Improvement and Evaluation of a Function for Tracing the Diffusion of Classified Information on KVM

Hideaki Moriyama[1*], Toshihiro Yamauchi[2*], Masaya Sato[3], and Hideo Taniguchi[2]
[1]Department of Creative Engineering,
National Institute of Technology, Ariake College, Omuta, Fukuoka 836-8585, Japan
[2]Graduate School of Natural Science and Technology
Okayama University, Okayama, Okayama 700-8530, Japan
[3]Faculty of Computer Science and Systems Engineering
Okayama Prefectural University, Soja, Okayama 719-1197, Japan

Received: October 21, 2021; Accepted: January 29, 2022; Published: February 28, 2022

Abstract

The leakage of computerized classified information can cause serious losses for companies and individuals. In a prior work, we addressed this by providing a function for tracing the diffusion of classified information in a guest operating system (OS). However, that method was vulnerable to attack and was tightly coupled to the OS. Hence, in another previous work, we applied the tracing function using a virtual machine monitor that hooks into system calls that handle classified information, allowing us to understand the diffusion path in a more robust and OS-agnostic fashion. However, as the overhead of the tracing function increases, so does the performance degradation of each system call. Hence, in the current research, the processing performance of the tracing function is further analyzed in depth by identifying the processes that cause the large overhead. We find that the performance overhead generated by outputting the diffusion path log is too burdensome. Therefore, improvements are implemented, and the effectiveness of the upgraded performance is described. Ultimately, the log-output overhead problem is improved.

Keywords: information leak prevention, performance improvement, virtual machine monitor

1 Introduction

As the corpus of computerized information continues to increase, so does the need to handle classified information on information systems. Notably, the leakage of computerized classified information can cause serious losses to companies and individuals. Such leakages often occur inadvertently and through mismanagement. To prevent this, it is important for users and managers to understand the risks associated with storing and handling classified information. Furthermore, cyberattacks tend to aim directly at the theft of said information, and the related methods have become increasingly sophisticated. It is therefore crucial to minimize the damage caused by leaks by immediately detecting the unauthorized transfer of classified information.

To trace the status of classified information stored in an information system and to manage the corresponding resources, we previously proposed an operating system (OS)-based function for tracing the diffusion of classified information across multiple computers [1, 2]. We also proposed a function that

*Corresponding authors: [1]National Institute of Technology, Ariake College, 150 Higashihagio-Machi, Omuta Fukuoka 836-8585, Japan, Tel: +81-(0)944-53-8732, Email: `hideaki@ariake-nct.ac.jp`; [2]Okayama University, 3-1-1 Tsushima-naka, Kita-ku, Okayama 700-8530, Japan, Email: `yamauchi@cs.okayama-u.ac.jp`

visualized this diffusion using a directed graph [3]. The developed functions efficiently identified the manipulation of classified information to help prevent leakage.

Notably, the original OS-based tracing function could be disabled by an attack on the OS. When disabled, the tracing function obviously fails to help the victim detect an information leak. Moreover, any time the OS kernel version was updated, the original tracing function had to be patched. Furthermore, the function could not be introduced to a closed-source OS (e.g., Windows) because doing so required modification of the source code.

To resolve these problems, in a consecutive work, we ported the tracing function to a virtual machine monitor (VMM) [4, 5], which allowed it to be implemented without modifying the OS. Furthermore, because a VMM is more robust than an OS, it is unlikely that attacks can specifically target this function. Later, we analyzed the processing performance of the VMM-based tracing function in detail and identified large-overhead processes, finding that it was difficult to immediately grasp the diffusion path because the tracing function was required to output the path information to a log at every iteration. Therefore, the objective of the current research is to improve the VMM-based tracing function and to report on its evaluation.

The rest of this study is organized as follows. Section 2 presents an overview of the VMM-based tracing function proposed in [4, 5]. We present the performance analysis results for the VMM-based tracing function in Section 3 and the problem of grasping the potential diffusion of classified information is presented in Section 4. We describe the improvement in Section 5, and Section 6 presents the evaluation results. We discuss the related studies in Section 7 and conclude with Section 8.

2 Function for Tracing the Diffusion of Classified Information in a Guest OS

2.1 Overview of the VMM-based Tracing Function

Here, we explain the VMM-based tracing function proposed in [4, 5], which manages any file or process that has the potential to diffuse classified information, which involves opening a classified file, reading its content, communicating with another process, or writing the content to another file. In short, the diffusion of classified information is caused by the following operations:

1. File operation
2. Inter-process communication
3. Child process creation

A user can always obtain the location of classified information using the list stored in the proposed VMM. Furthermore, when classified information diffusion is detected, the VMM-based tracing function records the pathname of the destination file, the inode number, the command name that caused the diffusion, and the process ID (PID). Therefore, a user can detect information leaks using the above information and suppress damage, even if some classified information is compromised.

Figure 1 shows an overview of the VMM-based tracing function, which uses a kernel-based virtual machine (KVM) and a 64-bit Linux OS with a 3.6.10 kernel as the VMM and the guest OS, respectively [4, 5]. The VMM-based tracing function traces the diffusion of the classified information using the following process.

1. A user program in the guest OS requests a system call.

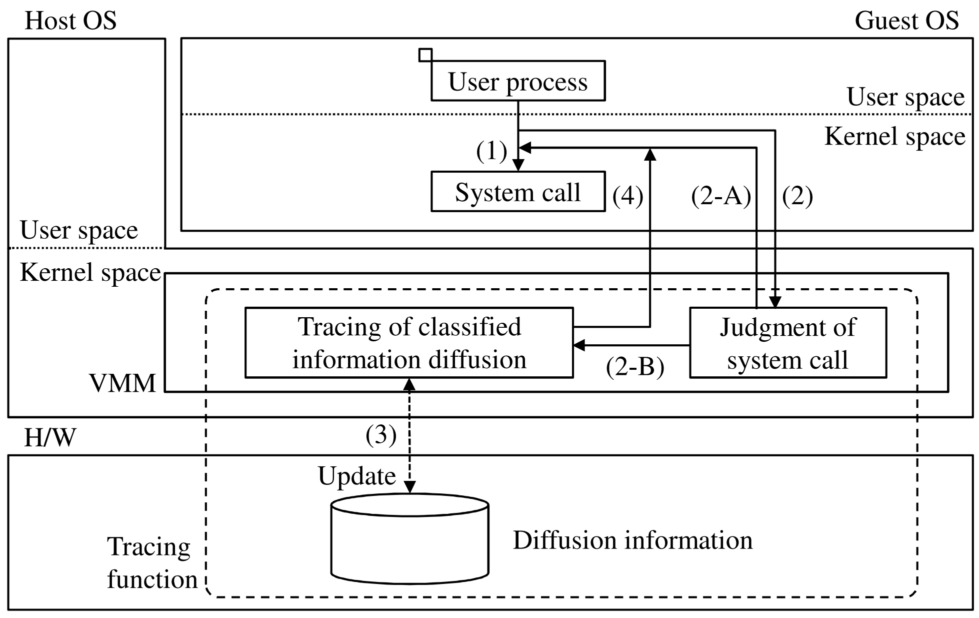


Figure 1: Overview of the tracing function

2. The VMM-based tracing function hooks the system call in the guest OS from the VMM. After identifying the hooked system call, one of the following system-call processing steps is implemented:
   (a) If the hooked system call is unrelated to the diffusion of classified information, control is returned to the guest OS, and the system-call process is continued.
   (b) If the hooked system call is related to the diffusion of classified information, the VMM-based tracing function collects the information needed to trace the diffusion.
3. If the classified information is diffused, the VMM-based tracing function updates the diffusion information using the information collected in Step 2-(b).
4. Control is returned to the guest OS, and the system-call process is continued.

Considering the above steps, the VMM-based tracing function provides the guest OS with capabilities that are equivalent to those of the OS-based tracing function without the need to modify the OS source code.

2.2 Collecting System-call Information with VMM

When a user program in the guest OS requests a system call, the VMM-based tracing function hooks the call using the hardware breakpoint and collects the relevant information. Figure 2 shows the construction of system-call hooking process using the hardware breakpoint on the KVM. When a user program in the guest OS requests a system call, the VMM-based tracing function hooks the system-call entry via the SYSCALL instruction so that the system-call request can be detected. To hook the SYSCALL instruction, the VMM-based tracing function sets the breakpoint address register to the SYSCALL address. The register then specifies the breakpoint address, with a debug exception being generated when memory access is implemented for the address. Thus, a debug exception occurs upon executing the SYSCALL

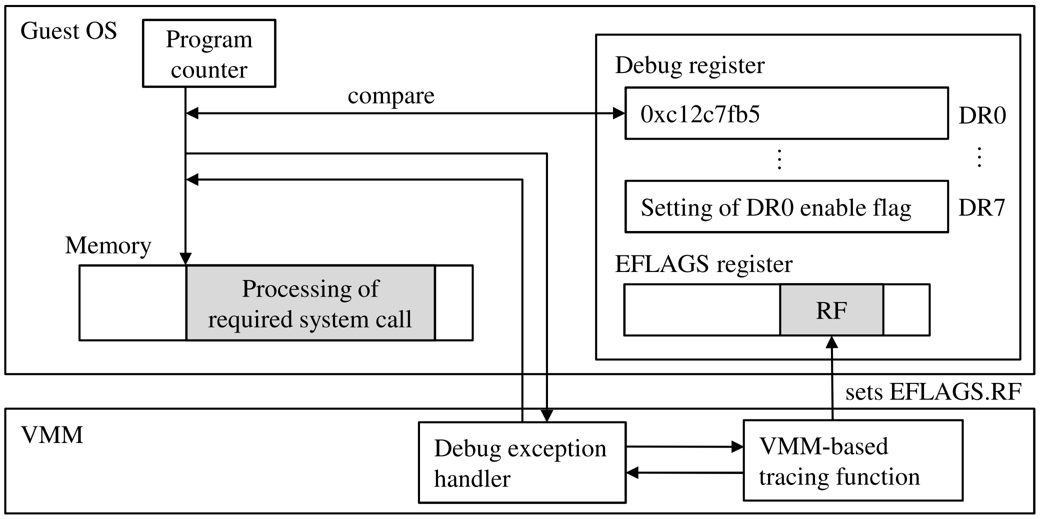


Figure 2: Construction of system-call hooking process using hardware breakpoint

instruction, and the VMM-based tracing function hooks the SYSCALL instruction with the VMM by detecting the debug exceptions in the guest OS.

Considering each system call, information concerning the success or failure of the call is returned, and the details of the file handled by the system call is the return value. It is necessary to collect these file details to enable the VMM-based tracing function to trace the diffusion of classified information. Thus, the VMM-based tracing function hooks the system-call exit via the SYSRET instruction making it possible to obtain the return value corresponding to the system call. To hook the SYSRET instruction, the VMM-based tracing function sets the breakpoint address register to the SYSRET address. Like the SYSCALL case above, a debug exception occurs upon execution of the SYSRET instruction. Therefore, the VMM-based tracing function hooks the SYSRET instruction with the VMM by detecting debug exceptions in the guest OS.

3 Overhead Analysis

Figure 3 shows the process flow of the VMM-based tracing function, which consists of two sub-flows. When a user program in the guest OS requests a system call, an exception occurs upon SYSCALL or SYSRET execution. Thereafter, the VMM-based tracing function judges whether the exception is a debug exception. If so, the VMM-based tracing function determines whether the instruction at the breakpoint address register is a SYSCALL or a SYSRET instruction. To identify the hooked system call and determine whether it is related to the diffusion of the classified information, the VMM-based tracing function uses a system-call number. Thereafter, the VMM-based tracing function collects the information (e.g., page table and file descriptor) required for tracing the diffusion for each system call. Finally, the VMM-based tracing function updates the diffusion information, and control is returned to the guest OS.

To clarify the process flow of the VMM-based tracing function with a large overhead, we performed a brief evaluation. The VMM-based tracing function registers the potential leakage of classified processes and files. Therefore, to evaluate a processing task including these registrations, we measured the execution time of a `cp` command, which targeted a classified file. The `cp` command process includes a read system call, which involves registration of the potential leakage of a classified process and a write system

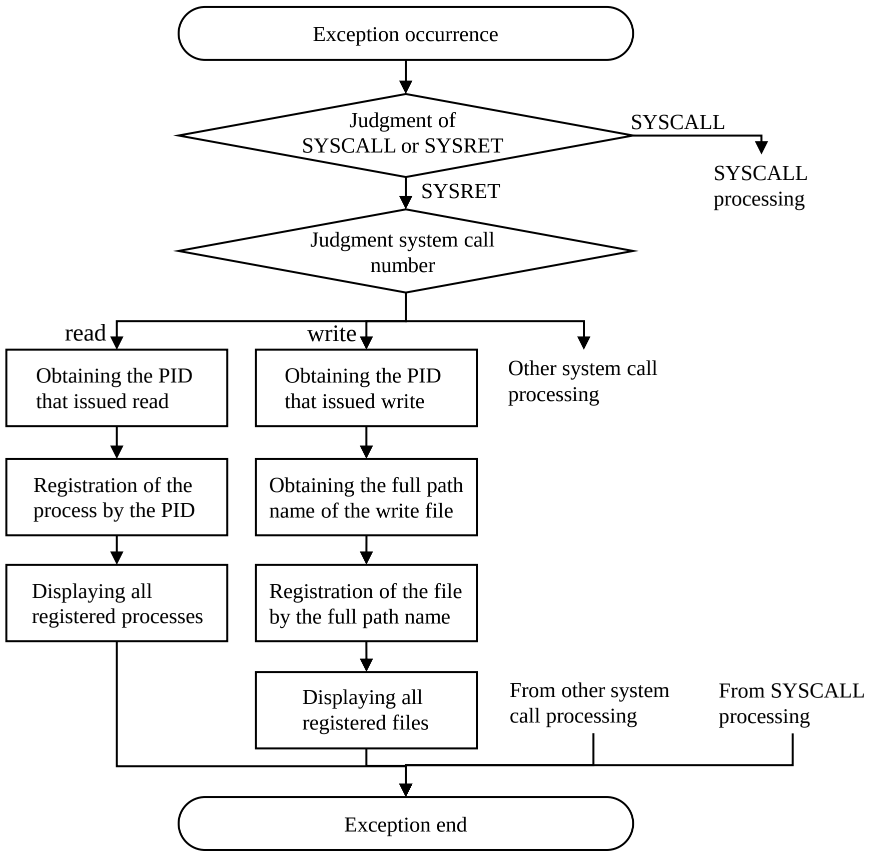

Exception occurrence
Judgment of SYSCALL or SYSRET
SYSCALL
SYSCALL processing
SYSRET
Judgment system call number
read
write
Obtaining the PID that issued read
Obtaining the PID that issued write
Other system call processing
Registration of the process by the PID
Obtaining the full path name of the write file
Displaying all registered processes
Registration of the file by the full path name
Displaying all registered files
From other system call processing
From SYSCALL processing
Exception end

Figure 3: Process flow of the tracing function

call. This includes the registration of the potential leakage of a classified file. Considering this evaluation, the VMM-based tracing function determined that classified information diffusion had occurred and executed the tracing process. We inserted “get timestamp” processes into the VMM-based tracing function and compared the execution times of the hooking the SYSCALL and SYSRET instructions. Table 1 lists the specifics of the evaluation environment. We evaluated the VMM-based tracing function using an Intel Xeon CPU E5-2609 (1.7 GHz, 8 CPUs) with a 64-GB memory. The guest OS was allocated one virtual CPU and 1-GB memory.

We found that the processing times of the VMM-based tracing function when hooking the SYSCALL and SYSRET instructions were 6.2 and 97.8 μs, respectively. The SYSRET hooking process was larger because it included determining whether the hooked system call was related to the diffusion of classified information. This process was implemented for every read and write system call, and it generated an increasingly large overhead for system-call execution. Hence, it was judged to be necessary to reduce this overhead as much as possible.

To reduce the large overhead during processing, we analyzed the hook process in detail prior to the SYSCALL instruction of the VMM-based tracing function and evaluated the execution times of all processing tasks comprising the VMM-based tracing function. The evaluation environment is similar to the one described in Table 1. Moreover, as in the previous evaluation, we measured the execution time

Table 1: Evaluation environment

Host machine	
CPU	Intel Xeon CPU E5-2609 (1.7 GHz)
Number of Cores	8
Memory	64 GB
OS	Fedora 18 (Linux Kernel 3.6.10)
VMM	KVM-kmod-3.6
Guest machine	
Number of vCPUs	1
Memory	1 GB
OS	Fedora 18 (Linux Kernel 3.6.10)

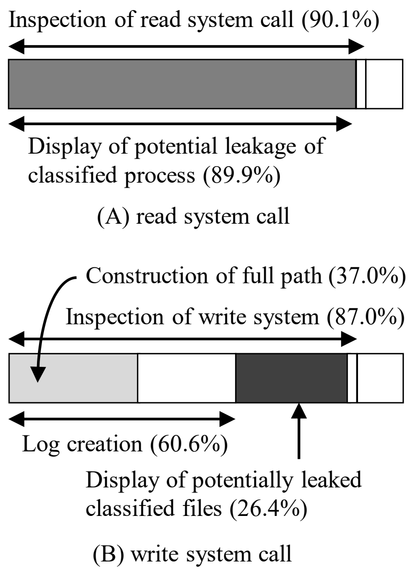


Figure 4: Detailed processing time proportions for hooking the SYSRET instruction

of a cp command, which targeted a classified file.

Figure 4 shows the detailed processing time proportions for hooking the SYSRET instruction. Figure 4 (A) shows the processing time proportions for hooking a read system call. Here, 90.1% of the processing time was consumed by processing the read system-call inspection, which includes hooking the read system call until transitioning to the guest OS. It also includes determining whether the file treated by the read system call is related to the diffusion of classified information. If so, the inspection process for the read system call collects and registers the information about the user process that requested it and outputs the results as log data. We analyzed the inspection process, and found that displaying the potential leakage of a classified process occupied 89.9% of the inspection processing time, caused by the list of processes being displayed whenever a new process registration occurs.

Figure 4 (B) shows the processing-time proportions for hooking a write system call. Here, 87.0% of the processing time was consumed by processing the write system call inspection, which includes hooking the write system call until transitioning to the guest OS. It also includes determining whether the file treated by the write system call is related to the diffusion of classified information. If so, the inspection process for the write system call collects and registers the information about this file and outputs the results as log data. We analyzed the inspection process and found that log creation occupied 60.6% of the inspection processing time, caused by the file list being displayed whenever a new file registration occurs. On the other hand, the process to display the list of potentially leaked classified files occupied 26.4%, and that of full path construction occupied 37.0%, which occurs when a process potentially involving leakage creates a new file or updates a file via a write system call. All of these types of overhead all considered excessive.

Therefore, an improved VMM-based tracing function must satisfy the following requirement:

Requirement 1: The processing overhead of the tracing function must be reduced.

4 Grasping the Potential Diffusion of Classified Information

When the administrator of the guest OS or that of the computer confirms the diffusion of classified information, the following two requirements are necessary:

Requirement 2: The administrator must grasp the list of the managed processes and files from the start of the tracing function to the present.

Requirement 3: The administrator must grasp the list of the managed processes and files currently registered.

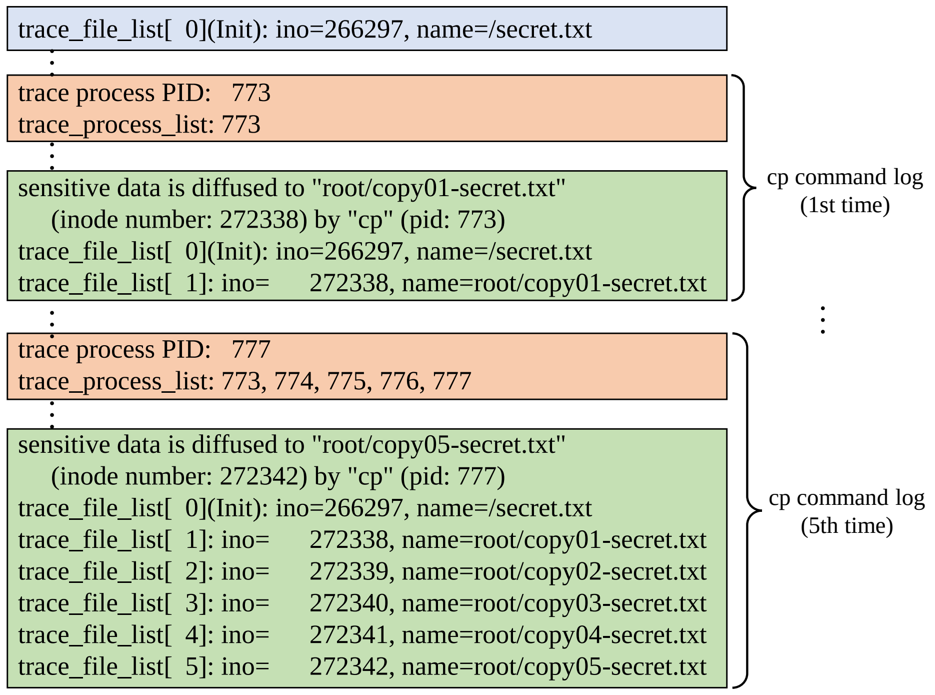
Considering service execution, by satisfying Requirement 2, the administrator can grasp the classified information that has been referenced, updated, or newly generated by the service. This will enable the administrator to determine whether the service is handling the classified information as intended. Moreover, by satisfying Requirement 3, the administrator can grasp the currently classified information.

The process tracing function output a list of all managed processes that may have potentially leaked information from the start of tracing to the present for each newly registered process. Similarly, the file tracing function output a list of all managed files that may have been leaked from the start of tracing to the present for each newly registered file. Figure 5 illustrates this log. Figure 5 illustrates a log in which `/secret.txt` (inode = 266297) is registered as a managed file with the potential to diffuse classified information. The file was duplicated five times using the `cp` command. Considering its fifth execution, the tracing function registered a newly managed process (PID = 777) when the read system call occurred, and the tracing function output information about the five managed processes (PID = 773, 774, 775, 776, and 777) as a system log from the start of tracing until the end. Moreover, the tracing function registered a newly managed file, `copy05-secret.txt` (inode = 272342), when the write system call occurred, and the tracing function output information about the six managed files (inode = 266297, 272338, 272339, 272340, 272341, and 272342) as a system log from the start of tracing until the end.

Notably, the tracing function required an editing process based on the log of all diffusions to satisfy Requirement 2. However, the function could not satisfy Requirement 3 because it did not output the log when a process or file was no longer being managed.

5 Improvement

To reduce the large overheads and satisfy Requirement 1, we decided to display information only when necessary. Therefore, we implemented Improvements 1 and 2 as follows:



```
trace_file_list[  0](Init): ino=266297, name=/secret.txt
  :
trace process PID:   773
trace_process_list: 773
  :
sensitive data is diffused to "root/copy01-secret.txt"
    (inode number: 272338) by "cp" (pid: 773)
trace_file_list[  0](Init): ino=266297, name=/secret.txt
trace_file_list[  1]: ino=      272338, name=root/copy01-secret.txt
  :
trace process PID:   777
trace_process_list: 773, 774, 775, 776, 777
  :
sensitive data is diffused to "root/copy05-secret.txt"
    (inode number: 272342) by "cp" (pid: 777)
trace_file_list[  0](Init): ino=266297, name=/secret.txt
trace_file_list[  1]: ino=      272338, name=root/copy01-secret.txt
trace_file_list[  2]: ino=      272339, name=root/copy02-secret.txt
trace_file_list[  3]: ino=      272340, name=root/copy03-secret.txt
trace_file_list[  4]: ino=      272341, name=root/copy04-secret.txt
trace_file_list[  5]: ino=      272342, name=root/copy05-secret.txt
```

Figure 5: Example of the classified information diffusion log

Improvement 1: Instead of the inspection process in a read system call displaying all potential leakages, it was changed to display a newly registered process only when a new process registration occurs.

Improvement 2: Instead of the inspection process in a write system call displaying all potential leakages of classified files, it was changed to display a newly registered file only when a new file registration occurred.

For write system calls, the process of constructing a full pathname also involved a large overhead; however, we did not change this procedure, because it is needed for tracing diffusion whenever a new file associated with a potential leakage is registered. Nonetheless, we believe that this information is not necessary for inspecting diffusion. By introducing a process that obtains the full pathname only when required by the user, it will be possible to reduce this type of overhead in the future.

Additionally, to satisfy Requirements 2 to 3, we implemented Improvements 3 and 4, respectively, as follows:

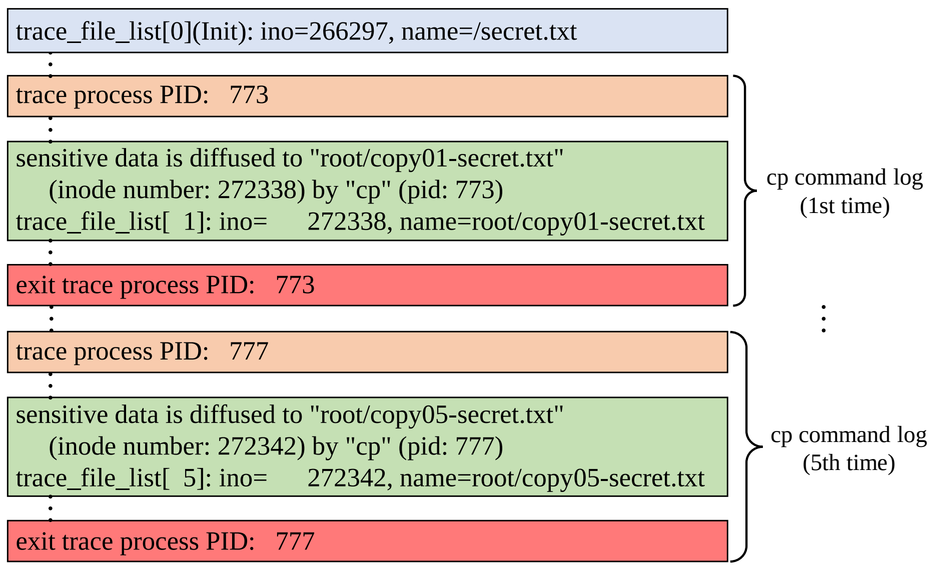
Improvement 3: The tracing function outputs log messages every time a process exits.

Improvement 4: The tracing function outputs log messages every time a file is removed.

In addition to the four improvements listed above, we implemented the following two:

Improvement 5: We introduced a processing function that integrates all classified information logs.

Improvement 6: We employ a processing function that edits all classified information logs.



```
trace_file_list[0](Init): ino=266297, name=/secret.txt
:
trace process PID:   773
:
sensitive data is diffused to "root/copy01-secret.txt"
    (inode number: 272338) by "cp" (pid: 773)
trace_file_list[  1]: ino=      272338, name=root/copy01-secret.txt
:
exit trace process PID:   773
:
trace process PID:   777
:
sensitive data is diffused to "root/copy05-secret.txt"
    (inode number: 272342) by "cp" (pid: 777)
trace_file_list[  5]: ino=      272342, name=root/copy05-secret.txt
:
exit trace process PID:   777
```

Figure 6: Example of the classified information log of the improved tracing function

Consequently, Improvements 5 and 6 also satisfy Requirements 2 and 3, respectively.

Figure 6 illustrates the classified information log following the implementation of the two new and existing improvements. Like Figure 5, Figure 6 illustrates a log in which `/secret.txt` (inode = 266297) is registered as a managed file, and the file is duplicated five times using the `cp` command. Considering the fifth execution of the `cp` command, the tracing function registers a newly managed process (PID = 777) and outputs only this process information as a system log. Moreover, the tracing function registers the newly managed file, `copy05-secret.txt` (inode = 272342), and outputs only this file information as a system log. When the fifth execution of the `cp` command is completed, the tracing function also outputs the finished process (PID = 777) information as a system log leveraging Improvements 3 and 4.

The processing function of Improvement 5 integrates all classified information logs. The processing function for editing all the logs shown in Improvement 6 is realized by excluding the finished process or removing file information.

6 Evaluation

6.1 Overview

We implemented the improved method described in Section 3 and evaluated the improvements to the VMM-based tracing function performance by measuring each of the following three processes:

1. Processing time of the VMM-based tracing function at a `cp` command:
   To measure the processing time of the VMM-based tracing function in the read and write system calls, we used a `cp` command, like in the evaluation described in Section 3. We targeted a classified file in the execution of this program and compared each result. As explained in Section 3, we inserted a “get timestamp” process in the VMM-based tracing function and compared the execution times before and after implementing the improvements.

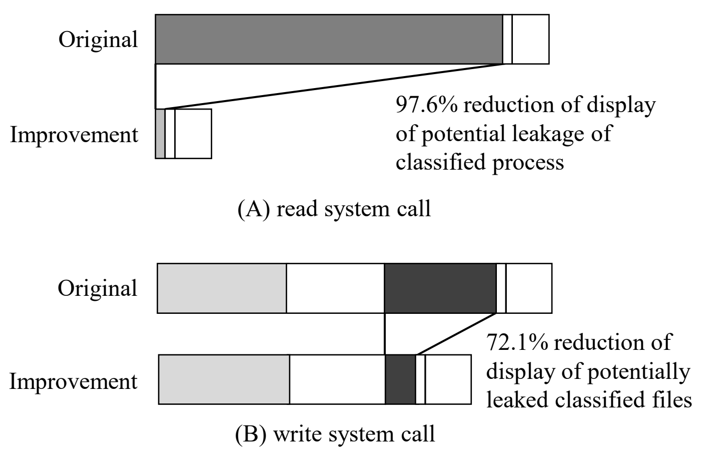


Figure 7: Acceleration effect of improvements: processing times of read and write system calls

2. Processing time of the read and write system calls (overall):
   To measure the processing time of read and write system calls, we built a workload program that obtains all data related to a file using the read system call and writes them to another file using the write system call. We targeted a classified file and an unclassified file in the execution of this program and compared each result.

3. Benchmark for accessing files:
   To measure the processing time of a benchmark for file access, we used the Flexible I/O Tester (Fio) benchmark, which measures the processing time using four types of file access patterns: Random Read, Random Write, Sequential Read, and Sequential Write. We prepared 1,000 files whose size is 4-kB each and accessed them using a block size of 4-kB. We measured each case in which the tracing function registered the classified and unmanaged files. Moreover, we measured each case using both unimproved and improved tracing functions.

6.2 Processing Time of the VMM-based Tracing Function with the `cp` Command

Figure 7 shows the performance acceleration achieved from the VMM-based tracing function. Considering Figure 7 (A) and regarding the inspection processing time, the time needed to display the potential leakage of classified processes was reduced by 97.6% for a read system call. The proportion of processing time needed to display the potential leakage from hooking the read system call to transitioning to the guest OS dropped to 18.4% of the overall processing time.

Moreover, as shown in Figure 7 (B), considering the inspection processing time, the time needed to display the potential leakage was reduced by 72.1% for a write system call. The proportion of processing time needed to display the potential leakage from hooking the write system call to transitioning to the guest OS dropped to 9.4% of the overall processing time.

From these results, we confirmed that reducing the processing overhead was significant to improved VMM-based tracing function performance via Improvements 1 and 2, which targeted classified processes

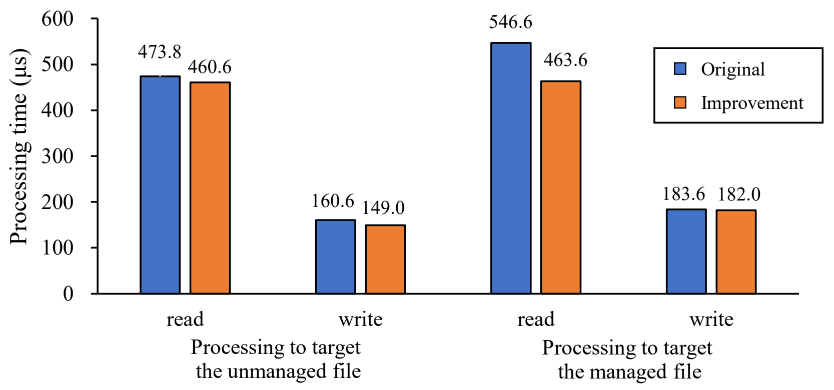


Figure 8: Processing time of the read/write system calls

or files, respectively.

6.3 Processing Time of the Read and Write System Calls (Overall)

We created a 100-kB text file with random alphanumeric characters and measured the execution times of the read system calls. Figure 8 shows their processing-time improvements. For the evaluation, we compared processing times before and after the improvements.

Considering the original VMM-based tracing function before improvement, the processing time of the read system call that targeted a classified file was 546.6 μs. After improvement, it was 463.6 μs, and the processing time was reduced by 83.0 μs. The processing time of the write system call, which targeted a classified file, was 183.6 μs in the original VMM-based tracing function. It dropped to 182.0 μs after improvement, and the processing time reduced by 1.60 μs. Regarding this evaluation, the processing time in Figure 8 includes read and write system calls that treated 100-kB data and processed the hooking system call. Although it was measurably improved in this case, the processing time outside the VMM-based tracing function was exceedingly large, making it difficult to confirm the improvement effect on the write system call.

The processing time of the read system call, which targeted an unclassified file, was 473.8 μs in the original VMM-based tracing function. After improvement, it was 460.6 μs, and the processing time reduced by 13.2μs. The processing time of the write system call, which targeted an unclassified file, was 160.6 μs in the original VMM-based tracing function It dropped to 149.0 μs after improvement, and the processing time reduced by 11.6 μs. Considering the VMM-based tracing function, which targeted an unclassified file, the effectiveness of reducing the processing time was insignificant both before and after improvement. Therefore, we observed that these results do not adequately demonstrate improved effects. Nevertheless, it showed an error of processing system calls and the hooking process.

Regarding the environment of the original VMM-based tracing function, the processing time of the read system call, which targeted a classified file, was 546.60 μs, and that of the read system call, which targeted an unclassified file, was 473.80 μs. The processing time for the classified file was 72.8 μs longer than that of the unclassified file. On the other hand, regarding the environment of the improvement VMM-based tracing function, the processing time of the read system call, which targeted a classified file, was 463.6 μs, and that of the read system call, which targeted an unclassified file, was 460.6 μs. Therefore, the difference of the processing time in the read system call between targeting a classified file and targeting an unclassified file was 3.0 μs, which is modest. From these measurement results, we

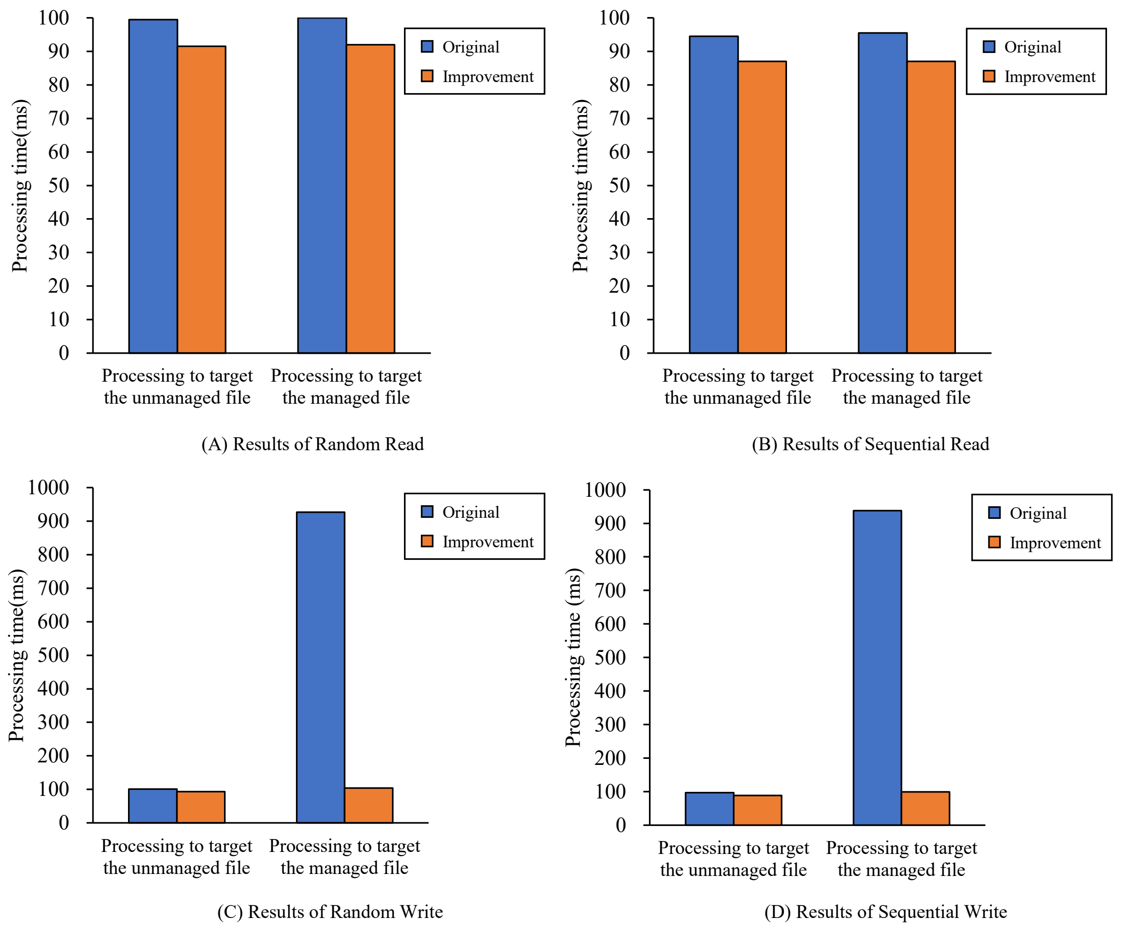


Figure 9: Processing time using the fio benchmark

confirmed that the effectiveness of reducing the processing overheads was significant in the improved VMM-based tracing function that targeted the classified file.

Regarding the environment of original VMM-based tracing function, the processing time of the write system call, which targeted a classified file, was 183.6 μs, and that of an unclassified file was 160.6 μs. The processing time for the classified file was 23.0 μs longer than that of the unclassified file. On the other hand, regarding the environment of the improvement VMM-based tracing function, the processing time of the write system call, which targeted a classified file, was 182.0 μs, and that of the read system call, which targeted an unclassified file, was 149.0 μs. Therefore, regarding the write system call, the processing time of the improved tracing function increased to 33.0 μs longer than that of the original tracing function. From these results, it is difficult to confirm the effect of improvements on the write system call because the processing time outside the VMM-based tracing function was exceedingly large.

6.4 Benchmark for File Access

The measurement results obtained using the fio benchmark are shown in Figure 9. The measurement results of the random read are shown in Figure 9 (A). The effectiveness of reducing processing overhead was not significant in either case. For example, the processing time of the targeted managed file was approximately 100 ms in the original and approximately 92 ms after improvement. Therefore, after improvement, the processing time was reduced by approximately 8 ms.

Additionally, the same characteristic in Figure 9 (A) was observed in Figure 9 (B). The processing time of the target managed file was approximately 95 ms in the original and 87 ms after improvement. Therefore, after improvement, the processing time was reduced by approximately 8 ms.

Therefore, considering the read access pattern, the reduction in processing time was less than 10%. The effectiveness of the improvement was insignificant for both random and sequential access patterns regardless of whether the trace target files were managed.

The measurement results of the random write are shown in Figure 9 (C). The effectiveness of reducing processing overheads was significant when the tracing function targeted a managed file. For example, the processing time of the target managed file was approximately 927 ms in the original and 104 ms after improvement. Thus, after improvement, the processing time was reduced by approximately 823 ms. However, the effectiveness of reducing the processing time was insignificant when the tracing function targeted an unmanaged file. Subsequently, the processing time was approximately 95 ms in the original and 87 ms after improvement. Therefore, after improvement, the processing time was reduced by approximately 8 ms.

The same characteristic demonstrated in Figure 9 (C) was observed in the sequential write shown in Figure 9 (D). The processing time of the target managed file was approximately 938 ms in the original and 99 ms after improvement. This indicates that, after improvement, the processing time reduced by approximately 839 ms. The processing time was approximately 97 ms in the original and 88 ms after improvement. Hence, after improvement, the processing time was reduced by approximately 9 ms.

Considering the write access pattern, the reduction in processing time was approximately 89%, and the effectiveness of the improvement in targeting the managed file was significant for both random and sequential writes. Meanwhile, the reduction in processing time was approximately 8–9%, and the effectiveness of the improvement targeting the unmanaged file was insignificant.

Considering these results, the effectiveness of the improvement in the read access pattern for the managed file was insignificant, and that in the write access pattern was substantial. The effectiveness of the improvement in the read access pattern appears insignificant, compared with the processing time in the original and after improvement, because the registration of the managed process occurred once. However, the effectiveness of the improvement in the write access pattern appears substantial, compared with the processing time in the original and after improvement, because the registration of the managed files occurred 1,000 times.

7 Related Work

Table 2 shows a comparison of the proposed method with those of existing works, focusing on the purpose of each approach, the event monitoring technique, the monitoring system implementation location, and the isolation method. Next, we provide a detailed narrative comparison.

Virtualization technology is widely used for security monitoring [6, 7], and it mainly consists of in-virtual machine (VM) [6, 8] and out-of-VM [9, 10, 7] approaches. Considering the in-VM approach, monitoring codes are inserted inside a VM and protected from malicious processes and distrusted kernels by the hypervisor. In-VM monitoring can quickly collect semantic information; hence, it is utilized for malware analysis and other security capabilities. Sharif et al. proposed a secure in-VM monitoring (SIM) method using hardware virtualization [6]. Secure in-VM code runs inside a distrusted VM to monitor its interior; however, the code and private data are placed in separate hypervisor-protected guest address spaces to provide an efficient monitoring method. Nonetheless, it requires monitor insertion into a guest VM. SecPod uses a similar approach to protect monitor codes inside a VM [8]. Although SIM uses shadow page tables for code protection, SecPod uses both shadow and nested page tables to reduce

Table 2: Comparison of the proposed method with those of existing works

Method	Purpose	Monitoring events	Implementation place	Isolation
Proposed method	Monitoring guest VM's system call and tracing the diffusion of classified information	System call	On KVM's VMM (outside of the monitored VM)	Monitoring system is isolated from target VM
Sharif et al. [12]	Secure VM monitoring by using memory protection and virtualization	System call	Inside of the monitored VM	Monitoring system is implemented inside a target VM
Zhan et al. [18]	Monitoring by SecPod: a framework for virtualization-based security systems	Paging operation	Inside of the monitored VM	Monitoring system is implemented inside a target VM
Pfoh et al. [11]	Monitoring by Nitro: for hardware-based system call tracing and monitoring	System call	On KVM's hypervisor (outside of the monitored VM)	Monitoring system is isolated from target VM
Garfinkel et al. [5]	Monitoring by a system of a VMI-based architecture for intrusion detection	The hardware state	On VMI IDS (outside of the monitored VM)	Monitoring system is isolated from target VM
Srinivasan et al. [14]	Monitoring by a process out-grafting: out-of-VM and fine-grained process execution monitoring system	Fine-grained process execution	On Security VM (outside of the monitored VM)	Monitoring system is isolated from target VM
Hizver et al. [6]	Monitoring by a real-time kernel data structure monitoring (RTKDSM) system	Process operation	On Monitoring VM and inside hypervisor (outside of the monitored VM)	Monitoring system is isolated from target VM
Shi et al. [13]	Monitoring all hypercalls belongs to the VMs of one hypervisor	Hypercall	On Dom0 and hypervisor of Xen (outsid of the monitored VM)	Monitoring system is isolated from target VM
Zhan et al. [18]	Monitoring by a page-level dynamic VMI-based kernel CFI checking system	Kernel function	On Privileged VM (outside of the monitored VM)	Monitoring system is isolated from target VM
Jia et al. [9]	Monitoring by T-VMI which eliminates the risk of privacy leakage from VMM by using TrustZone	Event-trigger	On trusted firmware (outside of the monitored VM)	Monitoring system is isolated from target VM
Enck et al. [1]	Monitoring by TaintDroid, dynamic taint tracking and analysis system	Multiple information-flow of sensitive data	Inside of the monitored OS (Android)	Monitoring system is implemented inside target kernel
Ji et al. [8]	Monitoring by a system to investigate attacks using information flow tracking	System call	Inside of the monitored OS	Monitoring system is implemented inside target kernel
Huseynov et al. [7]	Checking virtual machines for the presence of keyloggers using artificial immune system (AIS) based technology	Events (interrupts, system calls, memory writes, network activities, etc.)	Outside of the monitored VM	Monitoring system is isolated from target VM
Taubmann et al. [16]	VM monitoring using VMI method with less performance degradation	System call	On Monitoring VM and inside hypervisor (outside of the monitored VM)	Monitoring system is isolated from target VM

unnecessary world switches during virtualization, and nested page tables are used for protection. Both methods employ similar concepts for VM interior monitoring and securing the monitor codes from a distrusted kernel. In contrast, the VMM-based tracing function does not require monitor insertion; thus, it can be adapted to various OSs in a non-intrusive manner.

Nitro [7] is a fast system-call tracing mechanism that addresses the latter problem of out-of-VM monitoring. Monitoring information for which the location and format are undefined by hardware specifications necessitates a large overheads. Therefore, Nitro collects only guest register values for greater efficiency. If a user requires more information, Nitro acquires it. The performance improvement method for the VMM-based tracing function reported herein utilizes a similar approach. By acquiring information only when necessary, our improvement technique can reduce the overhead caused by diffusion tracing.

Hizver et al. proposed a method for improving VM monitoring performance [11, 12, 13]. To reduce the performance degradation on a VM introspection (VMI) method, Hizver et al. proposed another method for monitoring at regular intervals instead of constantly. However, it has been shown that missed detection can occur when monitoring at regular intervals. Similarly, Shi et al. achieved performance improvements by setting the extended page tables (EPT) protection for monitoring at regular intervals [12].

Considering our proposed method, missed detection does not occur because the monitoring of the diffusion of classified information is constant. Additionally, we achieved performance improvements by reducing the output log file. We did not compare performance evaluations in this case because these methods differ from ours, considering the purpose of the system and the target environment.

Zhan et al. proposed a method for fine-grained control-flow integrity for VM verification, satisfying the performance requirements of actual operations [13]. Their method recommends code execution detection in page units compared with the correct processing flow, which detects a branch to prevent the VMM from being called frequently and to suppress performance degradation. However, the detection accuracy of this method is lower than that of branch detection. Considering our proposed method, we avoided reducing the importance of classified information during the improvement.

Jia et al. proposed a method to guarantee the integrity of VMM program code and the validity of data because the VMM used by the VMI and host OS can be damaged by an attack in a cloud environment [14]. Their proposed method was based on the premise of VMM safety.

Enck et al. proposed a system for information-flow tracking on Android [15] that tracks information flows via taint analysis using modified libraries. Our proposed method does not require the modification of programs running on VMs.

Ji et al. proposed a system that investigates attacks using information flow tracking [16], achieving low overhead by recording system-call events and accurately monitoring using on-demand process replay. Although the proposed method collects all the information required for tracing the diffusion of classified information, suppressing the log output reduces unnecessary performance degradations. Moreover, owing to the on-demand log display function, the system manager can analyze diffusion using log information.

Huseynov et al. mentioned that, to detect attacks on a kernel layer, such as a keylogger, it is necessary to inspect the kernel integrity [17]. They proposed a secure environment by constantly checking VMs using artificial immune-system-based technology. Our proposed method cannot detect keylogger processing because it aims to trace the diffusion of classified information from registered files.

Owing to the VMI VM monitoring method that causes large performance degradation, Taubmann et al. proposed a VMI monitoring method that reduces performance overhead [18], which was caused by monitoring various events. Therefore, it is possible to reduce performance overheads by filtering out unnecessary events. On the other hand, the proposed method of this work hooks all system calls of the target VM, but the VMM-based tracing function eliminates the monitoring of unnecessary system calls by filtering the system-call number. Therefore, the proposed method reduces performance overhead.

8 Conclusion

In this paper, we proposed an improved VMM-based tracing function and reported its performance evaluation. First, we analyzed the process flow of the VMM-based tracing function, and conducted a brief evaluation based on the use of the `cp` command. Hence, we found a large processing time for hooking the SYSRET instruction, indicating large overhead processing. Subsequently, we evaluated the processing of the VMM-based tracing function in detail and determined that the processing time for the display of a potential leakage of a classified process occupied 89.9% of the processing time from hooking a read system call until transitioning to the guest OS. A classified file occupies 26.4% of the processing time for hooking a write system call until transitioning to the guest OS. We clarified the problems regarding the processing and outputting of a log. To resolve the problem of performance overhead, we decided to acquire or display information only when necessary.

On the contrary, to enable the administrator of the guest OS or the computer to determine whether the service is handling the classified information as intended, the administrator must grasp the list of managed processes and files from the start of the tracing function to the present and those that are currently registered.

Thus, we implemented and evaluated an improvement method. We measured the performance of each of the three processing methods. First, we evaluated the processing time of the VMM-based tracing function by applying the cp command. We found that the proportion of processing time used to display the potential leakage of classified processes was 18.4% of the processing time by hooking a read system call until transitioning to the guest OS. Moreover, the classified files became 9.4% of the processing time, from hooking a write system call until transitioning to the guest OS. Second, we evaluated the processing time of read and write system calls. The processing time of the read system call, which targeted a 100-kB classified file, was 546.6 μs in the original VMM-based tracing function and 463.6 μs after the improvement. Hence, the processing time was reduced by 83.0 μs. Similar to the evaluation of the read system call, the processing time of the write system call was reduced by 1.60 μs after the improvement. Nonetheless, there was almost no difference between the original and improved performance, considering the scale of overhead processing. Therefore, we found that the effect of the improvement on the read system call, which targeted a classified file, was especially high. Third, we evaluated the processing time of benchmark file access. The reduction in processing time was approximately 89%, considering the write access pattern. Furthermore, the effectiveness of the improvement in targeting the managed file was significant. Considering these results, we achieved bona fide performance improvements for the VMM-based tracing function.

Acknowledgments

This work was partially supported by JSPS KAKENHI Grant Numbers 19H04109 and 19K20246.

References

[1] T. Tabata, S. Hakomori, K. Ohashi, S. Uemura, K. Yokoyama, and H. Taniguchi. Tracing classified information diffusion for protecting information leakage. *IPSJ Journal*, 50(9):2088–2102, September 2009.

[2] N. Otsubo, S. Uemura, T. Yamauchi, and H. Taniguchi. Design and evaluation of a diffusion tracing function for classified information among multiple computers. In *Proc. of the 7th FTRA International Conference on Multimedia and Ubiquitous Engineering (MUE'13), Seoul, Korea*, volume 240 of *Lecture Notes in Electrical Engineering*, pages 235–242. Springer Netherlands, May 2013.

[3] K. Fukushima, T. Yamauchi, and H. Taniguchi. Implementation of mechanism to support tracing diffusion of classified information by visualization and filtering function. *IPSJ Journal*, 53(9):2171–2181, September 2012.

[4] S. Fujii, M. Sato, T. Yamauchi, and H. Taniguchi. Evaluation and design of function for tracing diffusion of classified information for file operations with kvm. *The Journal of Supercomputing*, 72:1841–1861, February 2016.

[5] S. Fujii, M. Sato, T. Yamauchi, and H. Taniguchi. Design of function for tracing diffusion of classified information for ipc on kvm. *Journal of Information Processing*, 24(5):781–792, September 2016.

[6] M.I. Sharif, W. Lee, W. Cui, and A. Lanzi. Secure in-vm monitoring using hardware virtualization. In *Proc. of the 16th ACM Conference on Computer and Communications Security (ACM CCS'09), Chicago, IL, USA*, pages 477–487. ACM, November 2009.

[7] J. Pfoh, C. Schneider, and C. Eckert. Nitro: Hardware-based system call tracing for virtual machines. In *Proc. of the 2011 International Workshop on Security (IWSEC'11), Tokyo, Japan*, volume 7038 of *Lecture Notes in Computer Science*, pages 96–112. Springer, Berlin, Heidelberg, November 2011.

[8] X. Wang, Y. Chen, Z. Wang, Y. Qi, and Y. Zhou. Secpod: a framework for virtualization-based security systems. In *Proc. of the 2015 USENIX Annual Technical Conference (USENIX ATC'15), Santa Clara, CA, USA*, pages 347–360. USENIX, July 2015.

[9] T. Garfinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Proc. of the 2003 Network and Distributed System Security Symposium (NDSS'03), San Diego, CA, USA*, pages 191–206. The Internet Society, February 2003.

[10] D. Srinivasan, Z. Wang, X. Jiang, and D. Xu. Process out-grafting: An efficient "out-of-vm" approach for fine-grained process execution monitoring. In *Proc. of the 18th ACM Conference on Computer and Communications Security (ACM CCS'11), Chicago, IL, USA*, pages 363–374. ACM, October 2011.

[11] J. Hizver and T.-C. Chiueh. Real-time deep virtual machine introspection and its applications. In *Proc. of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE'14), New York, NY, USA*, volume 49, pages 3–14. ACM, July 2014.

[12] J. Shi, Y. Yang, and C. Tang. Hardware assisted hypervisor introspection. *SpringerPlus*, 5:647:1–647:23, May 2016.

[13] D. Zhan, L. Ye, B. Fang, H. Zhang, and X. Du. Checking virtual machine kernel control-flow integrity using a page-level dynamic tracing approach. *Soft Computing*, 22:7977–7987, 2018.

[14] L. Jia, M. Zhu, and B. Tu. T-vmi: Trusted virtual machine introspection in cloud environments. In *Proc. of the 2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID'17), Madrid, Spain*, pages 478–487. IEEE/ACM, May 2017.

[15] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L.P. Cox, J. Jung, P. McDaniel, and A.N. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems*, 32(2):1–29, June 2014.

[16] Y. Ji, S. Lee, E. Downing, W. Wang, M. Fazzini, T. Kim, A. Orso, and W. Lee. Rain: Refinable attack investigation with on-demand inter-process information flow tracking. In *Proc. of the 2017 ACM Conference on Computer and Communications Security (ACM CCS'17), Dallas, TX, USA*, pages 377–390. ACM, October 2017.

[17] H. Huseynov, K. Kourai, T. Saadawi, and O. Igbe. Virtual machine introspection for anomaly-based keylogger detection. In *Proc. of the 21st IEEE International Conference on High Performance Switching and Routing (HPSR'20), Newark, NJ, USA*, pages 1–6. IEEE, May 2020.

[18] B. Taubmann and H.P. Reiser. Towards hypervisor support for enhancing the performance of virtual machine introspection. In *Proc. of 20th International Conference on Distributed Applications and Interoperable Systems (DAIS'20), Valletta, Malta*, volume 12135 of *Lecture Notes in Computer Science*, pages 41–54. Springer, Cham, June 2020.

Author Biography

Hideaki Moriyama received his B.E., M.E., and Ph.D. degrees from Okayama University, Japan in 2007, 2009, and 2012, respectively. He has been an Assistant Professor at the Department of Electronics and Information Engineering at the National Institute of Technology, Ariake College, since 2012; in 2014, he became a Lecturer. Since 2020, he has been serving as an Associate Professor. His research interests include operating systems and virtualization technology. He is a member of IPSJ.

Toshihiro Yamauchi received the 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 with the Japan Society for the Promotion of Science. In 2002, he became a Research Associate with the Faculty of Information Science and Electrical Engineering, Kyushu University. In 2005, he became an Associate Professor with the Graduate School of Natural Science and Technology, Okayama University. Since 2021, he has been serving as a Professor at Okayama University. His research interests include operating systems and computer security. He is a member of ACM, IEEE, USENIX, IPSJ, and IEICE.

Masaya Sato received his B.E., M.E., and Ph.D. degrees from Okayama University, Japan in 2010, 2012, and 2014 respectively. In 2013 and 2014 he was a Research Fellow of the Japan Society for the Promotion of Science. He had served as an Assistant Professor of Graduate School of Natural Science and Technology at Okayama University from 2014 to 2021. Since 2021, he has been serving as an associate professor at Okayama Prefectural University. His research interests include computer security and virtualization technology. He is a member of IPSJ, IEICE, and ACM.

Hideo Taniguchi received a B.E. degree in 1978, a M.E. degree in 1980 and a Ph.D. degree in 1991, all from Kyushu University, Fukuoka, Japan. In 1980, he joined NTT Electrical Communication Laboratories. In 1988, he moved to Research and Development Headquarters, NTT DATA Communications Systems Corporation. He has been an associate professor of computer science at Kyushu University since 1993, a professor of the Faculty of Engineering at Okayama University since 2003. He has been a dean of Faculty of Engineering from April 2010 to March 2014 and a vice president from April 2014 to March 2017 at Okayama University. His research interests include operating system, real-time processing and distributed processing.