r15=0x259E890
kvm:syscall mode:syscall based: rax == 3 occured: any+0 hex = rax=0x3 rcx=0x3F3F8
D95C7 rdx=0x7FFFFFFF rbx=0x0 rsp=0x7FFF3D3DAB38 rbp=0x5 rsi=0x5326 rdi=0x5 r8 =0x
1 r9 =0x0 r10=0x0 r11=0x246 r12=0x1 r13=0x25A0080 r14=0x3F412FAD30 r15=0x259E890
kvm:syscall mode:syscall based: rax == 3 occured: any+0 hex = rax=0x3 rcx=0x7FFF7
FD30290 rdx=0x7F9D39EEF3C0 rbx=0x0 rsp=0x7FFF7FD30348 rbp=0x1 rsi=0xFFFFFFFE rdi=
0x6 r8 =0x0 r9 =0x401 r10=0x0 r11=0x7FFF7FD2FD98 r12=0x7F9D39EF2560 r13=0x7F9D39E
AA140 r14=0xFFFFFFFF r15=0x7F9D39F33A60
kvm:syscall mode:syscall based: rax == 3 occured: any+0 hex = rax=0x3 rcx=0x3F3F8
```

Figure 5 The Running Snapshots of three system call interception and identification

B. Performance Overhead

As to the performance overhead evaluation, we select three test cases from unixbench: syscall, arithmetic and shell. When the test case syscall is executing, it is just a system call loop and belongs to an extreme case; arithmetic is another extreme case, because it mainly does computing task with almost no system call behaviors, so it can be seen as the opposite of syscall test case; The last case is shell which sits in between the above two cases and can be considered as the normal application case.

The following experimental results are all collected from the test cases running in Guest OSes. Every table includes three different performance overhead ratio which refers to the ratio of performance before and after enabling system call interception. For example in table 2, the syscall test case is 538 before enabling and 38.9 after enabling, so the performance overhead ratio is 13.830=538/38.9.

Guest OS	Disable Interception	Disable Interception	Performance Overhead Ratio
Interrupt Based	538	38.9	13.830
Sysenter Based	1245.3	12.4	100.427
Syscall Based	1229.9	20.3	60.586

Table 2 syscall

Guest OS	Disable Interception	Disable Interception	Performance Overhead Ratio
Interrupt Based	1203.3	633.3	1.900
Sysenter Based	1238.8	475.0	2.608
Syscall Based	1213.3	552.8	2.195

Table 3 shell

Guest OS	Disable Interception	Disable Interception	Performance Overhead Ratio
Interrupt Based	545.2	545.3	0.9998
Sysenter Based	548.1	548.0	1.0002
Syscall Based	635.2	634.2	1.0016

Table 4 arithmetic

The performance overhead ratio results of the syscall test case in Table 2 for software interrupt、sysenter and syscall are 18.830 、 100.427 and 60.586 separately. This demonstrates that the system call interception within VMM will bring a huge overhead. However, it is an extreme case because it is just executing a system call loop. Besides, sysenter has the biggest overhead, the main reason is that it not only adds the interception overhead, but also adds extra simulation overhead. Table 3 can be regarded as the normal and real application case; the results are 1.900, 2.608, 2.195. It suggests that in practical applications, the system call interception doesn't bring performance degradation in magnitude because there are not so much proportional system call instructions compared to all other instructions. Actually, it is within an accepted range; Table 4 is another extreme case, almost no system calls done when executing, this time, the performance overhead ratio is nearly 1, it is in line with the expectations.

V. Conclusions

In virtualized computer system, the key mechanism of getting Guest OS internal events is by the automatic trapping property of sensitive instruction executing in Guest OS. However, there always exist some instructions that cannot trap into VMM directly because of their non-sensitive property. As a result, it is difficult to monitor these behaviors within VMM. To handle this problem effectively, based on the convention of hardware and software, the paper proposes an idea that makes the non-trapping instructions trap through breaking their normal execution conditions so as to make it cause exception trap. Compared to the existing methods, it has the advantages of Guest OS transparent and doesn't need any modifications to Guest OS. Based on this idea, the paper specially explored how to intercept and identify the three different system call instructions of x86 architecture comming from Guest OS within VMM. The evaluation of the Qemu&Kvm based prototype demonstrates that VMM can successfully intercept and identify all the three system call mechanisms, and in normal cases, the performance overhead is within an acceptable range. Although the paper only exemplifies the application with system call instructions, it also applies to other similar environments. Besides, the prototype implements only a framework for system call interception and identification facility, in the future, we plan to extend it with practical application. As such, it would be feasible to monitor the Guest OS process level information outside virtual machines.

Acknowledgment

This work is in part supported by the National Grand Fundamental Research 973 Program of China under Grant No. 2011CB302501, National High-Tech Research & Development Program of China No. 2012AA010303, the National Science Foundation for Distinguished Young Scholars of China under Grant No. 60925009, the National Natural Science Foundation of China under Grant No.(61173007，61100013 ， 61100015 and 61202059), the Key Lab of Computer Architecture Open Projects CARCH201203 and Huawei Cooperation Projects YBCB2011030.

References

[1] Rosenblum., T.G.a.M., A virtual machine introspection based architecture for intrusion detection. In Proc. Network and Distributed Systems Security Symposium, February 2003

[2] Pfoh, J., C. Schneider, and C. Eckert, Exploiting the x86 Architecture to Derive Virtual Machine State Information, in Proceedings of the 2010 Fourth International Conference on Emerging Security Information, Systems and Technologies. 2010, IEEE Computer Society. p. 166-175.

[3] Pfoh, J., C. Schneider, and C. Eckert, A formal model for virtual machine introspection, in Proceedings of the 1st ACM workshop on Virtual machine security. 2009, ACM: Chicago, Illinois, USA. p. 1-10.

[4] Popek, G.J. and R.P. Goldberg, Formal requirements for virtualizable third generation architectures. SIGOPS Oper. Syst. Rev., 1973. **7**(4): p. 121.

[5] Prosnitz, B., Blackbox No More:Reconstruction of Internal Virtual Machine State. 2007.

[6] Onoue, K., Y. Oyama, and A. Yonezawa, Control of system calls from outside of virtual machines, in Proceedings of the 2008 ACM symposium on Applied computing. 2008, ACM: Fortaleza, Ceara, Brazil. p. 2116-1221.

[7] Forrest, S., S. Hofmeyr, and A. Somayaji. The Evolution of System-Call Monitoring. in Computer Security Applications Conference, 2008. ACSAC 2008. Annual. 2008.

[8] Bo, L., et al. A VMM-Based System Call Interposition Framework for Program Monitoring. in Parallel and Distributed Systems (ICPADS), 2010 IEEE 16th International Conference on. 2010.

[9] Jiang, X., X. Wang, and D. Xu, Stealthy malware detection and monitoring through VMM-based "out-of-the-box" semantic view reconstruction. ACM Trans. Inf. Syst. Secur., 2010. **13**(2): p. 1-28.

[10] Jones, S.T., A.C. Arpaci-Dusseau, and R.H. Arpaci-Dusseau, Antfarm: tracking processes in a virtual machine environment, in Proceedings of the annual conference on USENIX '06 Annual Technical Conference. 2006, USENIX Association: Boston, MA. p. 1-1..