Electronic Preprint for Journal of Information Processing Vol.22 No.4

Regular Paper

Secure and Fast Log Transfer Mechanism for Virtual Machine

Masaya Sato[1,a)] Toshihiro Yamauchi[1]

Received: December 1, 2013, Accepted: June 17, 2014

Abstract: Ensuring the integrity of logs is essential to reliably detect and counteract attacks because adversaries tamper with logs to hide their activities on a computer. Even though some studies proposed various protections of log files, adversaries can tamper with logs in kernel space with kernel-level malicious software (malware) because file access and inter-process communication are provided by an OS kernel. Virtual machine introspection (VMI) can collect logs from virtual machines (VMs) without interposition of a kernel. It is difficult for malware to hinder that log collection, because a VM and VM monitor (VMM) are strongly separated. However, complexity and unnecessary performance overhead arise because VMI is not specialized for log collection. This paper proposes a secure and fast log transfer method using library replacement for VMs. In the proposed method, a process on a VM requests a log transfer to a VMM using the modified library, which contains a trigger for a log transfer. The VMM collects logs from the VM and isolate them to another VM. The proposed method provides VM-level log isolation and security for the mechanism itself with low performance overhead.

Keywords: secure logging, virtual machine, library modification, digital forensics.

1. Introduction

Logging information about activities and events in a computer is essential for troubleshooting and for computer security. Logs are important not only for detecting attacks, but also for understanding the state of the computer when it was attacked. The importance of logs for computer security is described in Special Publication [1]. Adversaries tamper with logs to hide their malicious activities and the installation of malware on the target computer [2], [3], [4]. If logs related to those activities are tampered with, detection of problems might be delayed, and the delay could cause further damage to services. In addition, log tampering impedes the detection, prevention, and avoidance of attacks. With the growth of cloud computing in recent years, Virtual Machine (VM) security has become more important [5], [6]. This is particularly true for log forensics in cloud applications has [7]. However, existing logging methods are not designed for VMs or cloud applications.

Secure logging using VMs provides logging integrity and completeness [8]. Boeck et al. proposed a hardware-based secure log transfer method that proves log entry origin with respect to both machine and application in a hardware-based way [9]. These methods ensure confidentiality and integrity of log data while stored or in transit between local and remote machines. While this method can prevent attacks on logging daemons, adversaries can still tamper with logs in kernel space. Logs must go through an operating system (OS) kernel when transferred out of the computer. If malware has been installed, logs could be tampered with in kernel space. SecVisor [10] is a method that prevents the execution of illegal codes in kernel space. However, these methods depend on the structure of the OS kernel, making it difficult to adapt to various OSes. In a situation in which a single machine provides many VMs, different OSes could be running on each VM. VM introspection (VMI) [11] can be considered a VM logging method. However, it has difficulty with information granularity. Because it collects low-level VM information, interpretation and reconstruction of information is required.

This research involves log protection. Even though the importance of logging for cloud application has increased [7], there is no method specialized for logging in a VM environment. VMs are commonly used to provide a cloud computing environment. Providing services such as logging degrades the performance of application programs (APs) on VMs [8]. Specifically, a web server frequently logs information about access. Turning off the logging operation in the Apache web server results in a 22% performance improvement [12]. Thus, logging is one of the main reasons of performance degradation. Therefore, reducing performance overhead incurred by additional services is an important challenge.

This paper proposes a secure and fast log transfer method using library replacement. To trigger a log transfer to a VM monitor (VMM), we embed an instruction in a library function to trigger VM exit. On Linux and FreeBSD, we modified the standard C library, `libc`, which contains standard logging functions. When the VMM detects a VM exit, it collects the logs generated by APs in the source VM and transfers them to the logging VM, which stores the logs to a file. We assumed that the modified library is secured in memory using a method [13] that protects a specific memory area from being modified by kernel-level malware.

[1] Graduate School of Natural Science and Technology, Okayama University, Okayama 700–8530, Japan
[a)] m-sato@swlab.cs.okayama-u.ac.jp

When the proposed method is used, adversaries cannot tamper with logs in kernel space, because the VMM collects logs before they reach it. Because modification to a library is kept minimal, adapting to different OSes requires less effort. Performance degradation is minimal because the overhead is incurred only when an AP calls a logging function. The proposed method replaces only a library, which includes a function to send logs to a syslog daemon. Therefore, we can make the possibility of bug inclusion low. Additionally, bugs in a library have less effect than those in a kernel.

This paper also describes evaluations of the proposed system. We evaluate the system from the standpoint of log security, adaptability to various OSes, and performance overhead. To evaluate the system from the standpoint of log security, we analyze the security of a logging path. Experiments to tamper with logs in the logging path are also described. Adaptability of the proposed system is provided with case studies adapting to various OSes. Performance evaluations with APs commonly used in servers are described. VMI causes substantial overhead [14]. For practical use, performance degradation must be minimal. With these evaluations, this paper shows how the proposed system is practical for generally-used APs and a multi-VM environment.

The basic concept of the proposed system has been presented before [15]. This paper describes its detailed design, implementation, security analysis, and evaluation results.

The contributions made in this paper are as follows:

- We propose a secure log transfer method that replace a library in a VM. With the proposed system, a kernel-level malware cannot delete or tamper with logs. Moreover, by comparing collected logs and tampered-with logs, we can identify the area that is tampered with.
- We design a tamper-resistant system using a VMM. We implemented our entire system inside the VMM because of its attack-resistance.
- The proposed system is implemented with minimal modification to `libc`. Although making no modification would be preferable, modifying the library provides two advantages: slight overhead and ease of adaptation to various OSes. This also reduces the possibility of bug inclusion, thereby making the system more secure.

2. Log Transfer Method

2.1 Existing Log Transfer Methods

In Linux and FreeBSD, syslog is a protocol for system management and security monitoring. Syslog consists of a syslog library and a syslog daemon. A conceptual diagram of the logging path is shown in **Fig. 1**. As shown in Fig. 1, (1) an AP generates a log entry, and then (2) sends the log to a logging daemon. The logging daemon (3) receives and shapes the log. Finally, the logging daemon (4-A) stores the log to a local storage or (4-B) transfers the log to a logging daemon on a remote machine.

New syslog daemons and protocols [16], [17], [18], [19] have been developed to achieve greater security. New syslog daemons can transfer logs to out of a computer and can encrypt syslog traffic using transport layer security (TLS). These methods secure logs in the path (4-B). However, during log transfer, adversaries

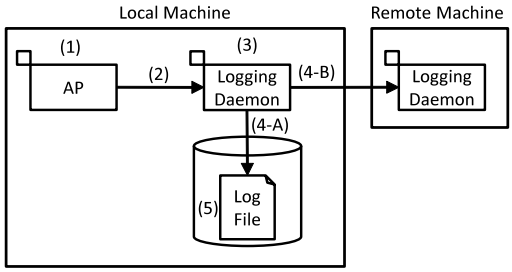

Fig. 1 Conceptual diagram of the logging path.

can delete or tamper with the log with a kernel-level attack [3] along path (2). Other methods using inter-process communications can be attacked in the same manner. Other malware tamper with logs by replacing syslog daemons [2]. If the logging daemon is tampered, all logs generated by the logging daemon are unreliable.

VMI [11] inspects VMs by retrieving hardware information about the target VM and constructing a semantic view from outside the VM. ReVirt [20] collects instructions-level information for VM logging and replay. CloudSec [21] performs a fine-grained inspection of the physical memory used by VMs and detects attacks that modify kernel-level objects. While these methods enable us to collect information inside VMs, they increase the complexity of the semantic view reconstruction and performance overhead. In addition, the reconstruction of a semantic view strongly depends on the structure of the OS.

To overcome this problem, in-VM monitoring methods [14], which inserts an agent into a VM, is proposed. It protects the agent from attacks from inside the VM. Inserting an agent is a practical and efficient way to collect information, however, it is difficult to adapt to various OSes because the implementation of an agent depends on the structure of the OS. The VMM-based scheme [22] can collect logs inside VMs without modifying a kernel or inserting agents. However, it has a large overhead and strong dependency on the architecture of the OS.

2.2 Problems of Existing Methods

Existing methods have the following four problems:

(1) Transferring log via inter-process communications can be preempted by kernel-level attacks.
(2) Collecting logs inside a VM by monitoring the behavior of APs or OSes cause unnecessary performance overhead.
(3) Collecting logs from various OSes requires efforts to adapt the method to a variety of OSes.
(4) Additional code increases the likelihood of bugs in the system.

No suitable method is currently available to transfer logs out of the VM. For security management, a secure logging method is required. Under the existing logging method, even if the existing logging functionality on a computer is compromised, it is possible to be aware of the existence of attacks by analyzing logs transferred by the computer. Because the amount of logs is changes from a regular operation, an administrator of the compromised computer can predict the existence of attacks. However, precise tracing of the attacks is difficult. But, if adversaries tamper with logs to hide their activities, predicting the existence of attacks is

considerably difficult because the amount of logs is not changed significantly. Monitoring from outside the VM is a new approach, because the monitor itself is secured by VM-level separation. On the other hand, the information obtained by the method is difficult to translate into a semantic view or is too fine-grained. While VMI and other introspection methods securely collect information inside a VM, constructing the semantic view of the VM is strongly depends on the structure of the target OSes. Adapting those methods to various OSes is nontrivial work. Inserting an agent into a VM can cause undesirable effects and make the VM unstable.

3. Secure and Fast Log Transfer by Library Modification in VM

3.1 Scope and Assumptions

This paper covers log protection from tampering via attacks to the kernel, to the logging daemon, and to files that contain logs. Attacking specific APs requires nontrivial work and it cannot tamper with logs completely; therefore, adversaries attack the point where all logs go through. For these reasons, preventing log tampering in kernel space and in a logging daemon is a reasonable challenge.

We assume attacks for a VMM is difficult. Even if a report [23] describes subverting the Xen hypervisor is possible, the condition that allow a such attacks is actually limited. Furthermore, attacks to VMMs can be detected by integrity scanning [24]. Therefore, we assume that attacking VMMs is sufficiently challenging.

3.2 Objectives

The objectives of this paper are as follows:

Objective 1 To propose a fast and tamper-resistant log transfer method.

Objective 2 To propose a log transfer method that is easy to adapt to various OSes.

The objective of our research is to address problems detailed in Section 2.2. To address those problems, providing a tamper-resistant log transfer method is necessary. Specifically, we aim to prevent log tampering from kernel-level malware like adoreng [3]. Moreover, low overhead is desired to implement the method to APs in the real world. Further, an OS-independent method is preferable, because it is assumed that various OSes are running on each VM.

3.3 Approach

Integrating logging modules into a VMM is an efficient way to secure the modules [8]. However, existing logging methods focus on logging and replay for VMs because those methods utilize information that can be collected natively by the VMM, including disk access, execution of sensitive or privileged instructions. Contrary to existing approaches, we focus on logs dealt by syslog.

Some studies focus on VMI, which collect information inside a VM. However, VMI methods cause performance degradation. Therefore, we focus on the speed of log transfers. We use the VMM to integrate a logging module to the outside of a VM for security and modify the library on a VM for fast log collection.

3.4 Requirements

To achieve the objectives, the followings are required.

Requirement 1 Transfer logs as soon as possible.

Requirement 2 Isolate logs from the VM.

Requirement 3 Secure the log transfer mechanism itself.

Requirement 4 Make the log transfer method OS-independent and small.

Requirement 5 Reduce unnecessary overhead related to log transfer.

In a logging path, logs generated by a process are passed to a kernel because the kernel provides the ability to send messages to other processes. Therefore, to prevent log tampering in kernel space, it is necessary to collect logs from outside the VM before the logs reach kernel space. To prevent tampering of log files, they must be isolated from the VM. To ensure the security of the log transfer method itself, we install the method outside the VM. With low dependency on the OS, migration to other OSes becomes easy. Moreover, a smaller program size helps to reduce the possibility of bugs. A VM exit, which is a CPU-mode transfer between a VM and a VMM, can cause additional overhead. To adapt the method to APs in the real world, unnecessary VM exits must be removed.

3.5 Overview of Proposed Method

The overall design of the proposed system is shown in **Fig. 2**. In the proposed system, the target VM works on a VMM and the VMM collects logs from the VM. We assume that all of the VMs are fully virtualized by Intel VT-x. An AP on the target VM can transfer logs with the proposed system as follows:

(1) An AP requests a log transfer to a VMM.
(2) The logging module inside the VMM receives the request and copies logs from the AP to the buffer inside the VMM.
(3) The VMM sends a notification to a logging AP inside the logging VM. Then, the VMM sends the logs to the logging AP.
(4) The logging AP receives the logs and stores them to a file. The logging VM accepts logs only from the VMM.

We modified the VMM to transfer logs from the target VM to the logging VM. The logging module, the log storing module and the buffer VMM are additional parts of the original VMM. We modified `libc` in the target VM to send a log transfer request

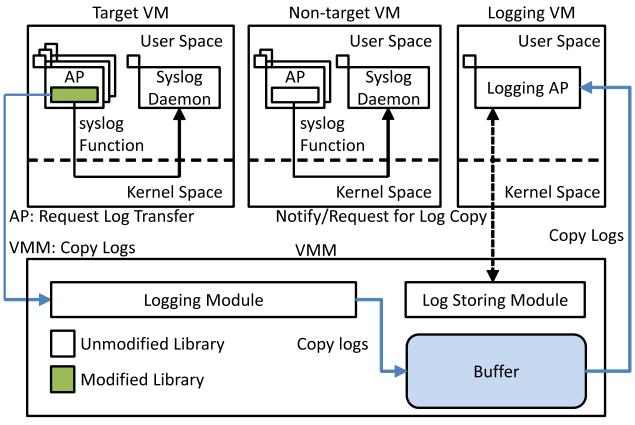

Fig. 2 Overview of proposed system.

to the VMM in each call of the syslog function. The modified library executes an instruction that causes a VM exit, which triggers a log transfer to the logging VM before sending logs to the logging daemon in the current VM. Only a VM that contains the modified library can send the request. In Fig. 2, the target VM requests a log transfer in every syslog function call; on the other hand, the non-target VM never makes the request.

The proposed method can be implemented for various types of VMMs. The VMM can be classified into two types: native virtual machine monitor (type 1) and hosted virtual machine monitor (type 2) [25]. A type 1 hypervisor traps privileged or sensitive instructions while a type 2 hypervisor interprets all instructions in a virtualized environment. With a type 1 hypervisor, we must modify a library on the VM to execute sensitive instructions to request a log transfer. Because a type 2 hypervisor basically interprets all instructions, a hypervisor can detect log generation and log transfer to a syslog daemon inside that VM. Thus, implementing our proposed method with type 2 hypervisor is easy. In this paper, we demonstrate implementation on a type 1 hypervisor because of its advantage in performance.

Collecting logs immediately after the invocation of the syslog function fulfills requirement 1. With this feature, tampering logs in kernel space is impossible. Using the logging VM to store logs fulfills requirement 2. Resources allocated to a VM, such as memory, network, disk space, and others, are separated from resources allocated to another VM; therefore, it is difficult to tamper with logs outside the VM during an attack. It is also difficult to attack a VMM from inside a VM; therefore, using a VMM and modifying a library fulfills requirement 3. Library modification also makes OS-adaptation easier and fulfills requirement 4. Finally, VM exits occur only when a syslog library function is called; therefore, requirement 5 is fulfilled.

3.6 Comparison of the Proposed Method and VMI

The proposed method and VMI are similar from the standpoint of collecting information inside the VM. However, there are the following differences between them:

- Security of logs.
- Dependency on data structures in the VM.
- Overhead.

The proposed system can achieve greater security of logs than a VMI. The VMI collects information about VMs by monitoring hardware states and some events. However, it is difficult to detect log generation by monitoring hardware states or events. Even if the VMI can detect log generation, when the VMI detects it after a mode transition to kernel space, logs can be tampered by kernel-level malware. By contrast, kernel-level malware cannot tamper with logs because the trigger of log transfer is given by a library in the user space of each VM.

To inspect the state of a VM, the VMI collects some information strongly related to the data structures in the VM. Thus, VMI must have enough knowledge about the layout of the data structures in the VM. Additionally, to inspect the state of a VM, the VMI must collect a lot of information (e.g., process list, process descriptor). This creates a strong dependency on the version of the OSes in the VMs.

As just described above, the VMI can inspect the state of a VM with fine-grained information; however, it creates a strong dependency on the data structure in the VM along with some overheads. On the other hand, the proposed system cannot collect much information about the VM; it achieves weak dependency on the data structures in the VM and has low overheads. VMI has a large overhead because it monitors the state of the VM with various fine-grained information. Research [14] shows that VMIs causes 690% in overhead while monitoring process creation. On the other hand, in-VM monitoring causes only 13.7% in overhead. Thus, the approach of the proposed system is efficient because the system can be considered as one of an in-VM monitoring system. Additionally, our proposed system only monitors the invocation of the syslog function. Therefore, overhead related to the proposed system arises only when an AP invokes the syslog function.

4. Implementation

4.1 Log Transfer Flow

Transferring logs from a VM to a VMM takes place in two phases: requesting the log transfer and copying the log. This section describes the implementation of each phase in Section 4.2 and Section 4.3, respectively. The modified code for the `libc` library is shown as Fig. 6 and explained in Section 6.3.

After transferring the logs to a VMM, the VMM sends the logs to the logging VM. The implementation of this phase is described in Section 4.4.

4.2 Log Transfer Request

We embed a `cpuid` instruction in a library to request the log transfer to the VMM from an AP. The instruction does not affect the CPU state; however, if executed in a virtualized environment, the instruction causes a VM exit. Therefore, we embedded the instructions into a library to request the copying of the logs to the external VMM before sending the logs to a logging daemon. The interface to the log transfer request is shown in **Table 1**. The embedded codes set the appropriate values to the registers and execute the `cpuid` instruction. Additional codes are shown in Section 6.3.

We utilize the `cpuid` instruction to counteract the detection of our approach that scans memory or a library file. One typical instruction is to call a VMM using `vmcall`. If we use the `vmcall` instruction as a trigger of the log transfer, adversaries can easily detect our approach by scanning memory because the instruction is not usedin regular APs. We count the number of `cpuid` and `vmcall` instructions included in `eglibc-2.11.3`. From the result, while `cpuid` appeared 13 times, `vmcall` was not found. To detect the embedding of a `cpuid` instruction, adversaries must identify where `cpuid` instructions exist or verify the library that has changed using the signatures of each library. For these rea-

Table 1 Interface of log the transfer.

Register	Explanations
rax	0xffff: the value represents a log transfer request.
rbx	Address of the buffer that contains logs to transfer.
rcx	Length of logs to transfer.

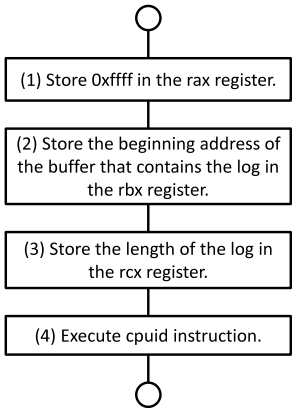

Fig. 3 Flow of log transfer request.

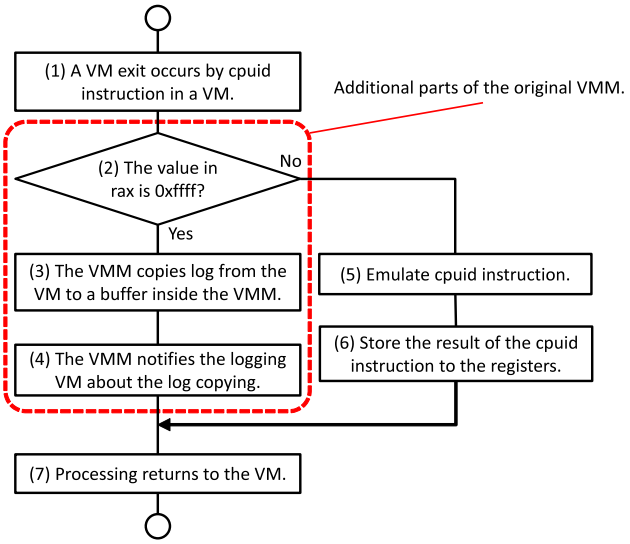

Fig. 4 Flow of log copying from AP to VMM.

sons, to make detection of our approach harder, we utilize the `cpuid` instruction.

Figure 3 depicts the flow of the log transfer request. At first, the AP on the target VM stores `0xffff` in the `rax` register, the starting address of the buffer in the `rbx` register, and the length of the buffer in the `rcx` register. Then, the AP executes the `cpuid` instruction to request a log transfer.

4.3 Log Copying from Target VM to VMM

Figure 4 depicts the flow of log copying by a VMM. A `cpuid` instruction triggers log transfer. After detecting the instruction, the VMM copies logs from the AP and notifies the logging VM if the value contained in the guest's `rax` register is `0xffff`. If not, the VMM does not copy logs and only emulates the instruction. If the buffer length is larger than the length indicated by the `rcx` register, the designated length of the buffer is copied to the VMM.

As shown in **Fig. 5**, the buffer inside the VMM is implemented as a ringed buffer to reduce the amount of logs lost during a high-load situation. Step (4) only sends notifications. When multiple VMs request log transfer, all logs are copied to the buffer inside the VMM. To identify logs which are from the VM, the VMM appends the VMID (Virtual Machine ID) to the head of the log entry.

In step (3), if the length of the logs is larger than the remaining space of the buffer inside the VMM, the VMM suspends the target VM and only copies part of the logs, whose length is less than the remaining space. When copying of the accumulated logs from the VMM to the logging VM is completed, the VMM resumes the copying of logs from the target VM. The VMM loops this procedure while the logs, which are requested by an AP to be transferred, remain in the target VM. Thus, if an AP requests log transfers with larger buffer lengths than the capacity of the buffer inside the VMM, the VMM can copy all the logs completely without loss. Because the target VM is suspended for the log transfer, the effects of the attack are limited. However, a log transfer request with a large buffer brings considerable performance degradation for the target VM.

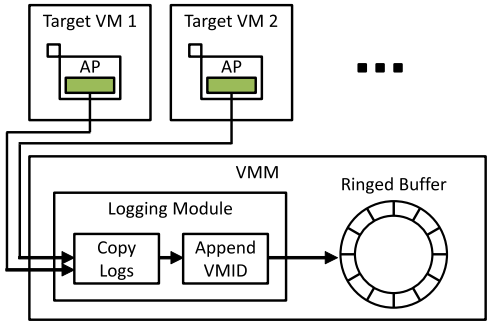

Fig. 5 Logging module and ringed buffer.

4.4 Log Copying to the Logging VM

Log copying to the logging VM is asynchronous, making the duration of log copying as short as possible. The VMM sends a notification that logs have accumulated in the buffer inside the VMM. The logging AP on the logging VM receives the notification and responds to it with a notification that the logging AP is ready to receive log messages. Here, the logging VM is separated from the target VM. After receiving the request, the VMM copies logs accumulated in the buffer to the logging AP. The logging AP writes logs to the file on the logging VM using the syslog function. Each log entry has a VMID to identify the log entry's origin. Except for this VMID, log entries stored by the logging AP have the same format as syslog, making existing log analysis tools usable for logs collected using the proposed method. Additionally, existing secure logging methods and log file encryption are applicable to the proposed method whereby the logging AP stores logs using the syslog function.

5. Security Analysis and Consideration

5.1 Limitations

If the adversaries are worried about the proposed mechanism, they can create ready-made APs which do not generate any logs. If the ready-made APs are compiled with a dynamic linked `libc`, the proposed method can possibly transfer logs. If the ready-made APs are static linked programs, transferring logs using the proposed method is impossible. However, if an AP stops sending logs, the system administrator of that computer would be aware of the existence of the attack by comparing logs before and after the logging functionality is suspended.

5.2 Logging Path Log Security

As shown in Fig. 1, a logging path using syslog has five phases. From this observation, we see that logs can be tampered with at the following points: (1) the time at which a process generates the log, (2) the time between the sending of the log and its receipt by the syslog daemon, (3) the time between receiving a log and storing it to a file, and (4) the time after the output of the log.

Kernel-level malware such as *adore-ng* [3] can tamper with logs at times (2) and (3). Attacks on a syslog daemon such as *tuxkit* [2] can tamper with logs at time (3). Adversaries who have privileges to write to the log file can tamper with logs at time (4).

With the proposed method, a log in the VM is transferred to outside the VM just before time (2) in Fig. 1. Once the log is outside the target VM, adversaries inside that VM cannot tamper with it. Therefore, the proposed method can avoid the effects of log tampering attacks at times (2), (3), and (4) because logs have already been transferred outside the target VM. Even though an adversary tampers with the log inside the target VM at times (2), (3), and (4), there is no effect on the evacuated logs on the logging VM.

5.3 Log Tampering Experiment

To test whether the proposed method can prevent log tampering or not, we tried to tamper with logs. First, we used *adore-ng* [3], a kernel-level malware that tampers with logs sent to the syslog daemon, to check if the proposed system can prevent log tampering in kernel space. *Adore-ng* tampers with logs by patching the runtime kernel code within memory. *Adore-ng* monitors interprocess communications using a socket function and deletes messages, if it contains words that would be disadvantageous if seen by the administrator of the computer. This experiment proves that the proposed method can prevent log tampering by kernel-level malware. Logs sent to the VMM with the proposed method were not tampered with while logs stored in the target VM were tampered with. Moreover, we can detect log tampering by comparing logs of the target VM with those of the logging VM. With this comparison, we can ascertain the purpose of the adversary.

Second, we tampered with a policy file of the syslog daemon as no logs are written to files. The policy file is loaded by the syslog daemon at start-up. With this attack, no logs are written, even if a syslog daemon is running. In this situation, we confirmed that the proposed system collects logs with no modification or loss. This result shows that the proposed system is resistant to attack on the policy file of the syslog daemon. The result also shows that log tampering by replacing a syslog daemon has no effect on the log collected by the proposed system. Thus, the proposed system is resistant to attacks such as *tuxkit* [2].

Third, we stopped a syslog daemon on a target VM to prevent logging. Obviously, no logs are transferred to the syslog daemon. We also confirmed that the proposed system can collect logs completely. However, this completeness depends on the log transfer flow. In GNU `libc`, a syslog function aborts log transfer when the establishment of a connection fails. Our prototype used for evaluation requests log transfer before establishing a connection to the syslog daemon; consequently, we can collect logs completely. This implies that logs might be lost if the library requests log transfer after establishing a connection.

Finally, we tampered with a log file in the target VM. This type of attack is used in the *LastDoor* backdoor [4]. It wipes specific entries in log files. Because the logs written to the file are already transferred to the logging VM, while logs in the target VM are tampered with, there is no effect on the log file in the logging VM.

These results show that the proposed system can collect almost all logs and that collected logs are not affected by attacks on the target VM. Additionally, adversaries tend to install log tampering malware at locations that all logs must pass through. For example, *adore-ng* [3] is installed to a kernel function and *tuxkit* [2] is installed to a syslog daemon. All logs sent by the syslog library function pass through the kernel function and the syslog daemon. Consequently, we can estimate that log tampering attacks on an AP, a source of logs, is rare.

5.4 Protecting Modified Library

5.4.1 Protection Method

There are two attacks on the proposed method that can be considered: replacing or modifying the replaced library file and manipulation of the memory area where the modified library is loaded.

Verification of the library file is effective for preventing attacks on the modified library file. We can detect modification of the library file by verifying the library file when a process loads the library to a memory. To verify the library file, we must preserve the signature of the file beforehand. When the file is loaded into the memory, the VMM detects the relevant load system call and compares the signatures of the original file and the file currently being loaded. From this, it can detect modification of the library file.

Using a hypervisor-based software runtime memory protection mechanism [13] is effective for preventing attacks on the memory area that includes the modified library. With it, any software, including a kernel, cannot manipulate the memory area of a designated process because the hypervisor mechanism isolates the area from other areas. The mechanism prepares another set of page tables to partition the virtual address space represented by the shadow page tables to protect the runtime memory against malware with root privileges. Linear address spaces of the protected APs and other software are separated from each other. Even if malware has root privileges, it cannot access the memory area of the protected APs. By using the mechanism to protect the modified library, the proposed method prevents adversaries from being able to modify that library.

5.4.2 Performance Estimation

Performance can be degraded during protection. Especially, the verification of the modified library requires substantial processing time because it checks file integrity. On the other hand, the protection of the memory area creates a slight overhead. Even if the memory area, which includes the modified library, is protected from write access, libraries are mapped to the memory as readable and executable. The protection overhead arises only when an illegal write to the area occurs. Therefore, the overhead that arises in the protection of the memory area is negligible.

Because the overhead created by the protection of the modified file dominates, estimation of that overhead is important for the estimation of the total overhead of the protection methods. To estimate the overhead, we created a prototype of the file integrity checker. The prototype runs on the logging VM and verifies the md5 hash of the modified library file. To create the md5 hash from the modified library, we used the `mhash` library. To read the modified library inside the target VM, we used the `guestmount` command. We estimated the performance overhead by measuring the time for creating the hash value with the modified library file inside the target VM. We create the md5 hash of `libc-2.13.so` and note its file size is 1,595,408 bytes. The environment used for the experiment is the same as that in the performance evaluation of the entire proposed system. The environment is shown in **Table 2**.

The results of the measurement show that creation of the hash value of the modified library file requires about 48 ms. The processing time for hash creation is measurable, because it requires disk access. However, its effect on the performance of resident programs is sufficiently small, because the overhead only arises during program start-up.

5.5 DoS Attack Using Proposed Method

Adversaries can use the proposed method as a tool for DoS attack. The proposed system has no authentication mechanism; therefore, a malicious process can send a massive number of log transfer requests. If a massive number of requests are sent to the VMM, the performance of the VMM and the logging VM degrades, and in turn, degrading the performance of each VM running on the VMM. If massive logs are transferred by the proposed method, the processing time of the VMM increases because the log transferring requires a lot of memory copies. Since all logs associated with the proposed method go through the logging VM, occupying of the processing of the logging VM by massive log transfer degrades the performance of the logging VM. Further, a mutual exclusion of the buffer inside the VMM degrades performance of the VMM. This situation leads to performance degradation of each VM on the VMM.

The following methods can address this problem:

(1) Authentication of a process that intends to request a log transfer.
(2) Ignoring log transfer requests from a process that is known to send a massive number of or large size requests.

Method (1) can prevent a log transfer request from an unauthenticated process. This requires implementing an additional interface for authentication. In method (2), we identify the malicious process by the frequency of its requests. If a specific process sends a massive number of requests in a short period, we classify the process as malicious. The issue with Method (2) is determining the criteria: the threshold number of requests and the time period.

After realizing that a process is malicious, the VMM treats the process in one of the following ways:

(1) Termination of the malicious process.
(2) Rejection of any log transfer request from the process.

Termination of the malicious process is a direct way to prevent attacks on the VMM. However, a legitimate process could stop running, if it is judged wrongly as malicious. The second method is more reasonable. If the frequency of requests from a suspicious process is reduced, we can judge the process as legitimate and accept requests from that process again. Even though the second method makes it easy to accept requests again, the method must manage information about all processes running on each VM. However, the cost of such management is worth considering. For these reasons, process authentication is our future work.

Table 2 Software used for evaluation.

VMM	Xen 4.2.0
OS (The logging VM) OS (The target VM)	Debian (Linux 3.5.0 64-bit) FreeBSD 9.0.0 64-bit Debian (Linux 2.6.32 64-bit)
Web server Database management system Syslog daemon	thttpd 2.25b PostgreSQL 9.2.4 rsyslogd 4.6.4
Benchmark	ApacheBench 2.3 pgbench 9.2.4 LMbench version 3

5.6 Applying the Proposed Method to Various Events

The proposed method focuses only on syslog events, but the method can be applied to other events, because it is based on requests from APs on a VM. For example, the Apache web server stores its logs using its own interface. By modifying the server to execute the `cpuid` instruction in its logging interface, the VMM can detect that request and collect logs. Because logging in a web server is one of the main reasons for performance degradation, the proposed system must be efficient. However, applying the proposed method to various events raises maintenance costs. Thus, it depends on the trade-off between adaptability and maintenance cost.

6. Evaluation

6.1 Purpose and Environment

We evaluated the proposed system from the following standpoints:

- Completeness of log collection
  We tested the system in a high-load environment by sending a massive number of log transfer requests from an AP in the target VM.
- Effort to adapt to various OSes
  Ease of adaptation to various OSes was also evaluated.
- Performance evaluation
  Performance overhead in syslog and database management system (DBMS) are evaluated.
- Performance in multi-VM environment
  We measured the performance of a web server with many VMs to clarify the performance overhead incurred by the proposed system in a multi-VM environment.
- Memory footprint
  We evaluated the increase in the memory footprint with the proposed system.

Software used for evaluation is described in Table 2. We implemented a prototype of the proposed system with the Xen [26]

hypervisor.

6.2 Completeness of Log Collection

To ensure that the proposed system can collect all logs in the target VM with no loss, we tested the proposed system in a high-load environment. In an experiment, we sent a log transfer request 10,000 times within approximately 0.26 seconds. The length of the log in each request was approximately 30 bytes. All logs were successfully transferred to the logging VM. No logs were incomplete or lost. This result shows that our proposed method is sufficient in terms of completeness of log collection in a high-load environment.

6.3 Effort to Adapt to Various OSes

In the prototype, we implemented the proposed method with FreeBSD and Linux as the target VMs and Xen as a VMM. To adapt to various OSes, modification to the target VM must be minimal. We added 20 additional lines of codes to `libc` on FreeBSD and Linux. **Figure 6** shows the result from the `diff` command. As shown in Fig. 6, we can adapt the proposed system to the `libc` library by inserting the `cpuid_logxfer()` function before the invocation of a send system call. The rest of the additional code is just the definition of the `regs` structure and the `cpuid_logxfer()` function. **Figure 7** shows the definition of the function. The function consists of (1) setting the registers with the appropriate values and (2) executing the `cpuid` instruction. Based on the size of the additional code, adapting the proposed system to various OSes would be a small effort.

6.4 Dependency on OS Structure

As described in Section 6.3, the proposed method depends on the `libc` library. Though the proposed method does not depend on a structure within the OS, each OS has its own library. This section provides a quantitative comparison between the proposed method and conventional methods stated in Section 2.1.

```
void
  __vsyslog_chk(int pri, int flag, const char *fmt, va_list ap)
  {
           int saved_errno = errno;
           char failbuf[3 * sizeof (pid_t) + sizeof "out of memory []"];

+          reg_t regs;
+          regs.rax = 0xffff;
+
  #define   INTERNALLOG LOG_ERR|LOG_CONS|LOG_PERROR|LOG_PID
           /* Check for invalid bits. */
           if (pri & ~(LOG_PRIMASK|LOG_FACMASK)) {
***************
*** 278,283 ****
--- 297,308 ----
           if (LogType == SOCK_STREAM)
             ++bufsize;

+          regs.rbx = (unsigned long)buf;
+          regs.rcx = bufsize;
+
+          cpuid_logxfer(regs.rax, &regs);
+          regs.rax = regs.rbx = regs.rcx = 0;
+
           if (!connected || __send(LogFile, buf, bufsize, send_flags) < 0)
             {
               if (connected)
```

Fig. 6 The result of diff command between source codes of unmodified library and modified library.

```
void cpuid_logxfer(unsigned long rax, reg_t *reg)
{
       __asm__ __volatile__(
               "cpuid"
               :"=a"(reg->rax)
               :"a"(rax), "b"(reg->rbx), "c"(reg->rcx)
       );
       return;
}
```

Fig. 7 Definition of cpuid_logxfer() fucntion.

First, we analyzed the total amount of codes of the proposed method that depends on `libc`. Despite the additional lines in the `libc` on Debian being 20, almost all codes are the definition of `cpuid_logxfer()` function and the `regs` structure. Essential dependent codes are the setting of the registers with the appropriate value and execution of the `cpuid` instruction. Insertion of them into the syslog function is only required for the proposed method. Thus, the dependency on the implementation of the OS or library is slight.

Next, we researched the dependency on the OS structure of the existing methods as stated in Section 2.1. VMI [11] and CloudSec [21] are strongly dependant on the structure of the OS because it requires memory analysis according to definitions of data structure used in the kernel. If the version of the kernel is updated, corrections to the definitions may be required. Thus, these methods strongly depend on OS structure. In-VM monitoring method [14] has less dependency than the methods above because it inserts agents into the kernel on a VM. The agent is created along the procedure, which is introduced by the vendor of the kernel. The VMM-based log collection scheme [22] requires the address of the entry point to the kernel on a VM to hook system calls onto that VM. Further, the area of a kernel log buffer must be known to the VMM. The VMM must distinguish and adapt to each OS. While no modification to OSes and APs is required, it depends on the data structure of the OS.

The dependency on the structure of the kernel of a OS obviously makes the application of those methods difficult. In general, modification of kernels or device drivers is more difficult than that of APs. Thus, application of the existing methods is difficult. On the other hand, the proposed methods only requires library modification. Further, if modified libraries are already compiled and prepared for delivery, the administrator of the computer can apply the proposed method by just replacing the library file. The cost of applying the proposed method is clearly less than that of existing methods.

6.5 Performance Evaluation

6.5.1 Measured Items and Environment

We measured the performance of the syslog function, some system calls, and an AP. We also measured the performance overhead of an AP in a multi-VM environment. The performance measurements of both the syslog function and the APs show the additional overhead incurred by the proposed system. The performance measurement of some system calls shows that the proposed system causes additional overhead only when the syslog function is called.

We measured the performance using a computer, with a Core i7-2600 (3.40 GHz, 4-cores) and 16 GB memory. For each measurement, one virtual CPU (VCPU) is provided and 1 GB memory is allocated to each VM. Hyper-threading is disabled. Each VCPU is pinned to a physical CPU core to avoid measurement instability. If many VMs worked on one physical CPU, the performance of APs on those VMs would be unstable.

6.5.2 Syslog Function and System Call

In the proposed system, the modified library requests a log transfer when an AP calls the syslog function. To clarify the overhead incurred by the proposed system, we measured and compared the performance of the syslog function with an unmodified Xen and the proposed system. **Table 3** compares the performance of the syslog function between Xen and the proposed system. In the proposed system, the additional overhead of the syslog function is 1.91 μs (6.08%), which is small, because the function is not called frequently.

Table 4 shows the function call counts in thttpd when accessed 100 times by ApacheBench. We measured the library function call counts using `ltrace`. Table 4 shows that the percentage of syslog function calls in thttpd to be about 1%. Additionally, we measured the performance impact of the library functions in thttpd with the same workload. **Table 5** shows the measurement result. These results were obtained using Ubuntu 13.04. The function `__syslog_chk` is the same as `syslog`. As shown in Table 5, the performance impact of the syslog function is only 0.18%. As such, the 6.08% overhead incurred by the syslog function has limited impact on the performance of APs.

Additionally, we measured the performance of some system calls using LMbench, which measures the performance of file creation and deletion, process creation, system call overhead, and other processes. According to this measurement, the additional overhead is not significant.

Table 3 Performance comparison of the syslog function.

	Time (μs)	Overhead (μs (%))
Xen	31.47	–
Proposed system	33.38	1.91 (6.08%)

Table 4 Frequency of library function calls when serving web page request with thttpd web server.

Function name	Count	Rate (%)	Function name	Count	Rate (%)
strncasecmp	1,600	17.77	strftime	200	2.22
strlen	1,400	15.55	accept	200	2.22
strcpy	800	8.89	gmtime	200	2.22
vsnprintf	600	6.67	__errno_location	200	2.22
memmove	400	4.44	time	100	1.11
strchr	400	4.44	close	100	1.11
select	301	3.34	read	100	1.11
gettimeofday	301	3.34	getnameinfo	100	1.11
strstr	300	3.33	strcat	100	1.11
fcntl	300	3.33	readlink	100	1.11
strpbrk	300	3.33	strrchr	100	1.11
strcasecmp	200	2.22	syslog	100	1.11
__xstat	200	2.22	writev	100	1.11
strspn	200	2.22			

Table 5 Performance impact of the library functions on thttpd.

Function name	Rate (%)	Function name	Rate (%)
writev	76.90	memmove	0.13
poll	17.71	gmtime	0.10
strncasecmp	0.82	strcasecmp	0.10
strlen	0.81	strftime	0.10
strcpy	0.51	strspn	0.09
close	0.30	strcat	0.09
__vsnprintf_chk	0.30	read	0.08
__xstat	0.23	getnameinfo	0.05
strchr	0.22	memcpy	0.05
fcntl	0.22	time	0.05
__syslog_chk	0.18	strrchr	0.05
accept	0.17	__strcpy_chk	0.04
readlink	0.15	malloc	0.02
strpbrk	0.14	mmap	0.00
strstr	0.14	open	0.00
__errno_location	0.14	realloc	0.00
gettimeofday	0.14		

6.5.3 Comparison with Log Transfer using rsyslogd

To clarify the latency of log transfer using rsyslogd, we measured the latency in the proposed system. Because rsyslogd can transfer logs in UDP and TCP, we measured the latency of the log transfer with rsyslogd in UDP, rsyslogd in TCP, and the proposed system. To measure the performance overhead during log transfer, we also measured the latency of log storing with local rsyslogd. We modified a rsyslogd policy so that a fine-grained time stamp is appended to each log entry. We sent a log message 1,000 times. The length of each log entry is nearly 300 bytes, approximately the same as a regular access log in the Apache web server. All measurements are performed on a VM. In a log transfer with rsyslogd, a log generated on a VM is transferred to another VM.

Table 6 shows the measurement results. From these results, we see that log transfer with the proposed system is faster than with rsyslogd. Nevertheless, log transfer with rsyslogd in UDP is faster than TCP, and slower than with the proposed system. Assuming a high-load environment, a fast log transfer mechanism is preferable. With the proposed system, aggregating logs from VMs is achieved with low overhead.

6.5.4 Database Management System

We measured the performance overhead using the proposed system in a DBMS using PostgreSQL as the DBMS. We configured PostgreSQL to call the syslog function for each transaction. We used pgbench to measure PostgreSQL's performance. The workload with pgbench includes five commands per transaction. The benchmark measures the transactions per second (TPS) of a DBMS. The concurrency of transactions is set to one.

Table 7 shows a comparison in performance of the PostgreSQL DBMS. Higher TPS is better. Performance degradation with the proposed method is less than 1%. The proposed method degrades performance of a CPU-intensive process. Because PostgreSQL accesses the disk heavily, the overhead incurred with the proposed method becomes small. To verify that the proposed system is CPU intensive, we measured the performance with `tmpfs`, which provides a memory-based file system. Since transactions do not require access to a disk, performance overhead with the proposed method would be expected to be higher. Table 7 shows

Table 6 Latency of log transfer in rsyslogd with UDP, rsyslogd with TCP, and the proposed system.

	Time after 1,000 times of log transfer (ms)	Difference (ms)
Local rsyslogd	29.98	—
rsyslogd in UDP	57.13	27.16
rsyslogd in TCP	71.12	41.14
Proposed System	37.76	7.78

Table 7 Performance comparison of PostgreSQL.

tmpfs	VMM	TPS	Relative performance
disabled	Xen	400.37	–
	Proposed system	395.76	0.99
enabled	Xen	1,448.80	–
	Proposed system	1,372.60	0.95

Table 8 Throughputs of web server (requests/s) in multi-VM environment.

File size	VMM	Number of VM						
		0	2	4	6	8	10	12
1 KB	Xen	1396.9	1329.27	1295.61	1225.22	1171.51	1231.72	1172.15
	Proposed system	1231.06	1150.54	1057.95	1017.53	987.24	1015.69	946.61
	Relative performance	0.88	0.87	0.81	0.83	0.84	0.82	0.8
10 KB	Xen	680.61	658.15	639.76	627.9	628.56	609.45	615.64
	Proposed system	664.48	626.12	612.93	559.02	582.24	578.89	589.58
	Relative performance	0.98	0.95	0.89	0.92	0.93	1.00	0.96
1,000 KB	Xen	11.41	11.41	11.4	11.39	11.38	11.39	11.39
	Proposed system	11.41	11.41	11.4	11.39	11.37	11.39	11.06
	Relative performance	1.00	1.00	1.00	1.00	1.00	1.00	0.98

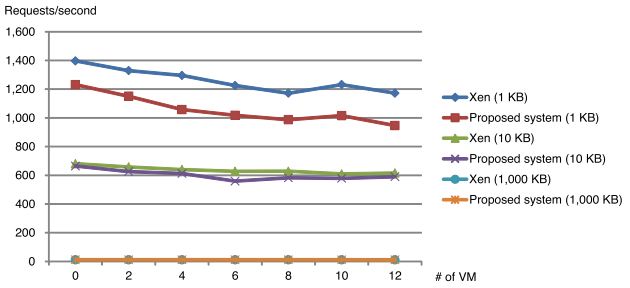

Fig. 8 Performance comparison of web server in a multi-VM environment. Horizontal axis shows the number of other VMs. Vertical axis shows throughput of a web server in requests/s; higher measurements are better.

that the relative performance of an unmodified Xen with tmpfs is about 5%. The performance degradation is greater than in the case without tmpfs. If processing is I/O intensive, performance degradation with the proposed method decreases. Thus, the proposed method is suitable for I/O-intensive APs. In this measurement, we configured PostgreSQL to call syslog for each transaction, but logging frequency in general DBMS use decreases. Thus, performance degradation can be assumed to be almost negligible under normal use.

6.5.5 Web Server in a Multi-VM Environment

To examine the ability of our proposal to scale to its target of many domains, we measured the performance of a web server in a VM with many other VMs. These VMs have a process that sends logs using the syslog function every second. This evaluation is performed on a machine that has four CPU cores; the logging VM is placed on core 0, a VM that has a web server is placed on core 1, and other VMs are placed on cores 2 and 3 to measure the pure performance changes of the web server. We placed 2, 4, 6, 8, 10, and 12 VMs on cores 2 and 3. The number of VMs on cores 2 and 3 is the same. The same scheduling priority is configured for each VM. The performance is measured using ApacheBench on a remote machine with a 1 Gbps network.

Table 8 shows the performance in each environment. **Figure 8** shows changes in performance for each environment. If the number of VMs increases, the performance of the web server degrades. Performance degradation with the proposed system is less than approximately 10% when the file size is larger than 10 KB. Especially, when the file size is 1,000 KB, performance degradation is negligible. From these results, we can estimate that the change in relative performance related to the number of VMs is sufficiently small. Despite the change in number of VMs, the change in relative performance is approximately the same. Consequently, the proposed system is efficient in a multi-VM environment.

Table 9 Comparison of memory footprint of VMM (MB).

	Xen	Proposed System
Total memory	16,288	16,288
Free memory	15,082	15,082
Memory for Domain-0	1,024	1,024
Memory footprint of VMM	182	182

6.5.6 Memory Footprint

To evaluate the memory efficiency of the proposed system, we calculate the memory footprint of the unmodified Xen and the proposed system. It is difficult to calculate a memory footprint directly; the total memory recognized by Xen, the free memory recognized by Xen, and the memory allocated to Domain-0 are used for the estimation. The total memory and the free memory recognized by Xen can be determined using the `xl info` command. The total memory allocated to Domain-0 is determined using `xl list`, which shows the name, ID, total amounts of memory, allocated VCPUs, state, and consumed time of each VM. From this information, we estimated the memory footprint of Xen and the proposed system.

Table 9 shows calculated results. The memory footprint of VMM is calculated using Total memory) - (Free memory) - (Memory for Domain-0). An increase in the memory footprint of VMM cannot be observed in the results. Considering the granularity of the calculation, the increase in the memory footprint is less than 1 MB. Thus, the proposed system is efficient in terms of memory usage.

Furthermore, we compared the size of the executable file of the VMM. The size of unmodified Xen is 805,133 bytes, while that of modified Xen, the proposed system, is 806,517 bytes. According to this comparison, the modified Xen is 1,384 bytes larger than the unmodified Xen.

7. Related Works

7.1 Secure Logging

Accorsi classified and analyzed secure logging protocols [27]. In that work, extensions of syslog, including syslog-ng [17], syslog-sign [18], and reliable syslog [19] are distinguished as protocols that provides security in the transmission of log messages, not in the storage phase. We focus on the transmission phase, because our proposal is highly related to that phase. Accorsi asserted that only reliable syslog fulfills security requirements that guarantee the authenticity of audit trails. Even if those protocols

can detect and verify a log message as having not been tampered with, they cannot prevent deletion of or tampering with logs. At this point, those protocols differ from our proposal. Therefore, this paper proposes protection of log messages from the viewpoint of system security. By combining our proposal and the existing secure logging protocols, we can increase the security of logged data.

7.2 VM Logging

ReVirt [20] logs non-deterministic events on a VM for replay. Because the method logs events for the analysis of attacks, it involves types of data different from those of our proposal. While ReVirt logs instruction-level information, our proposal collects log messages for syslog. With our proposal, we can easily monitor the target VM without deep analysis of logged information because those logs are already formatted.

A VM has also been used to separate logged information [28]. While that study separates information about file system logs, our proposal separates logs for syslog. Since that study used a split device driver model of Xen provided for para-virtualization, the proposal can be applied only in a para-virtualized environment. Our prototype is implemented in a fully virtualized environment; however, implementing our proposed method in a para-virtualized environment is easy.

VMI [11] and other introspection methods [14] can be considered as logging methods with a VM. In that regard, these methods are similar to ours. However, information gathered using those methods is formatted differently from syslog; therefore, existing tools are not applicable to the analysis of these data. In contrast, with our proposal, existing tools work well without modification, because the format of the information gathered by our proposal is the same as that of messages produced by syslog. One VMM-based log-tampering and loss detection scheme [22] can gather information from a VM without modification to a library in that VM. Their system [22] collects logs using the log collection method desribed previously [29]. Even though modification to a library is unnecessary with that log collection method, it requires modification of a VMM to adapt to various OSes. Modification of a VMM requires restarting all VMs on that VMM. In addition, it creates measurable overheads. In contrast, although our proposal requires modification of a library on a VM, it requires no modification of the VMM to adapt to various OSes and has less overhead. Moreover, less effort is required to adapt the proposed method to various OSes.

8. Conclusions

Our secure log transfer method that replaces a library in a VM provides processes on a VM with the ability to transfer logs without involving the VM kernel. Thus, even though kernel-level malware tampers with logs on that VM, logs gathered by our proposed method have no effect. In addition, we implemented the proposed system using a VMM, so attacking the proposed system from a target VM is difficult because of the properties of the VMM. Moreover, adapting the method to various OSes is easy, because of its implementation with library modifications. Furthermore, by modifying a library in a VM, the proposed method reduces log transfer performance overhead. Modifying the library on a VM to request a log transfer to a VMM, instead of monitoring the behavior of each VM, reduces unnecessary overhead related to monitoring and achieves fast log transfers.

Evaluation of its resistance to log tampering shows that tampering with logs from the target VM is difficult. The experiment of adapting to different OSes showed that only 20 lines of code need to be added to the `libc` library. Performance evaluation shows that performance degradation of the syslog function is only about 6%. Performance degradation is negligible, if the processing of the AP is I/O intensive. Performance evaluation in a multi-VM environment shows that the proposed system has sufficient performance with multiple VMs.

We believe that the prototype is sufficient for claiming its effectiveness of our proposed method, even though protection of the modified library is not yet implemented. For practical use, implementation and evaluation of the protection mechanism are required. We consider them as our future work.

Acknowledgments This work was supported by Grant-in-Aid for JSPS Fellows.

References

[1] Kent, K. and Souppaya, M.: Guide to Computer Security Log Management, Special Publication 800-92 (2006).
[2] spoonfork: Analysis of a rootkit: Tuxkit, available from ⟨http://www.ossec.net/doc/rootcheck/analysis-tuxkit.html⟩ (accessed 2013-11-15).
[3] stealth: A new Adore root kit, available from ⟨http://lwn.net/Articles/75990/⟩ (accessed 2013-11-15).
[4] Symantec: Backdoor.lastdoor, available from ⟨http://www.symantec.com/security_response/writeup.jsp?docid=2002-090517-3251-99⟩ (accessed 2013-11-12).
[5] Subashini, S. and Kavitha, V.: A survey on security issues in service delivery models of cloud computing, *Journal of Network and Computer Applications*, Vol.34, No.1, pp.1–11 (2011).
[6] Grobauer, B., Walloschek, T. and Stocker, E.: Understanding Cloud Computing Vulnerabilities, *IEEE Security & Privacy*, Vol.9, No.2, pp.50–57 (2011).
[7] Marty, R.: Cloud application logging for forensics, *Proc. 2011 ACM Symposium on Applied Computing, SAC '11*, pp.178–184 (2011).
[8] Chen, P.M. and Noble, B.D.: When virtual is better than real, *Proc. 8th Workshop on Hot Topics in Operating Systems, HOTOS '01*, pp.133–138 (2001).
[9] Boeck, B., Huemer, D. and Tjoa, A.M.: Towards More Trustable Log Files for Digital Forensics by Means of "Trusted Computing", *Proc. 2010 24th IEEE International Conference on Advanced Information Networking and Applications*, pp.1020–1027 (2010).
[10] Seshadri, A., Luk, M., Qu, N. and Perrig, A.: SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes, *SIGOPS Oper. Syst. Rev.*, Vol.41, Issue 6, pp.335–350 (2007).
[11] Garfinkel, T. and Rosenblum, M.: A Virtual Machine Introspection Based Architecture for Intrusion Detection, *Proc. Network and Distributed Systems Security Symposium*, pp.191–206 (2003).
[12] Hu, Y., Nanda, A. and Yang, Q.: Measurement, analysis and performance improvement of the Apache Web server, *1999 IEEE International Performance, Computing and Communications Conference*, pp.261–267 (1999).
[13] Dewan, P., Durham, D., Khosravi, H., Long, M. and Nagabhushan, G.: A hypervisor-based system for protecting software runtime memory and persistent storage, *Proc. 2008 Spring Simulation Multiconference, SpringSim '08*, pp.828–835 (2008).
[14] Sharif, M.I., Lee, W., Cui, W. and Lanzi, A.: Secure in-VM Monitoring using Hardware Virtualization, *Proc. 16th ACM Conference on Computer and Communications Security, CCS '09*, pp.477–487 (2009).
[15] Sato, M. and Yamauchi, T.: Secure Log Transfer by Replacing a Library in a Virtual Machine, *Advances in Information and Computer Security*, Lecture Notes in Computer Science, Vol.8231, pp.1–18 (2013).
[16] Adiscon: Adiscon's rsyslog: The enhanced syslogd for Linux and Unix rsyslog, available from ⟨http://www.rsyslog.com/⟩ (accessed

2013-11-15).
[17] Balabit IT Security: The free software BalaBit: Syslog Server — syslog-ng Logging System, available from ⟨http://www.balabit.com/network-security/syslog-ng/⟩ (accessed 2013-11-15).
[18] Kelsey, J., Callas, J. and Clemm, A.: Signed syslog messages, available from ⟨http://tools.ietf.org/html/rfc5848⟩ (accessed 2013-11-12).
[19] New, D. and Rose, M.: Reliable delivery for syslog, available from ⟨http://www.ietf.org/rfc/rfc3195.txt⟩ (accessed 2013-11-12).
[20] Dunlap, G.W., King, S.T., Cinar, S., Basrai, M.A. and Chen, P.M.: ReVirt: Enabling intrusion analysis through virtual-machine logging and replay, *SIGOPS Oper. Syst. Rev.*, Vol.36, No.SI, pp.211–224 (2002).
[21] Ibrahim, A., Hamlyn-Harris, J., Grundy, J. and Almorsy, M.: CloudSec: A security monitoring appliance for Virtual Machines in the IaaS cloud model, *Proc. 2011 5th International Conference on Network and System Security* (*NSS*), pp.113–120 (2011).
[22] Sato, M. and Yamauchi, T.: VMM-Based Log-Tampering and Loss Detection Scheme, *Journal of Internet Technology*, Vol.13, No.4, pp.655–666 (2012).
[23] Wojtczuk, R.: Subverting the Xen Hypervisor (2008).
[24] Rutkowska, J. and Wojtczuk, R.: Preventing and Detecting Xen Hypervisor Subversions.
[25] Popek, G.J. and Goldberg, R.P.: Formal Requirements for Virtualizable Third Generation Architectures, *Commun. ACM*, Vol.17, No.7, pp.412–421 (1974).
[26] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I. and Warfield, A.: Xen and the art of virtualization, *SIGOPS Oper. Syst. Rev.*, Vol.37, Issue 5, pp.164–177 (2003).
[27] Accorsi, R.: Log data as digital evidence: What secure logging protocols have to offer?, *Proc. 2009 33rd Annual IEEE International Computer Software and Applications Conference*, Vol.2, pp.398–403 (2009).
[28] Zhao, S., Chen, K. and Zheng, W.: Secure logging for auditable file system using separate virtual machines, *Proc. 2009 IEEE International Symposium on Parallel and Distributed Processing with Applications*, pp.153–160 (2009).
[29] Sato, M. and Yamauchi, T.: VMBLS: Virtual Machine Based Logging Scheme for Prevention of Tampering and Loss, *Availability, Reliability and Security for Business, Enterprise and Health Information Systems*, Lecture Notes in Computer Science, Vol.6908, pp.176–190 (2011).

Masaya Sato received his B.E. and M.E. degrees from Okayama University, Japan in 2010, 2012, respectively. He has been a Research Fellow of the Japan Society for the Promotion of Science since 2013. He has been a doctoral student of Graduate School of Natural Science and Technology at Okayama University since 2012. His research interests include computer security and virtualization technology. He is a member of IPSJ.

Toshihiro Yamauchi received his B.E., M.E. and Ph.D. degrees in computer science from Kyushu University, Japan in 1998, 2000 and 2002, respectively. In 2001 he was a Research Fellow of the Japan Society for the Promotion of Science. In 2002 he became a Research Associate in Faculty of Information Science and Electrical Engineering at Kyushu University. He has been serving as associate professor of Graduate School of Natural Science and Technology at Okayama University since 2005. His research interests include operating systems and computer security. He is a member of IPSJ, IEICE, ACM and USENIX.