Control of System Calls from Outside of Virtual Machines

Koichi Onoue
The University of Tokyo
7-3-1, Hongo, Bunkyo-ku,
Tokyo, Japan
koichi@yl.is.s.u-tokyo.ac.jp

Yoshihiro Oyama
The University of Electro-Communications
1-5-1, Chofugaoka, Chofu-shi,
Tokyo, Japan
oyama@cs.uec.ac.jp

Akinori Yonezawa
The University of Tokyo
7-3-1, Hongo, Bunkyo-ku,
Tokyo, Japan
yonezawa@yl.is.s.u-tokyo.ac.jp

ABSTRACT

A virtual machine monitor (VMM) can isolate virtual machines (VMs) for trusted programs from VMs for untrusted ones. The security of VMs for untrusted programs can be enhanced by monitoring and controlling the behavior of the VMs with security systems running in a VM for trusted programs. However, programs running outside of a monitored VM usually obtain only low-level events and states such as interrupts and register values. Therefore, it is not straightforward for the programs to understand the high-level behavior of an operating system in a monitored VM and to control resources managed by the operating system. In this paper, we propose a security system that controls the execution of processes from the outside of VMs. It consists of a modified VMM and a program running in a trusted VM. The system intercepts system calls invoked in a monitored VM and controls the execution according to a security policy. To fill the semantic gap between low-level events and high-level behavior, the system uses knowledge of the structure of a given operating system kernel. The user creates the knowledge with a tool when building an operating system. We implemented the system using Xen, and measured the overhead through experiments using microbenchmarks and a benchmark for the Apache web server.

Categories and Subject Descriptors

D.4 [**Operating Systems**]: Security and Protection

General Terms

Security

Keywords

Virtual Machine Monitors, Security, Sandboxing, System Call Interception

1. INTRODUCTION

SAC'08 March 16-20, 2008, Fortaleza, Ceará, Brazil

Virtual machine monitors (VMMs) [1, 2, 4] provide strong isolation between virtual machines (VMs). If attackers succeed in taking over an application program or an operating system kernel running in a VM, it does not get easier for them to take over the VMM or another VM.

Unfortunately, simply isolating execution environments by VMMs does not provide sufficient security. Indeed, a VMM can confine damages by attackers to VMs and/or applications that are taken over. However, a VMM cannot prevent exploited program parts from being used for a malicious purpose. For example, if a web server hosted in a VM is taken over, secrets kept in the server are revealed to the attacker. Moreover, the attacker can modify or delete the information stored in the operating system by abusing the privilege of the server. An effective countermeasure to address these problems is to combine the VM isolation scheme with security systems.

There are two types of schemes that combine VMMs and security systems. One is to execute an existing security system (e.g., malware detection systems and sandboxes) in an operating system running in a VM (guest OS). This scheme has several advantages. First, it requires no or little modification to the code of a VMM or a guest OS. Second, OS-level information, such as processes and files, is easily available to programs running in a guest OS. However, this scheme has a disadvantage. Some attacks are conscious of security systems. If the attack evades detection by security systems and successfully obtains the administrator privilege of the guest OS, the attack can stop the security systems.

The other type of scheme is to execute a security system outside a monitored guest OS. This solves the problem of attacks conscious of security systems. Even if an attacker obtains the administrator privilege of the guest OS, the attacker cannot stop the VMM or the security system running outside. On the other hand, this scheme has a limitation that security systems obtain only information on low-level events for interactions between the guest OS and virtual hardware. It is not straightforward for the security systems to understand and control the high-level behavior of the guest OS. If information on the high-level behavior is recovered from the information on low-level events, this limitation is overcome.

In this paper, we propose a security system that controls the behavior of processes in a guest OS based on the information on low-level events observed by a VMM. A VMM in the system intercepts system calls invoked in a guest OS, and controls the execution of the system calls. A security policy given by the users specifies which processes in which VMs

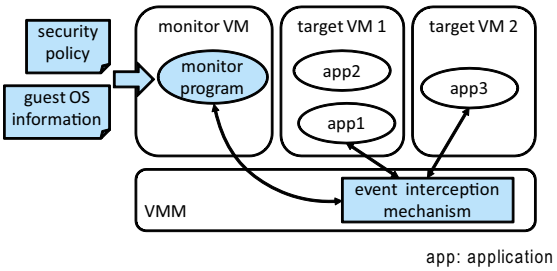

Figure 1: Structure of the proposed system.

are put under control, which system calls are controlled, and how they are controlled. To fill the semantic gap between low-level VM events and the high-level behavior of a guest OS, the system uses knowledge on the structure of a given operating system kernel. The knowledge is automatically created with a supportive tool when the image of an operating system is built. The knowledge includes the structures and symbols of critical data managed by the OS kernel. We implemented the proposed system using Xen [4] as a VMM and Linux as a guest OS. We conducted preliminary experiments using the system and measured the overhead.

This paper proceeds as follows. Section 2 gives an overview of the proposed system. In Section 3, we explain implementation details of the system. Section 4 shows experimental results. Section 5 describes related work, and Section 6 concludes this paper.

2. PROPOSED SYSTEM

2.1 Basic Structure

The structure of the proposed system is shown in Figure 1. The system runs a program that controls processes in a special VM. In this paper, we call the program a *monitor program* and we call the VM a *monitor VM*. A process whose behavior is controlled is called a *target process*. A VM that contains a target process is called a *target VM*. A monitor VM and a VMM in the system cooperates to control execution of system calls invoked by target processes. The VMM intercepts the entry and exit of system calls and sends the system call information to the monitor VM. According to a given security policy, the monitor VM makes a decision on a response action to take. The VMM executes the action. A monitor VM and a VMM can simultaneously control multiple target VMs running on the same VMM.

The following information on a guest OS kernel is essential in the proposed system. The first essential information is the locations of critical data in a data structure for process management (e.g., location of a process ID variable). On Linux, process information is kept in `task_struct` structures in the kernel space. A VMM must know the memory layout of the structure to read critical members in the structure. The second is the addresses of several routines in the kernel space corresponding to system call entry and exit. The routines are choke points through which executions of all system calls pass. The proposed system achieves system call interception by modifying an instruction in the routine. The last is the information on system calls. It consists of the relation between the number and the name of system calls, and a calling convention of system calls (e.g., usage of registers and stacks for passing system call arguments).

```
# vps 3 proclist-org.txt
  (choose processes from proclist-org.txt and
   writes them in proclist-cntl.txt)
# vconf 3 guest-os-info.txt syscall-info.txt
# vcntl 3 proclist-cntl.txt policy.txt
```

Figure 2: Usage example.

The information above depends on the version of an OS kernel and a configuration given to build an OS image. Hence, when the image of a guest OS is built, the user uses a supportive tool to automatically generate information that depends on the kernel image. Though the proposed system currently supports Linux only as a guest OS, it can support other UNIX-like OSes if the information on the OSes described above is available.

The system provides the following commands to the users. All commands must be executed in a monitor VM with the administrator privilege.

- `vps`: This command receives the ID of a target VM and a file path. It stores, in a given file, the information on all processes running in the taget VM. The information includes process IDs, user IDs, and command names. Intuitively, this command is regarded as a remote version of the `ps` command in UNIX systems.

- `vconf`: This command gives the system information about the guest OS. This command receives the ID of a target VM and two files. One file stores the layout of critical data structures of the guest OS kernel. The other stores the information on system calls.

- `vcntl`: This command starts the execution control of target processes. It receives the ID of a target VM, a file specifying target processes, and a security policy. Target processes are specified with process IDs or program paths.

A sample session of the system is shown in Figure 2. In this session, the user attempts to control processes in a target VM whose ID is 3. In the first line, `vps` outputs a list of processes in the VM to file `proclist-org.txt`. The user then chooses a list of processes that are put under control, and stores the information for specifying the processes in file `proclist-cntl.txt`. The user next tells the system the structure (`guest-os-info.txt`) and information on system calls (`syscall-info.txt`) of the guest OS. In the last line, the user starts controlling processes specified in file `proclist-cntl.txt` with a security policy written in file `policy.txt`.

2.2 Security Policy

The syntax and an example of security policy are shown in Figure 3 and Figure 4, respectively. Security policies specify controlled system calls with pattern matching of system call arguments. The `default:` field at the top level indicates an action taken for system calls that do not match with any pattern. The `default:` field written in the specification for each system call indicates a default response action for the system call. The `traceChild:` field indicates

PolicyFile	→	`default:` Action `traceChild:` YesNo SyscallSpec*
YesNo	→	`yes` | `no`
SysCallSpec	→	SysCallName `default:` Action ControlExpr*
ControlExpr	→	Condition* Action
Condition	→	FileCond | NetworkCond | Condition `and` Condition | Condition `or` Condition
FileCond	→	`fileEq`(*argnum,path*) | `filePrefix`(*argnum,pathPrefix*)
NetworkCond	→	`ip`(*ipaddr*) | `port`(*portnum*) | `protocol`(Protocol)
Protocol	→	`tcp` | `udp`
Action	→	`allow` | `deny`(*retnum*) | `killProc` | `policyChange`(*policyfile*)

Figure 3: Syntax of security policy (extracted).

```
 default:  deny(-1)
traceChild:  yes
open
   default:  allow
   fileEq(1,''/etc/passwd'')
      or filePrefix(1,''/etc/cron.d'')
      deny(-1)
execve
   default:  killProc
   fileEq(1,''/usr/bin/wserver'')
      policyChange(``wserver.pol'')
```

Figure 4: Sample security policy.

whether a child process is also controlled. Elements in "Action" indicate actions taken when a system call matches with a pattern. `allow` continues the execution of the system call. `deny`(*retnum*) causes the execution of the system call to fail with return value *retnum*. `killProc` terminates the target process. `policyChange`(*policyfile*) dynamically switches the security policy to the one specified by file *policyfile*.

When giving the sample security policy, all system calls except `read` and `execve` will fail with an error code −1. Opening file `/etc/passwd` and all files under directory `/etc/cron.d` will fail. Any other file can be opened. When a target process invokes `execve` with the argument `/usr/bin/wserver`, the system switches to the security policy stored in file `wserver.pol`. The execution of the `execve` system call with other arguments will fail.

2.3 Advantages

The proposed system has the following advantages.

Difficulty in attacking the security system: If an attacker takes over a process in a target VM, the attacker cannot take control of the monitor VM and the monitor process because of the isolation between VMs.

Process-granularity execution control: A response action is applied to anomalous target processes only. Other processes in the same target VM are not affected. On the other hand, if the granularity of execution control is a VM, a possible response action will be a coarse one such as a VM restart. A VM restart has a serious drawback in that benign processes are also terminated.

Uniform control of multiple VMs: If the same program is running in multiple guest OSes on a VMM, the system can simultaneously control multiple process instances of the program with the same security policy. For example, if the same web server program is running in ten guest OSes on the same VMM, one monitor VM can control ten servers with the same security policy.

Security enhancement of unprotected OSes: The proposed system is also useful in virtual hosting in which each VM and guest OS is managed by a different administrator. Some guest OSes may be managed by a novice administrator and may not be sufficiently protected against attacks. The proposed system provides a safety net for such guest VMs and guest OSes.

3. IMPLEMENTATION

We implemented the system using a para-virtualization version of Xen VMM 3.0.3. We used the Linux kernel 2.6.16.19 as a guest OS.

3.1 Obtaining Process Information

`vps` is a command for obtaining information on a list of processes in the target VM specified by a given VM ID. The VMM obtains the information as follows. First, if a target VM is running, it is stopped by the VMM. The VMM switches its address space to the address space of the target VM by using the execution states kept by the virtual CPU of the target VM. After that, the VMM obtains a list of processes by using register values of the target VM and tracking a linked list of `task_struct` structures. Finally, the VMM switches the address space to the original one, and sends the obtained information to the monitor VM.

After the Linux kernel 2.6.0, a pointer to a `task_struct` structure is a member of a `thread_info` structure. The VMM reaches a pointer to a `task_struct` structure by calculating a pointer to a `thread_info` structure using the kernel stack pointer. After the Linux kernel 2.6.20, a pointer to a `thread_info` structure is also available from the GS or FS register.

The memory area for the Xen VMM is shared among all VMs. The virtual address space is not changed at a system call invocation. Hence, the VMM can directly access `task_struct` structures managed by guest OSes.

3.2 Managing Target Processes

When command `vcntl` is executed, the VMM adds specified process IDs or program file paths to the list of conditions for target processes.

When the `execve` system call is invoked in a target VM, the VMM checks whether a given program path is in the

above list. If it is, the VMM adds the ID of the invoking process to the list. Otherwise, the VMM does nothing.

When the `fork`, `vfork` or `clone` system call is invoked in a target VM, the monitor VM checks the given security policy. If it specifies that a new process is also controlled (the value of the `traceChild` field is `yes`), the VMM adds the new process to the list. Otherwise the new process is not controlled.

3.3 Controlling System Calls

When a system call is invoked in a target VM, the VMM checks whether the invoking process is in the list of target processes. If it is not, the VMM does nothing special and lets the system call continue. Otherwise, the VMM reads the execution state of the target VM and obtains the number and arguments of the system call. The VMM then notifies the monitor process that a system call is invoked. The monitor process decides on a response action and tells the VMM what it is. The VMM executes the action and continues the target process (if the target process is still alive).

A target process is suspended until a response action is taken. Note that, when a system call is checked, only a corresponding target process is suspended and the enclosing target VM is *not* suspended. Hence, negative effects on other processes due to the execution control are minimized.

Currently, the proposed system does not have a mechanism to prevent race condition attacks in which another thread modifies system call arguments [15]. Many techniques have been proposed to prevent race condition attacks [7, 8], and several of them can be applied to the proposed system. However a tradeoff between security and performance exists in prevention of race condition attacks. It is an interesting work to evaluate the effectiveness of the techniques in our setting.

3.4 Obtaining System Call Arguments

A monitor VM uses system call arguments to control the execution of system calls. When a system call is intercepted, the VMM stores the value of each system call argument. If the argument type is an on-memory object, the monitor VM reads the address space of the user process in a target VM. Examples of on-memory objects are path names given to an `open` system call and network addresses given to a `connect` system call.

When the monitor VM accesses an address in the user space of a target VM, a page fault may occur. The proposed system avoids page faults as follows. If a system call argument is a virtual address in the user space, the VMM reads the page table of the target process and checks whether a page enclosing the address is present. If not, the VMM makes the target VM do the page fault handling. When the guest OS completes the handling, the VMM first notifies the monitor VM of the system call invocation.

3.5 Response Actions

The VMM takes a response action to a target VM when an invoked system call matches to a rule in a given security policy. In the response action of system call failure (`deny`(*retnum*)), the VMM changes the invoked system call to `getpid`. The VMM replaces a return value of `getpid` with *retnum*. In the response action of killing a target process (`killProc`), the VMM changes the invoked system call to `getpid` and updates a member related to signal handling in the `task_struct` structure of the target process. It causes a kill signal to be sent to the target process at the exit of the system call.

3.6 System Call Interception

3.6.1 System Call Handling in Linux and Xen

The Linux kernel 2.6 running on the IA-32 architecture supports two schemes for invoking system calls. One uses software interrupts (int 0x80 instruction). The other uses the SYSENTER instruction. Either scheme is chosen at the boot time according to the CPU version.

SYSENTER is an instruction provided by IA-32 architectures later than Pentium II. The behavior of SYSENTER is configured with the value in the Model Specific Register (MSR). Updating values in MSR requires the highest privilege. When SYSENTER is executed, values in MSR are loaded in several registers, including the instruction pointer register and the stack pointer register. In addition, SYSENTER sets the privilege level to zero (the highest).

In the para-virtualization version of Xen, a system call is handled without passing through the VMM. When a system call is invoked, the control moves directly to a system call handler in a guest OS. The source code of a guest OS for Xen is modified to achieve that; a program part for configuring an interrupt descriptor table is modified.

3.6.2 System Call Interception in the Proposed System

We modified the para-virtualization version of Xen to intercept events related to system calls. The source code of a guest OS does not need to be modified to introduce our interception mechanism (the original guest OS code for Xen can be used with the proposed system).

To intercept system call invocations using software interrupts, the system patches the binary code of a guest OS kernel. When the `vconf` command is executed, the VMM overwrites the first byte of the code of a system call interrupt handler with a binary of an HLT instruction. The code address is collected when the guest OS is built. The HLT is a privileged instruction, so if it is executed at the user level, an interrupt is raised and the control returns to the VMM. To continue the intercepted process, the VMM emulates the overwritten instruction and passes the control back to the guest OS kernel.

To intercept system call invocations using SYSENTER, the system configures SYSENTER-related fields of MSR at the boot time. The values stored in MSR include an address of the VMM's code that handles system calls in a target VM. When a target process executes SYSENTER, the control moves to the VMM. Then VMM then checks the system call and controls the execution. After that, it sets the privilege level to one (the level of guest OS kernels in Xen for the IA-32 architecture) and moves the control to the code of the guest OS kernel that handles system calls. When a guest kernel attempts to modify SYSENTER-related values in MSR, the VMM intercepts and invalidates the attempt. As a result, the real MSR is not modified and the VMM keeps the virtual MSR value.

The VMM can intercept SYSEXIT instruction without special modification, because SYSEXIT traps if it is executed at privilege level one.

The VMM must also intercept the exit of system calls,

	Linux	Xen	Xen+Intercept (SYSENTER)	Xen+Intercept (interrupt)	Xen+Control (SYSENTER)	Xen+Control (interrupt)	Linux+ ptrace	Xen+ ptrace
getpid	0.16	0.37	1.17	2.08	13.5	15.2	14.7	21.5
open-close	2.66	3.30	5.07	6.84	57.9	68.3	52.5	64.1
fork-wait	38.7	153	158	159	242	258	89.4	225

Table 1: Microbenchmark results. ([μs])

because, in some system calls, including `fork` and `accept`, the execution state must be checked at the exit of the system call. The interception is achieved by binary patching of a guest OS kernel in the same way as system call invocations are intercepted using software interrupts.

4. EXPERIMENTS

We conducted experiments using the proposed system. In the experiments, we ran one monitor VM and one target VM on top of a VMM. The memory sizes of the VMs were 512 MB and 256 MB, respectively. The security policy given in the experiments allowed all executions of system calls.

We compared the execution times with the following settings.

Linux: A native Linux running on a physical machine

Xen: A Linux running on Xen

Xen+Intercept (SYSENTER): A Linux running on Xen with an extension for intercepting system calls using SYSENTER (system calls are intercepted but they immediately continue the executions)

Xen+Intercept (interrupt): The same setting as the above except that system calls are invoked with software interrupts

Xen+Control (SYSENTER): Linux running on the proposed system (system call is invoked with SYSENTER)

Xen+Control (interrupt): The same setting as the above except that system calls are invoked with software interrupts

Linux+ptrace: A Linux running on a physical machine with an extension for intercepting system calls using `ptrace` system call

Xen+ptrace: A Linux running on Xen with an extension for intercepting system calls using `ptrace` system call

We measured the overhead incurred by the proposed system using a microbenchmark. The platform was Pentium 4 3.0 GHz (with Hyper-Threading), and 1 GB main memory. The microbenchmark was composed of the three programs described below.

getpid: This repeated the invocation of the `getpid` system call.

open-close: This opened a file in a home directory and immediately closed it with `open` and `close` system calls.

number of requests	1	2	32	256	1024
Xen	1.02	0.94	0.87	0.84	0.85
Xen+Intercept	1.23	0.94	0.88	0.84	0.85
Xen+Control	1.57	1.54	1.50	1.49	1.50

Table 2: Apache benchmark results. (Time per request[ms])

fork-wait: This forked a process, and the parent process waited for the child process. The child process immediately completed its execution. A given security policy specified that a child process was also controlled. The parent process repeatedly invoked `fork` and `waitpid` system calls.

Each program invoked 10000 sets of system calls.

Experimental results are shown in Table 1. The results show that the overhead incurred by Xen+Control (the proposed system) was close to the overhead incurred by Xen+ptrace. This indicates that the proposed system did not impose a large overhead compared with an approach that monitors processes from the inside of a guest OS with a process tracing facility provided by the guest OS. The difference in overheads between Xen and Xen+Interrupt was much smaller than the difference between Xen+Control. This indicates the overhead due to system call interception was much smaller than the overhead due to execution control of system calls including event communication between different VMs via VMM.

Next, we measured the overhead imposed on the performance of the Apache web server using the ApacheBench benchmark. We executed a web server Apache 2.0.54 in a target VM and executed ApacheBench 2.0.40 on a different machine in the same local area network. The size of a requested file was 1 KB. The number of requests was 1, 2, 32, 256, or 1024. The platform running Apache was Pentium 4 3.0 GHz (with Hyper-Threading) with a 1 GB main memory and a 1 Gbps network card. The platform running ApacheBench was Pentium 4 3.2 GHz (with Hyper-Threading) with a 2 GB main memory and a 100 Mbps network card. In this experiment, system calls were invoked with software interrupts.

Experimental results are shown in Table 2. The results show that the performance of Xen+Intercept was almost the same as the performance of Xen. This indicates that system call interception by VMM has little impact on the performance of Apache. On the other hand, the difference in performance between Xen+Control (the proposed system) and Xen+Interrupt was not small. This indicates that most of the overhead incurred by the proposed system is not due to system call interception, but due to execution control of system calls, including event communication between different VMs via VMM.

The performance of Xen+Control (the proposed system) was 1.5 or 1.8 times lower than the performance of Xen. Though the performance degradation is not very small, the proposed system provides enhanced security at the expense of performance. We expect the users to well consider the requirement on security and performance in their setting.

5. RELATED WORK

Much work has been done on enhancing the security of VMs running on a VMM. sHype [13] and NetTop [3] provide infrastructure for controlling information flows and resource sharing between VMs. While the granularity of execution control in these systems is a VM, the proposed system controls execution at the granularity of a process.

Livewire [9] is a intrusion detection system that controls the behavior of a VM from the outside of the VM. As in this work, Livewire uses knowledge on the guest OS to understand the behavior in a monitored VM. Livewire's VMM intercepts write accesses to a non-writable memory area and accesses to a network device. On the other hand, the proposed system intercepts and controls system calls invoked by target processes.

A number of security systems based on system call interception have been proposed so far. Most of the systems including the systems in [5, 6, 10, 12, 14] run a program for security on the same OS as protected programs and potential malware. This makes it easier for malware to attack security systems. On the other hand, in the proposed system, a security system is running outside the guest OS protectes. It is difficult for an attacker to stop the security system because the attacker cannot even observe it.

IntroVirt [11] is a system that detects and responds to intrusions from the outside of the VM. It intercepts the execution of a vulnerable code part and executes vulnerability-specific predicates in the target system. While security checks in IntroVirt are performed according to the vulnerability specification and a corresponding predicate, the proposed system works based on specification of how to check and control system calls.

6. CONCLUSION AND FUTURE WORK

This paper has proposed a system that enhances the security of virtual machines by controlling the execution of system calls from another VM. The system uses the information on the guest OS kernel and controls processes according to a given security policy.

Future work will include the as followings. First, we would like to raise the abstraction level of a security policy to reduce the description effort required. Next, we would like to apply the system to a wider range of applications not limited to web servers. Finally, we would like to extend the system to detect and control kernel-level malware. Currently, an attacker can bypass the control by the monitor program if the attacker should subvirt the guest OS kernel running in a target VM. We are working on a technique for detecting and/or preventing that kind of attacks.

7. REFERENCES

[1] *VirtualBox*. `http://www.virtualbox.org/`.

[2] *VMware Server*. `http://www.vmware.com/products/server/`.

[3] *NetTop*, 2004. `http://www.hp.com/hpinfo/newsroom/press_kits/2004/security/ps_nettopbrochure.pdf`.

[4] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, New York, October 2003.

[5] H. Feng, O. Kolesnikov, P. Fogla, W. Lee, and W. Gong. Anomaly Detection Using Call Stack Information. In *Proceedings of the 2003 IEEE Symposium on Security and Privacy*, Oakland, May 2003.

[6] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff. A Sense of Self for Unix Processes. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, Oakland, May 1996.

[7] T. Garfinkel. Traps and Pitfalls: Practical Problems in System Call Interposition Based Security Tools. In *Proceedings of the 10th Annual Network and Distributed Systems Security Symposium*, San Diego, February 2003.

[8] T. Garfinkel, B. Pfaff, and M. Rosenblum. Ostia: A Delegating Architecture for Secure System Call Interposition. In *Proceedings of the 11th Annual Network and Distributed Systems Security Symposium*, San Diego, February 2004.

[9] T. Garfinkel and M. Rosenblum. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *Proceedings of the 10th Annual Network and Distributed Systems Security Symposium*, San Diego, February 2003.

[10] I. Goldberg, D. Wagner, R. Thomas, and E. Brewer. A Secure Environment for Untrusted Helper Applications. In *Proceedings of the 6th USENIX Security Symposium*, San Jose, July 1996.

[11] A. Joshi, S. King, G. Dunlap, and P. Chen. Detecting Past and Present Intrusions through Vulnerability-Specific Predicates. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, Brighton, October 2005.

[12] N. Provos. Improving Host Security with System Call Policies. In *Proceedings of the 12th USENIX Security Symposium*, Washington, DC, August 2003.

[13] R. Sailer, T. Jaeger, E. Valdez, R. Caceres, R. Perez, S. Berger, J. Griffin, and L. Doorn. Building a MAC-based Security Architecture for the Xen Opensource Hypervisor. In *Proceedings of the 21st Annual Computer Security Applications Conference*, Tucson, December 2005.

[14] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni. A Fast Automaton-Based Method for Detecting Anomalous Program Behaviors. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, Oakland, May 2001.

[15] R. Watson. Exploiting Concurrency Vulnerabilities in System Call Wrappers. In *Proceedings of the 1st USENIX Workshop on Offensive Technologies*, Boston, August 2007.