LSM-based Secure System Monitoring Using Kernel Protection Schemes

Takamasa Isohara, Keisuke Takemori, Yutaka Miyake
KDDI R&D Laboratories
Saitama, Japan
{*ta-isohara, takemori, miyake*}@*kddilabs.jp*

Ning Qu, Adrian Perrig
CyLab/CMU
Pittsburgh, PA, USA
{*quning, perrig*}@*cmu.edu*

***Abstract*—Monitoring a process and its file I/O behaviors is important for security inspection for a data center server against intrusions, malware infection and information leakage. In the case of the Linux kernel 2.6, a set of hook functions called the Linux Security Module (LSM) has been implemented in order to monitor and control the system calls. By using the LSM we can inspect the activity of unknown malicious processes. However, a sophisticated attacker could breach the kernel configurations using the rootkits. Furthermore since the monitoring results of the malicious process activity are stored as a file on Hard Disk Drive (HDD), it will be easily manipulated by the attacker. In this paper, we propose a secure monitoring scheme that addresses the attacks against the monitoring module and its result for security inspection of the data center server. The monitoring module is implemented as a LSM-based function and protected by the kernel protection technique. The integrity of the monitoring result is guaranteed by using a Mandatory Access Control (MAC) of the Linux kernel and a mechanism of the trusted process invocation. This mechanism can serve as an infrastrucuture of secure inspection platform for data center server because the integrity of the monitoring module and its result is guaranteed.**

***Keywords*-Secure system monitoring; Linux Security Module; Lifetime kernel code integrity; Mandatory Access Control;**

I. Introduction

Monitoring process activity and its file I/O history is important for security inspection of a data center server against intrusions, malware infection and information leakage [1], [2], [3], [4], [5]. For example, a $syslog$ is a well known mechanism for system monitoring on the UNIX system. It receives the monitoring information from a service process that is compliant with the $syslog$ mechanism. The syslog mechanism is categorized as an application-level monitoring because both syslog process and monitored process runs as user space processes. However, this mechanism can only monitor the behavior of the process that outputs the syslog compatible message and it is required that a system administrator configures the process to output the monitoring results for syslog mechanism beforehand. Hence, an unknown malicious process that is invoked by an attacker cannot be monitored by this scheme. Generally, a kernel-based monitoring scheme is preferred for monitoring the activity of any unknown process. Because the kernel-based monitoring can hook system calls, such as $read()$, $write()$ and $fork()$ functions, all of the processes can be monitored on the server. In the case of the Linux kernel 2.6, a set of hook functions called LSM [6] has been introduced for developing an access control module inside the Linux kernel. By using the LSM, we can easily implement a LSM-based monitoring module that can track the process and file I/O operations. However, when a sophisticated attacker gets the root privilege and manipulates the system configurations as well as the LSM-based monitoring modules, the result of the monitoring module is not reliable. Thus, the runtime integrity of the kernel including the LSM-based module should be protected. Moreover, since the monitoring result of a process activity is targeted by the attacker to manipulate the evidence of malicious activities, the access control scheme that prevents manipulations of the monitoring result is important for a secure monitoring scheme.

In this paper, we propose a secure monitoring scheme that prevents the manipulation attacks on both the system monitor module inside the kernel and the monitoring result on the HDD. The monitoring module is implemented as a LSM-based function and protected by the SecVisor [7] that guarantees the integrity of the runtime kernel. The integrity of the monitoring result is guaranteed by using a LSM-based Mandatory Access Control (MAC) and the DigSig [8] that provides a secure invoking mechanism for a trusted user application. As our scheme guarantees the integrity of the monitoring module and its result, it can be applied for the security inspection of data center server.

The remainder of this paper is organized as follows. Section II discusses related works. Section III presents the security threats and requirements for the secure monitoring on the data center server. Section IV describes the design and implementation of our proposal scheme. In Section V, we evaluate the overhead of our scheme on an experimental system. And finally, Section VI concludes this paper.

II. Related Studies

The guarantee for integrity of both the kernel and applications is important for secure system monitoring on the server. This section explains related research in this area.

A. LSM (Linux Security Module)

In order to develop a system monitoring module, logging function should be inserted at the point of interested system call execution.

In the case of the Linux kernel 2.4, modifying a system call table that contains the address of system calls in the kernel memory is widely applied to insert the logging function into any interested system call. However, it means that the function of system call can be modified, so the code of system call is easily replaced to malicious code. In other words, modifying the system call table is vulnerable to the rootkit infection.

On the other hand, a scheme for an access control module called LSM [6] is introduced to the Linux kernel 2.6. The LSM inserts calls to hook functions at original system call to strictly check the permission of operation which is executed on the operating system such as process creation, file I/O. A number of security sensitive systems are implemented by using LSM [9], [10]. LSM also inserts logging function at the entry point of each system call. Thus LSM can be also used to develop a system monitoring module.

Figure 1 shows a comparison between the scheme of modifying system call table and the LSM schemes. As shown in Figure 1, the LSM does not require modification of system call procedure. Thus, the LSM can achieve better secure guarantees of the system call monitoring module.

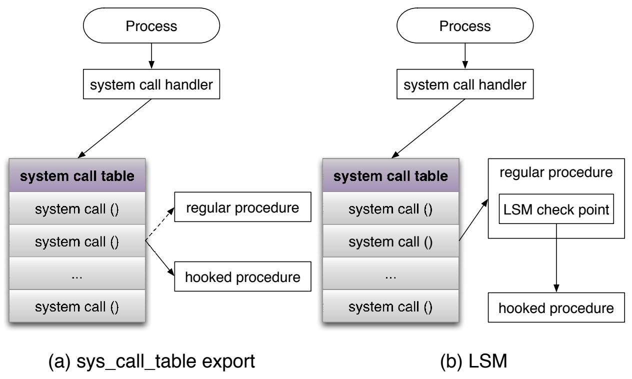


Figure 1. Comparison of system call hook method.

B. Process Authentication

The DigSig [8] provides a trusted invoking mechanism for a user application shown in Figure 2. All applications on the server are attached with code signatures that are used for authentication and verification of the code integrity.

When the *sys_execve* is called in the user space, *do_execve*, *search_binary_handler*, *load_elf_binary*, and *do_mmap* are called one after another in the kernel nspace. Finally, the *file_mmap* is called which is one of the check points of the LSM. Both the code image and the code signature of the user application are loaded into the *file_mmap*. A verifier hooks the *file_mmap* and checks the integrity of the user application by comparing its code image with its code signature. Only after the code is authenticated and the integrity of the code is verified, the trusted user application is invoked.

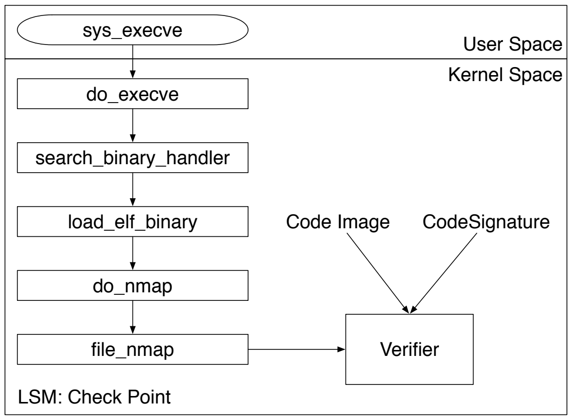


Figure 2. Integrity Check of Execution Code.

C. Lifetime Kernel Code Integrity

SecVisor [7] is a tiny hyper visor providing lifetime kernel code integrity by two steps: trusted boot and runtime kernel memory protection mechanisms.

The trusted boot mechanism achieves a chain of trust as follows: SecVisor runtime → Linux kernel. First, SecVisor loader starts SecVisor runtime by calling SKINIT, which performs the following steps: 1) initializes the CPU into some known states, 2) protects the target memory used by SecVisor runtime, 3) measures SecVisor runtime automatically and extends the measurement result into TPM, 4) and finally transfers control to SecVisor runtime. Then SecVisor runtime will further protect the target memory used as the Linux kernel, measure the Linux kernel, and extend the result into the TPM. When the integrity of the Linux kernel is verified, SecVisor runtime transfers control to the Linux kernel.

The non-manipulated kernel (called trusted kernel) gets control after being verified. The memory protection mechanism protects the kernel memory space from rootkits and direct memory access (DMA) attacks based on the support of AMD Secure Virtual Machine (SVM) technologies[11]. Figure 3 shows the memory permission mapping in user and kernel modes enforced by SecVisor. In kernel mode, only the memory containing the approved kernel code is executable, while in user mode, only application memory is executable. And in both modes, the approved kernel code is always read-only. Also, SecVisor sets up IOMMU to prevent any DMA access to the approved kernel code.

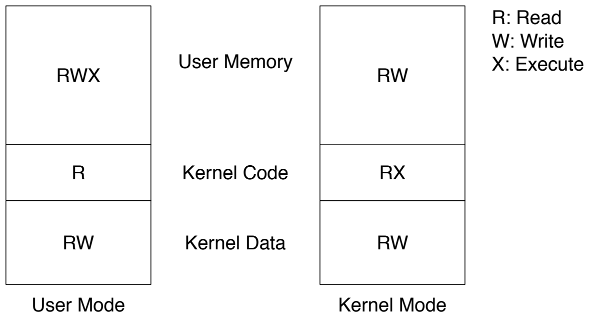


Figure 3. Dynamic Permission Control for User and Kernel Memories.

III. Security Threats and Requirements

A. Assumption of Security Threats

In this paper, we consider the process and file I/O behavior monitoring for a data center server that rarely ever shuts down and is managed by a trustworthy operator named administrator. In this model, the attacker is at a remote site. We assume that only the operator of the data center maintains the service processes and kernel configurations as a trusted administrator.

Since the administrator cannot preliminarily list the targeted malware process that is injected by the remote attacker, the system monitoring module should monitor all processes that run on the server.

In addition, when the attacker gains the root privilege on the data center server, the attacker can run malware process, modify the system configuration such as application and kernel components, and manipulate the monitoring results.

B. Requirements

Based on the above discussion, the requirements of a secure monitoring scheme for the data center server are listed as follows:

(A) The monitoring module should monitor all processes.
(B) The runtime integrity of the kernel including the monitoring module is protected against the attacker.
(C) The log file of the monitoring result recorded on a server disk should be protected against manipulation attack.

IV. Design and Implementation

In this section, we describe the design and implementation of a secure monitoring scheme on the data center server.

A. System architecture

Figure 4 shows the system architecture of our proposed monitoring scheme. Our scheme consists of two modules; a system monitoring module and a protection module for protecting the monitoring results. The monitoring module which is implemented based on LSM is protected by the SecVisor. The protection module is implemented with a access control functions and the DigSig functions that only invoke authorized applications on the server. In addition, the access control and the DigSig functions are also protected by the SecVisor.

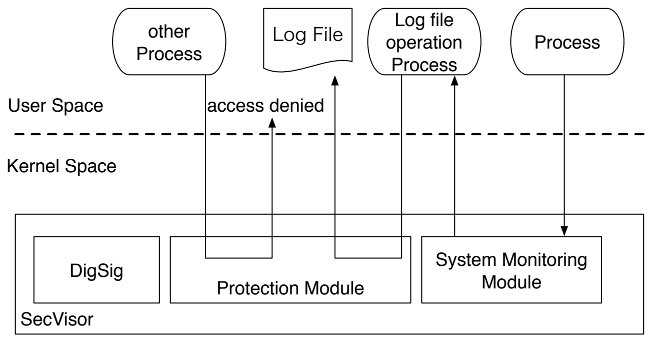


Figure 4. System architecture of proposed scheme.

B. Implementation of LSM-based system monitoring module

Considering the requirement (A), we implement the LSM-based system monitoring module which monitors the activity of all processes that run on the server. As the LSM provides about 140 hook functions, we need to select hook functions and pick out the information recorded into the system monitoring to implement the useful system monitoring module.

1) System monitoring points: Table I shows hook functions that have been implemented in the monitoring module. In the case of the process operation, we use *bprm_set_security*, *task_create* and *task_wait* hook functions to detect process invocation and termination events. In the case of the file I/O operation, because the *addition/deletion* for files and directories on the server is reasonable for system monitoring [12], we have selected hook functions suggested in Table I for inspection of the data center server.

Table I
LSM check points for system monitoring.

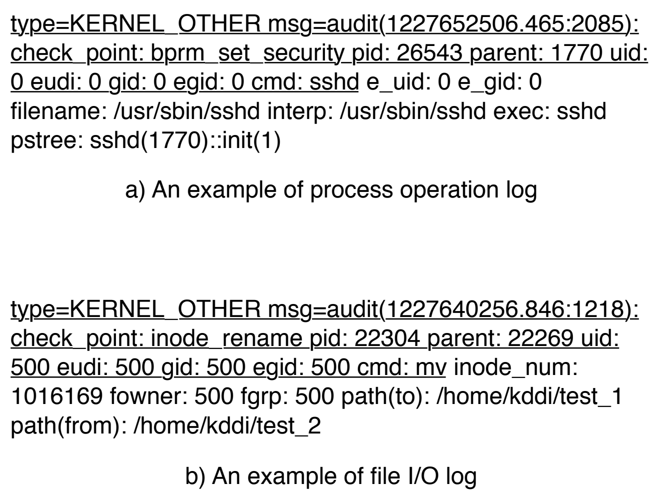
Category	Detection event	Name of LSM hook
Process	task create	bprm_set_security task_create
	task kill	task_wait
File I/O	file addition	inode_create
	directory creation	inode_mkdir
	directory deletion	inode_rmdir
	rename a file	inode_rename
	file deletion	inode_unlink inode_delete

2) Log format: Monitoring results are recorded as normal text file on the HDD. Table II shows the events recorded into the log file. In Table II, the common part is the information that is always recorded for both the process and file I/O log file. On the other hand, the original part is the recorded information specific to either the process or file I/O log file, respectively.

Figure 5 shows an example of log file. The underlined part shows the common part described in Table II. Figure 5 a) is collected when the log in event via a SSH service is occurred. Figure 5 b) indicates the rename operation of the sample text file from */home/kddi/test_1* to */home/kddi/test_2*.

Table II
MONITORING EVENTS.

Part	Category	Information
Common		· time of occurrence · name of hook function · process id that execute the events · process ID of parent process · user ID · group ID · execution command name
Original	Process	· name of binary file used for create process · process tree information from targeted process to *init* process
	File I/O	· inode number · path name



```
type=KERNEL_OTHER msg=audit(1227652506.465:2085):
check_point: bprm_set_security pid: 26543 parent: 1770 uid:
0 eudi: 0 gid: 0 egid: 0 cmd: sshd e_uid: 0 e_gid: 0
filename: /usr/sbin/sshd interp: /usr/sbin/sshd exec: sshd
pstree: sshd(1770)::init(1)
```

a) An example of process operation log

```
type=KERNEL_OTHER msg=audit(1227640256.846:1218):
check_point: inode_rename pid: 22304 parent: 22269 uid:
500 eudi: 500 gid: 500 egid: 500 cmd: mv inode_num:
1016169 fowner: 500 fgrp: 500 path(to): /home/kddi/test_1
path(from): /home/kddi/test_2
```

b) An example of file I/O log

Figure 5. An example of process and file I/O log.

C. Protection for Monitoring Module

By considering requirement (B), we take a countermeasure against kernel manipulation.

The attacks that breach kernel are categorized into two cases: the static attack and the dynamic attack.

In the case of a static attack, the attacker attempts to unload the system monitoring module.

Because of the countermeasure against static attack, the system monitoring module is compiled as a static kernel module to prevent the unload of module over the lifetime of the data center server. By this action, the attacker has to build a kernel and reboots the data center server to obtain the kernel that does not include the system monitoring module.

In the case of a dynamic attack, the attacker attempts to compromise the kernel space memory by using malicious code injection and execution. Compromising the kernel gives the attacker complete control over the system and malicious code execution can intercept and manipulate a correct system monitoring activity. According to [7], the malicious code injection is categorized as following three ways; intentional misuse of kernel modularization support, exploit software vulnerabilities, and kernel memory corruption via DMA-capable peripheral devices.

We apply SecVisor to prevent these attacks. The SecVisor protects kernel memory space by transparently modifying execution permission of user and kernel memory independent of kernel page tables, and ensures that only approved code can execute in kernel mode. In addition, SecVisor sets up IOMMU to prevent any DMA access to the approved kernel code.

D. Protection for Monitoring Result

By considering requirement (C), we propose the protection technique for the monitoring result stored in a HDD file against the manipulation attack. The protection technique combines with the MAC and the DigSig as shown in Figure 6.

The MAC controls file access permission of a dedicated process . Using the MAC is a reasonable way to protect the monitoring result and MAC function is easily achieved by using LSM-based security focused operating system [9], [10]. However, when the attacker obtains a security administrator account who can controls the MAC policy, attacker can inject and invoke a malicious process which is permitted to read/write a monitoring result on the data center server. Thus the attacker also can obtain the read/write permission through the malicious process. Hence, the mechanism which prevents the execution of process that illegally obtains the read/write permission of system monitoring results should be implemented on the data center server. To achieve this kind of function, the DigSig module is introduced since DigSig invokes only authorized applications on the server.

As both the MAC and the DigSig are implemented as a static kernel module , they are protected against the static and dynamic attacks in the same manner as described in section IV-C.

V. EVALUATION

In this section, we report the performance overhead of our LSM-based system monitoring mechanism that includes both MAC and DigSig.

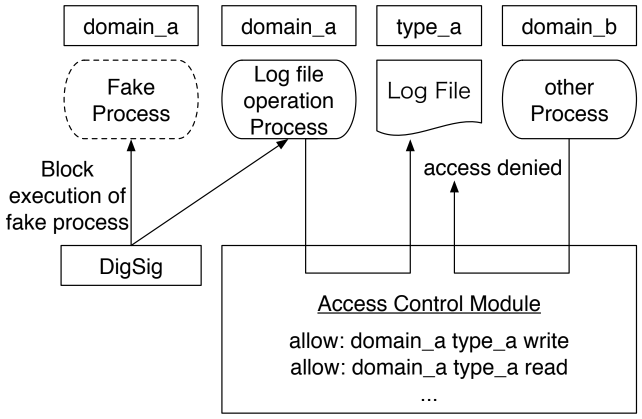

domain_a
domain_a
type_a
domain_b
Fake Process
Log file operation Process
Log File
other Process
Block execution of fake process
access denied
DigSig
Access Control Module
allow: domain_a type_a write
allow: domain_a type_a read
...

Figure 6. Architecture of protection technique for monitoring result.

A. Evaluation System

Our evaluation platform is implemented on a HP Compaq dc 5750 Micro tower that has 2.2 GHz AMD Athlon64 (Dual-core CPU) with AMD SVM hardware virtualization support and 2 GB RAM. The overhead is measured on the Fedora Core 6 Linux distribution. Our monitoring mechanism, DigSig and SecVisor are implemented on the Linux kernel 2.6.20.14. Table III shows components of evaluation systems.

Table III
MAJOR COMPONENTS OF EVALUATION SYSTEMS.

Evaluation System	Components
(i)	Linux kernel 2.6.20.14
(ii)	Linux kernel 2.6.20.14 + SecVisor
(iii)	Linux kernel 2.6.20.14 + System monitoring mechanism
(iv)	Linux kernel 2.6.20.14 + SecVisor + System monitoring mechanism

B. Micro benchmark

Since we implement a set of hook functions for the process and file I/O operation, we use a subset of process and file system micro benchmarks from `lmbench` [13] to measure the overhead of operations for process and file.

Table IV shows the results of our experiments. Null Call indicates the round trip time between user mode and kernel mode. Both (ii) and (iv) shows dramatically increased overhead which is incurred by SecVisor. For the benchmark of Fork and Exec besides the overhead incurred by SecVisor, our monitor mechanism also introduces a large overhead in which is shown in (iii). The file create and delete micro benchmark also shows a bigger overhead in (iii) than in (ii) because of the file I/O hook functions for recording the monitoring results. In our system monitoring module, file create is checked by 1 hook function *inode_create* but the file delete is checked by 2 hook functions *inode_unlink* and *inode_delete*, so the file delete benchmark shows a bigger overhead than the file create benchmark. For the fork benchmark to the file delete benchmark, (iv) shows the biggest overhead due to using both SecVisor and the LSM-based system monitoring module.

Table IV
EXECUTION TIMES OF `LMBENCH` PROCESS AND FILE MICRO BENCHMARKS. ALL TIMES ARE IN μS.

System	Null Call	Process		File	
		Fork	Exec	Create	Delete
(i)	0.09	117	353	13.4	12.1
(ii)	4.84	1398	3434	21.5	16.9
(iii)	0.13	584	1267	256.3	691.6
(iv)	4.81	4709	7771	484.3	1280.4

C. Application benchmark

As the application benchmark, we choose the Linux kernel compile. We compile the source code of kernel version 2.6.20.14 by executing "make" in the top-level source directory.

Our result is presented in Figure 7. The user CPU time indicates the execution time consumed in user space, system CPU time indicates the kernel mode execution time including system call and other kernel operations.

In the case of (ii), the system CPU time increases a lot because the SecVisor needs to switch memory access permission frequently. In the case of (iii), the other CPU time increases because the system monitoring module takes time to record the monitoring results into the HDD. In the case of (iv), the overhead comes from both the hook function of the monitoring module that processes the file I/O operations and the SecVisor that controls the execution permissions of user and kernel memory. Since the kernel compile time of (iv) is only 1.84 times as the regular Linux in (i), the overhead is acceptable.

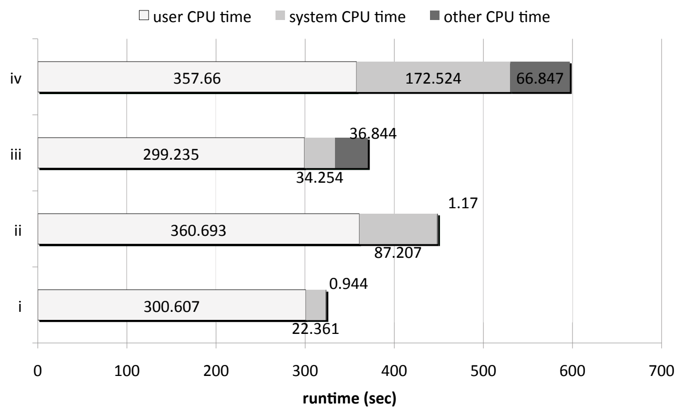


Figure 7. Application performance comparison between regular kernel and our scheme.

VI. Conclusion

This paper presents a secure system monitoring mechanism on a Linux 2.6. The system monitoring module is implemented as a LSM-based module and is protected by SecVisor. The protection scheme for the monitoring results prevents malicious manipulation by combining the function of the MAC and DigSig. Moreover, the MAC and DigSig modules are also protected by the SecVisor. We have implemented our proposed mechanism and evaluated the performance overhead. The results show that the overhead of the proposed mechanism is acceptable with the secure monitoring. As our proposal mechanism guarantees the integrity of the monitoring module and its result, it is expected to be adopted for the security inspection of data center server.

References

[1] S. Forrest, S. Hofmeyr, A. Somayaji, and T. Longstaff, "A sense of self for unix processes," in *Security and Privacy, 1996. Proceedings., 1996 IEEE Symposium on*, May 1996, pp. 120–128.

[2] S. A. Hofmeyr, S. Forrest, and A. Somayaji, "Intrusion detection using sequences of system calls," *J. Comput. Secur.*, vol. 6, no. 3, pp. 151–180, 1998.

[3] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni, "A fast automaton-based method for detecting anomalous program behaviors," in *SP '01: Proceedings of the 2001 IEEE Symposium on Security and Privacy*. Washington, DC, USA: IEEE Computer Society, 2001, p. 144.

[4] E. Eskin, W. Lee, and S. J. Stolfo, "Modeling system calls for intrusion detection with dynamic window sizes," in *In Proceedings of DARPA Information Survivabilty Conference and Exposition II (DISCEX)*, 2001.

[5] C. Warrender, S. Forrest, and B. Pearlmutter, "Detecting intrusions using system calls: Alternative data models," in *In IEEE Symposium on Security and Privacy*. IEEE Computer Society, 1999, pp. 133–145.

[6] C. Wright, C. Cowan, S. Smalley, J. Morris, and G. Kroah-Hartman, "Linux security modules: General security support for the linux kernel," in *USENIX Security Symposium*, August 2002, pp. 17–31.

[7] A. Seshadri, M. Luk, N. Qu, and A. Perrig, "SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes," in *ACM SOSP Symposium*, October 2007.

[8] A. Apvrille, M. Pourzandi, D. Gordon, and V. Roy, "Stop malicious code execution at kernel-level," in *Linux World*, vol. 2, January 2004.

[9] N. S. Agency, "Security-enhanced linux," http://www.nsa.gov/research/selinux/.

[10] "Tomoyo linux project," http://tomoyo.sourceforge.jp/index.html.en.

[11] "AMD virtualization," http://www.amd.com/us-en/0,3715_15781,00.html?redir={SQOP08}.

[12] "Tripwire," http://www.tripwire.com/.

[13] L. McVoy and C. Staelin, "lmbench: portable tools for performance analysis," in *ATEC '96: Proceedings of the 1996 annual conference on USENIX Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, 1996, pp. 23–23.