Noname manuscript No.
(will be inserted by the editor)

Evaluation and Design of Function for Tracing Diffusion of Classified Information for File Operations with KVM

Shota Fujii · Masaya Sato ·
Toshihiro Yamauchi · Hideo Taniguchi

Received: date / Accepted: date

Abstract Cases of classified information leakage have become increasingly common. To address this problem, we have developed a function for tracing the diffusion of classified information within an operating system. However, this function suffers from the following two problems: first, in order to introduce the function, the operating system's source code must be modified. Second, there is a risk that the function will be disabled when the operating system is attacked. Thus, we have designed a function for tracing the diffusion of classified information in a guest operating system by using a virtual machine monitor. By using a virtual machine monitor, we can introduce the proposed function in various environments without modifying the operating system's source code. In addition, attacks aimed at the proposed function are made more difficult, because the virtual machine monitor is isolated from the operating system. In this paper, we describe the implementation of the proposed function for file operations and child process creation in the guest operating system with a kernel-based virtual machine. Further, we demonstrate the traceability of diffusing classified information by file operations and child process creation. We also report the logical lines of code required to introduce the proposed function and performance overheads.

Keywords Information Leak Prevention · Virtualization · Semantic Gap · VMM

S. Fujii · M. Sato · T. Yamauchi · H. Taniguchi
Graduate School of Natural Science and Technology, Okayama University, 3-1-1 Tsushima-naka, Kita-ku, Okayama 700-8530, Japan
E-mail: fujii@swlab.cs.okayama-u.ac.jp

M. Sato
E-mail: sato@cs.okayama-u.ac.jp

T. Yamauchi
E-mail: yamauchi@cs.okayama-u.ac.jp

H. Taniguchi
E-mail: tani@cs.okayama-u.ac.jp

1 Introduction

As personal information has become increasingly valuable, need for preventing information leaks increasing. According to an analysis [1] of the incidents of personal information leakage, leaks often occur as a result of inadvertent handling and mismanagement, and this accounts for approximately 57% of all known cases of information leaks. To prevent information leaks, it is important for the user to grasp the situation surrounding the classified information. On the other hand, incidents that aim at stealing classified information are occurring with increasing frequency. In such a context, there is always the risk of increased damage when the victim cannot detect the information leak.

To trace the status of classified information in a computer, and to manage the resources that contain the classified information, we have developed a function for tracing the diffusion of classified information [2] (particularly, an operating system (OS)-based tracing function). This function manages any process that has the potential to diffuse classified information. In addition, the function represents the extent of the diffusion by using a directed graph [3], and it traces the diffusion of the classified information in multiple computers [4]. However, the function cannot be introduced in a closed-source OS such as Windows, because introducing it requires the modification of the source code. Moreover, the OS-based tracing function is executed within the OS, and therefore, has the potential of being detected and disabled. If the OS-based tracing function is disabled, as already mentioned, the victim cannot detect the information leak and there is a risk of increased damage. Further, when the kernel version is updated, the function must be adapted to the newly updated kernel.

Similar to the OS-based tracing function, a large number of method protecting sensitive files have been proposed [5][6][7]. However, since they are implemented within the OS, there is a problem that the operational environment is limited and they can be detected and disabled, just like the OS-based tracing function. To resolve the above mentioned problems, methods that protect sensitive files from outside the OS have been proposed [8][9]. These methods demonstrate the effectiveness of implementing the security system outside the OS. However, it is difficult to identify the cause of information leaks because these methods are aimed at only information leakage prevention and do not aim to grasp the diffusion and leakage path of classified information.

To resolve these problems, we designed a function for tracing the diffusion of classified information in a guest OS by using a virtual machine monitor (VMM). This VMM-based tracing function is implemented by modifying the VMM. Therefore, the VMM-based tracing function can be implemented without modifying the OS's source code. Further, it is expected that attacks specifically target this function will be difficult, because the VMM is more robust than the OS.

This paper describes the implementation of the function for file operations and child process creation with a kernel-based virtual machine (KVM) [10]. A preliminary description of the VMM-based tracing function has already been presented in our previous paper [11]; it focused on introducing the basic implementation. The VMM-based tracing function hooks system calls that possibly cause information leakage. Moreover, the function traces the status of the classified information in the guest OS from outside it. We have implemented a prototype of the VMM-based tracing function for a Linux guest. This paper also describes an evaluation including traceability, amount of the modified source code, and performance of the VMM-based tracing function.

In summary, we make the following contributions:

- We have pointed out the problems with the OS-based tracing function and the existing methods as follows: 1) The existing approaches require modification to the OS's source code and 2) there is a risk that the tracing function will be disabled when the OS is attacked.
- We have designed a function for tracing the diffusion of classified information in a guest OS by using a VMM. This makes it possible to introduce the tracing function without modifying the OS's source code. Moreover, attacks aimed at the proposed function are made more difficult, because the VMM is isolated from the guest OS. The tracing function will also continue to be available even if the kernel version is updated, provided that the system call specifications and the data structure remain unchanged.
- We have evaluated and reported the traceability, modified code size, and performance of the VMM-based tracing function.

The remainder of this paper is organized as follows: Section 2 presents an overview of the OS-based tracing function. We present the design of the VMM-based tracing function in Section 3 and describe the implementation details in Section 4. Section 5 presents the experimental results. We discuss the related works in Section 6 and conclude with Section 7.

2 OS-Based Function for Tracing the Diffusion of Classified Information

2.1 Classified Information Diffusion Path

The OS-based tracing function [2] manages any file or process that has the potential to diffuse classified information. Classified information can be diffused by any process that involves opening the classified file, reading its content, communicating with another process, or writing such content to another file. Therefore, the diffusion of classified information is caused by the following operations:

(1) File operation
(2) Inter-process communication
(3) Child process creation

The OS-based tracing function traces the diffusion of classified information by monitoring these operations.

2.2 Overview of the OS-Based Tracing Function

Figure 1 shows an overview of the OS-based tracing function. The OS-based tracing function traces the diffusion of classified information as follows:

(1) System calls that are related to the diffusion of classified information are hooked.

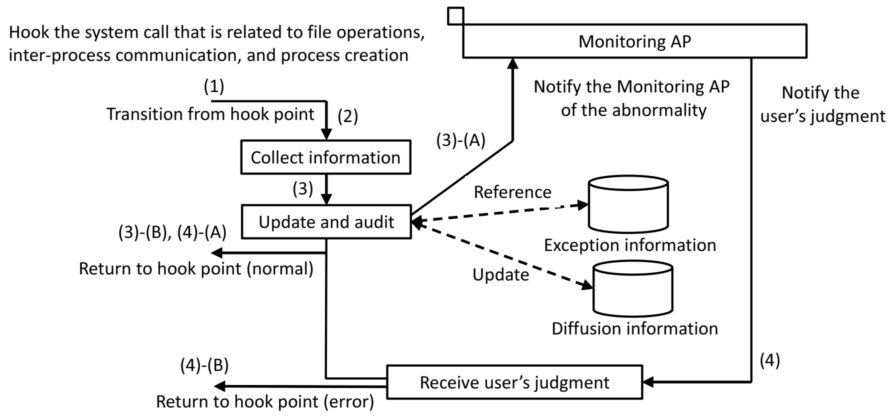

Hook the system call that is related to file operations, inter-process communication, and process creation
(1)
Transition from hook point
(2)
Collect information
(3)
Update and audit
(3)-(B), (4)-(A)
Return to hook point (normal)
(3)-(A)
Monitoring AP
Notify the Monitoring AP of the abnormality
Notify the user's judgment
Reference
Exception information
Update
Diffusion information
(4)
(4)-(B)
Return to hook point (error)
Receive user's judgment

Fig. 1 Overview of the OS-based tracing function.

(2) The OS-based tracing function collects information for tracing the diffusion of classified information such as the file that is handled by the system call or the transmission-destination process.

(3) The OS-based tracing function updates the diffusion information by using the information that is collected in (2) and audits its potential for leaking classified information.

  (A) When the audit detects the possibility of a classified information leak, it notifies the monitoring application program (AP).

  (B) When the audit does not reveal any possibility of classified information leakage, control is returned to the system call.

(4) After receiving the results of the user's judgment from the monitoring AP, the OS-based tracing function controls the system call in accordance with the user's judgment.

  (A) When the user's judgment is affirmative, the system call processing is continued.

  (B) When the user's judgment is negative, the system call processing is terminated as an error.

In addition, the OS-based tracing function excludes files and processes that are unrelated to the diffusion of classified information. These files and processes are registered with the exception information.

2.3 Problems with the OS-Based Tracing Function

The tracing function has the following problems:

Problem 1 The OS's source code must be modified before introduction.
In order to introduce the OS-based tracing function, it is necessary to modify the OS's source code. Therefore, the OS-based tracing function cannot be introduced in a closed-source OS such as Windows. Furthermore, when the kernel version of the OS is updated, the OS-based tracing function must modify the source code again after the OS is updated.

Problem 2 There is a risk of an attack invalidating the tracing function
The OS-based tracing function is implemented in the OS. Therefore, an adversary or a malicious user can invalidate the function by attacking the OS. If the function is invalidated, it becomes difficult to prevent information from being leaked and grasp the location of the classified information.

Further, as described in Section 1, there are similar problems in the existing methods for protecting sensitive files. In this paper, we propose a method that resolves both problems.

3 Function for Tracing the Diffusion of Classified Information in a Guest OS by Using a Virtual Machine Monitor

3.1 Requirements

To resolve the problems detailed above in Section 2.3, the following are required:

Requirement 1 The OS's source code must not be modified.
One solution to Problem 1 is to avoid the modification of the OS's source code. This ensures that the function can be implemented in a closed-source OS such as Windows.

Requirement 2 The function should be isolated from the OS.
Isolating the function from the OS is a solution to Problem 2. Such a solution makes it difficult for an adversary or a malicious user to attack the function directly.

3.2 Overview of the VMM-Based Tracing Function

The VMM-based tracing function is functionally equivalent to the OS-based tracing function. In particular, the VMM-based tracing function manages any file or process that has the potential to diffuse classified information. Moreover, the VMM-based tracing function traces the status of the classified information in a computer, and manages the resources that contain the classified information by monitoring three operations as described in Section 2.1. The user can always grasp the location of the classified information by using the list of the classified information stored in the VMM. Furthermore, when the diffusion of classified information is detected, the VMM-based tracing function records the pathname of the destination file, inode number, command name that is the cause of diffusion, and process ID (PID). Therefore, the user can detect the information leaks by using the above information and suppress the damage even if the classified information is leaked.

Figure 2 shows an overview of the VMM-based tracing function. The VMM-based tracing function traces the diffusion of the classified information as follows:

(1) A user program in the guest OS requests a system call.
(2) The VMM-based tracing function hooks the system call in the guest OS from the VMM. After identifying the hooked system call, the following system call processing is performed.

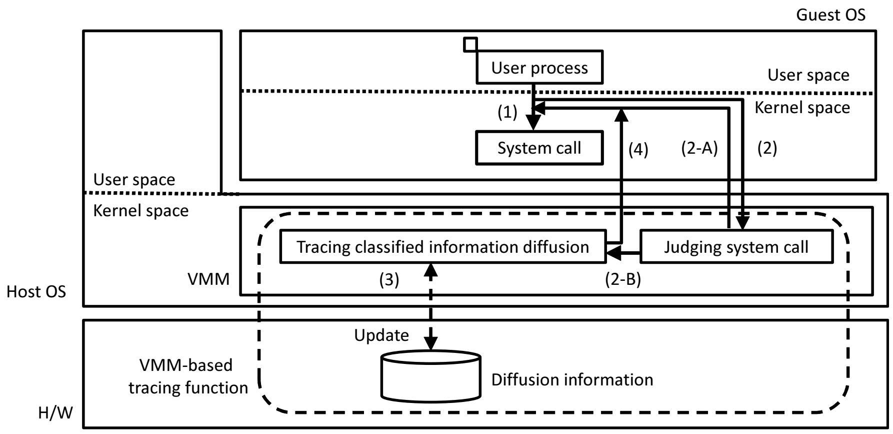


Fig. 2 Overview of the VMM-based tracing function.

(A) When 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) When the hooked system call is related to the diffusion of classified information, the VMM-based tracing function collects the information needed to trace its diffusion.

(3) The VMM-based tracing function updates the diffusion information by using the information that is collected in (2-B) if the classified information is diffused.
(4) Control is returned to the guest OS and the system call process is continued.

Given these steps, the VMM-based tracing function provides the guest OS with functions that are equivalent to the OS-based tracing function, without the need to modify the OS source code.

3.3 Tasks

To implement the VMM-based tracing function, the following tasks are required:

Task 1 Collecting the system call information with the VMM.
The classified information is diffused by the system call. Therefore, it is necessary to hook the system call. Further, the VMM-based tracing function collects the system call's information owing to the judgment of whether the system call is related to the classified information diffusion.

Task 2 Collecting the OS information with the VMM.
The VMM-based tracing function manages any file or process that has the potential to diffuse classified information. Therefore, it is necessary to collect the information from the OS, such as the processes that are running, their transmission destination, and the files handled by the processes that are running.

In Sections 3.4 and 3.5, we describe the procedure by which the above tasks are accomplished. Further, this procedure is tailored for a 64-bit version of Linux in which the system call is executed by SYSCALL/SYSRET.

3.4 Collecting System Call Information with Virtual Machine Monitor

3.4.1 Hooking a System Call Entry

The VMM-based tracing function hooks the system call entry (viz., SYSCALL). By hooking SYSCALL, we can detect system call requests. In order to hook SYSCALL, the VMM-based tracing function sets the value of the guest OS's `MSR_LSTAR` to the value of an unused page address. Executing SYSCALL changes the instruction pointer to the value in `MSR_LSTAR`, resulting in a page fault. Therefore, the VMM-based tracing function can hook the SYSCALL with the VMM by detecting page faults in the guest OS.

3.4.2 Hooking a System Call Exit

Each system call returns information concerning the success or failure of the system call, and the details of the file handled by the system call as a return value. It is necessary to collect the details about the file that is handled by the running process so that the VMM-based tracing function can trace the diffusion of the classified information. Thus, the VMM-based tracing function hooks the system call exit (viz., SYSRET). By hooking SYSRET, it is possible to obtain the system call's return value. In order to hook SYSRET, the VMM-based tracing function sets the breakpoint-address register to SYSRET's address. A breakpoint-address register specifies the breakpoint address and a debug exception is generated when a memory access is made to the breakpoint address. Thus, a debug exception occurs upon executing SYSRET. Therefore, the VMM-based tracing function can hook SYSRET with the VMM by detecting debug exceptions in the guest OS.

3.4.3 Collecting Information

It is necessary to judge whether the hooked system call is related to the diffusion of the classified information. To identify the system call, the VMM-based tracing function uses a system call number. In addition, it is necessary for the VMM-based tracing function to identify the transmission-destination file or process. A system call takes the file or process information to an argument. By obtaining the system call's argument, it is consequently possible to identify the transmission-destination file or process. Furthermore, as we have already described, the VMM-based tracing function obtains the system call's return value and utilizes the return value for identifying the transmission-destination file or process.

3.5 Collecting OS Information with Virtual Machine Monitor

The VMM-based tracing function traces the diffusion of classified information using information from the OS, such as process information and file information. Then, the semantic gap [12] must be bridged so that the VMM-based tracing function can obtain the OS information with the VMM. The semantic gap is the gap between the guest OS as it is viewed from the outside and the view of it from the inside. To bridge the semantic gap, the VMM-based tracing function constructs a semantic view by retrieving information about the guest OS beforehand.

4 Implementation of VMM-Based Tracing Function for File Operations and Child Process Creation

4.1 Environment

In this section, we describe the implementation of the VMM-based tracing function by using a KVM as the VMM and a 64-bit Linux OS with the 3.6.10 kernel as the guest OS. The VMM-based tracing function detects requests for system calls by hooking SYSCALL, and it obtains return values by hooking SYSRET. Therefore, the system call in the guest OS is executed by SYSCALL/SYSRET. Further, the guest OS is fully virtualized with Intel Virtualization Technology (VT).

4.2 Tracing Classified Information for Each Path

4.2.1 File Operation

The VMM-based tracing function hooks the open(), read(), write(), and close() system calls that are related to file operations. Further, to trace the diffusion of classified information by file operations, the VMM-based tracing function collects the following information:

(1) Current-process identifier
(2) Identifier of the file that is handled by the system call.

It is necessary for the VMM-based tracing function to collect the current-process identifier in order to judge whether the process requesting the system call is a management target when the VMM-based tracing function hooks each system call. To identify the current process, the VMM-based tracing function uses the PID. The VMM-based tracing function obtains the PID when the function hooks the SYSCALL. Moreover, it is necessary for the VMM-based tracing function to identify the file that is handled by the system call when the VMM-based tracing function judges whether the file that is read is a management target, and to register the written file with the diffusion information. To identify the file that is handled by the system call, the VMM-based tracing function uses the inode number. The VMM-based tracing function obtains the inode number by following the data structure from the process-control block to the file structure. Then, the VMM-based tracing function identifies the inode number by using the file descriptor. The file descriptor is obtained with the system call's return value in cases where open() is hooked. Likewise, the file descriptor is obtained by the system call's argument in cases where read(), write(), and close() are hooked.

4.2.2 Child Process Creation

The VMM-based tracing function hooks the clone() system call, which is related to child process creation. Moreover, in order to trace the diffusion of classified information by child process creation, the VMM-based tracing function collects the following information:

(1) System call's product identifier

(2) Parent-process identifier
(3) Child-process identifier

The clone() system call creates not only a new process but also a new thread. The threads in the same process share resources such as information related to the opened files. Therefore, thread creation does not diffuse the classified information outside the process. On the other hand, child process creation diffuses resources from the parent process to the child process. Thus, it is necessary to judge whether the clone() creates a process or a thread. If the thread is created, the `CLONE_THREAD` flag, which is the argument of clone(), is set. Consequently, by auditing the argument of clone(), we can judge whether the product of clone() is a process or thread. The `CLONE_THREAD` flag is obtained from the clone() argument when the VMM-based tracing function hooks the clone() system call entry.

Moreover, when the parent process is a management target, there is a risk that the classified information will be diffused to the child process. Therefore, to judge whether the parent process is a management target, it is necessary to collect the parent-process identifier. To do so, the VMM-based tracing function uses the parent process's PID. The parent process's PID is obtained from the process-control block when the VMM-based tracing function hooks the clone() system call entry.

Furthermore, the VMM-based tracing function registers the child process with the diffusion information when the VMM-based tracing function judges that the classified information is diffused to the child process. Thus, the child-process identifier must be obtained. To identify the child process, the VMM-based tracing function uses the child process's PID. When clone() creates a new process, the return value is the child process's thread ID (TID) and the TID is identical to its PID. Thus, the child process's PID is obtained from the return value of clone() when the VMM-based tracing function hooks the clone() system call exit.

5 Evaluation

5.1 Experimental Setup

We evaluated the traceability of the VMM-based tracing function. In addition, we measured the following two items.

(1) Lines of code (LOC)
(2) Overhead

To evaluate the cost for implementation, we compared the amount of LOC of the OS-based tracing function and the VMM-based tracing function. Further, we measured the overhead incurred by the VMM-based tracing function and compared it with the overhead incurred by the OS-based tracing function. Because the additional overhead due to virtualization is expected to be generated by implementing the tracing function within the VMM, we also compared the performance of the VMM-based tracing function with that of an unmodified VMM. With this comparison, we evaluated the performance overhead of the VMM-based tracing function excluding the virtualization overheads. We measured the performance of the system call, microbenchmark, and application (AP) in a virtualized environment.

Table 1 shows the evaluation environment. We evaluated the VMM-based tracing function with Core i5-3470 (3.2 GHz, 4 CPUs) and 4,096 MB of memory. The guest OS is allocated one virtual CPU and 1,024 MB of memory. Hyper-threading and EPT are disabled.

5.2 Traceability

5.2.1 Evaluation Methods of Traceability

To evaluate the traceability of the VMM-based tracing function, we performed the following scenario.

(Assumed Scenario 1) Export to external device
After editing a text file using the text editor, we write out the edited data onto a USB memory. The same processing is performed for an unmanaged file.
(Assumed Scenario 2) Copy of the directory unit
Prepare a directory that has 10 files, i.e., 5 classified files and 5 unclassified files, and copy this directory to another directory.

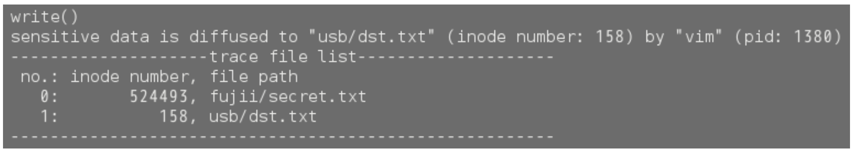
Using the above scenarios, we verify whether the VMM-based tracing function can trace the diffusion of the classified information.

5.2.2 Experimental Result of Traceability

Figure 3 shows the log generated by the VMM-based tracing function when (Assumed Scenario 1) is executed. As described in Section 3.2, when the diffusion of classified information is detected, the VMM-based tracing function records the pathname of the destination file, inode number, command name that is the cause of diffusion, and PID. In (Assumed Scenario 1), the classified file is `fujii/secret.txt` (inode number: 524493), and the file written in the USB memory is `usb/dst.txt` (inode number: 158). From the message shown in Fig. 3, we can infer that the

Table 1 Evaluation environment.

CPU		Intel Core i5-3470, 3.2 GHz
OS	Guest	Fedora 18 (Linux 3.6.10, 64bit)
	Host	Fedora 18 (Linux 3.6.10, 64bit)
Memory	Guest	1,024 MB
	Host	4,096 MB
VMM		KVM-kmod-3.6



```
write()
sensitive data is diffused to "usb/dst.txt" (inode number: 158) by "vim" (pid: 1380)
--------------------trace file list--------------------
 no.: inode number, file path
   0:       524493, fujii/secret.txt
   1:          158, usb/dst.txt
-------------------------------------------------------
```

Fig. 3 Log generated by the VMM-based tracing function when (Assumed Scenario) is executed.

classified information is diffused to `usb/dst.txt` by `vim`, which is the text editor. It also confirms that `usb/dst.txt` and `158`, which is the inode number of `usb/dst.txt`, are recorded in the trace file list. In contrast, the classified information is not diffused by the operation of the unclassified file. Then, we performed the same processing for an unmanaged file. In the above experiment, we observed that the information of the process executed on the unclassified file is not recorded.

However, when (Assumed Scenario 2) is executed by `cp -r src_dir dst_dir`, the 10 new files are all judged to be the classified file. Although five files are judged to be the classified files and other five files are judged to be unclassified file, the false positive occur. This false positive occurred because the process collectively executes `cp`, which is judged to be the classified process at the time point of reading data from a classified file and the files written out from this process are all judged to be classified files. On the other hand, when (Assumed Scenario 2) is executed by `find src_dir | xargs -iX cp X dst_dir`, a misdetection does not occur. This is because each copy operation is executed by one process and the above problem is avoided by combining `find`, `xargs`, and `cp`. In this scenario, there is a possibility that misdetection may occur. However, a false negative does not occur.

According to the above results, we can say that the VMM-based tracing function traces the diffusion of classified information accurately. Further, the VMM-based tracing function causes no false negative. Even if the function detects an information leak excessively, it is important that no information leak occurs.

5.3 Lines of Code

5.3.1 Evaluation Methods of Lines of Code

We count the logical LOC and the number of files modified for implementing the tracing function. The logical LOC is the number of coding lines excluding only line made by a symbol, whitespace, and comment. To count the logical LOC, we use LocMetrics[13]. Then, the counting target is the logical LOC-related file operation and process creation.

Further, the scale of the source code is a great variation in each implementation environment owing to the fact that the OS-based tracing function is implemented within the OS and the VMM-based tracing function is implemented within the VMM. Thus, we institute a counting target to directories that have the stored files modified for implementing the tracing function. In particular, the kernel, fs/, and init/ directory under Linux OS, and the x86/ directories under the KVM are treated as a counting target.

5.3.2 Comparative Result of Lines of Code

The result of counting logical LOC and the number of files modified for implement the tracing function is presented in **Table 2**. The amount of logical LOC of the VMM-based tracing function is 10 more lines than that of the OS-based tracing function. This is attributed to the function that collects the information from the OS, such as process information and file information. The difference in the rate of logical LOC is 0.90% and it can be said that this is extremely small.

Table 2 Comparison of logical LOC and the number of files modified for the tracing function.

	Logical LOC			Number of files		
	Total	Added	Rate (%)	Total	Added/Modified	Rate (%)
OS-based tracing function	47,222	763	1.61	101	14	13.9
VMM-based tracing function	35,555	894	2.51	49	10	20.4

As already said in Subsection 5.3.1, the modified files for implementing the OS-based tracing function are scattered in multiple directories. This is due to the fact that the OS-based tracing function is implemented by modifying each system call that is related to the diffusion of the classified information. In contrast, the VMM-based tracing function traces the diffusion of the classified information by hooking the entry point of the system call unitary. Thus, the modified files for implementing the VMM-based tracing function are localized in a single directory. Further, the total number of files modified for implementing the VMM-based tracing function is 10, and it is within the 70% as compared to that of the OS-based tracing function.

Thus, we can conclude that the VMM-based tracing function can be implemented only by slight addition and localizing the range of modification as compared to the OS-based tracing function.

5.4 Overheads

5.4.1 Evaluation Method of Overheads

The evaluation items are listed below. In this evaluation, we use the virtualized host shown in Table 1 and a physical host, which has the same environment as the environment for the virtualized host. Further, Fedora 18 Linux with kernel version 3.6.10 runs in each environment.

(1) System Call
We measure the performance values of the write(), read(), close(), and clone() system calls related to the diffusion of classified information and compare them with those of the OS-based tracing function measure in [2]. Then, the fork() system call is used for the create process in the environment of the OS-based tracing function. On the other hand, the clone() system call is used for the create process in the environment of the VMM-based tracing function. Therefore, we measure the performance of the clone() system call and compare it with the performance of the fork() system call measured in [2]. Moreover, the VMM-based tracing function hooks all system calls even if the system call is unrelated to the diffusion of classified information. Then, we measure the performance of the getpid() system call that is unrelated to the diffusion of classified information.

(2) Microbenchmark
We use LMbench[14] as a microbenchmark. To evaluate the influences on the performance of the basic functions of an OS, we measured the latency of it.

(3) Application
Performance degradation with the VMM-based tracing function will occur in each VM-Exit caused by the system call invocation. To evaluate the impact of

these overheads on the application programs, we measure the performance of building bzImage that issues a large number of read() and write() system calls.

5.4.2 System Call

Table 3 shows the overhead of system calls incurred by the OS-based tracing function (3-1) and by the VMM-based tracing function (3-2, 3). In Table 5-3, *Bare* shows the measurement prior to the introduction of the VMM-based tracing function and *Traced* shows the measurement after the introduction of the VMM-based tracing function. *Operation of managed files* and *Operation of unmanaged files* in *Traced* show the measurement conducted while operating the managed/unmanaged files as sensitive files. *Overheads* are calculated by using the following formula: (measurement in each environment – measurement before the introduction of the function in a virtualized environment).

The overhead of the write(), read(), close(), and clone() system calls is 426.16%, 644.44%, 742.69%, and 12.16%, respectively, and these values are relatively large. The actual measurement is 1.96–12.35 μs and is more 47,800 times than that of getpid() that is unrelated to the diffusion of classified information. When the VMM-based tracing function determines that the hooked system call is related to the diffusion of classified information, it hooks the SYSRET and obtains the system call's return value. Subsequently, it stores system call's arguments and returns the control to the guest OS. Moreover, the VMM-based tracing function obtains the OS information (e.g., inode number) by using the system call's arguments and

Table 3 Overhead of system calls incurred by the VMM-based tracing function (μs).

3-1 Overhead's rate of OS-based tracing function (%).

	Operation of unmanaged file	Operation of managed file
write (file)	5.24	33.80
read (file)	31.15	31.15
close (file)	32.35	38.24
fork	2.66	6.89
getpid	-	-

3-2 Results in real environment.

write (file)	1.24
read (file)	0.42
close(file)	0.69
clone	16.88
getpid	0.0078

3-3 Results in virtualized environment.

	Bare	Traced		Overheads)	
		Operation of unmanaged file	Operation of managed file	Operation of unmanaged file	Operation of managed file
write (file)	0.60	2.76	3.17	2.16 (359.06%)	2.57 (426.16%)
read (file)	0.30	2.25	2.26	1.95 (640.52%)	1.96 (644.44%)
close(file)	0.28	2.32	2.40	2.03 (714.58%)	2.12 (742.69%)
clone	101.60	108.84	113.95	7.24 (7.13%)	12.35 (12.16%)
getpid	0.0078	0.0079	0.0079	0.0038 (5.00%)	0.000041 (5.42%)

return values. It is suspected that this additional processing is the cause of the large overheads.

In contrast, the overhead of getpid(), which is unrelated to the diffusion of classified information, is 0.000041 μs and the rate of overhead is 5.14%, which is a relatively small value. When the VMM-based tracing function determines that the hooked system call is unrelated to the diffusion of classified information, it does nothing and the control is returned to the guest OS. Consequently, the overhead of the system call that is unrelated to the diffusion of classified information is relatively small.

To summarize, owing to the additional processing for tracing information, the overhead of the system calls that are related to the diffusion of classified information is large as compared to that of the other system calls.

5.4.3 Microbenchmark

Table 4 shows the latency of the basic functions of an OS measured by using LMbench. By comparing the measurement result in the real environment and after the introduction of the VMM-based tracing function in the virtualized environment, we find that the performance of *fork proc*, *exec proc*, and *sh proc* is influenced by about 499–3521 μs. On the other hand, by comparing the measurement results obtained before and after the introduction of the VMM-based tracing function in the virtualized environment, we find that the differences in these item's performance are within about 102–904 μs. Therefore, these items' overhead is mainly caused by virtualization and the overhead of the VMM-based tracing function related to these items is small compared to that of the virtualization.

Next, let us consider *null call*, *null I/O*, *stat*, *open clos*, and *sig inst*. By comparing the measurement result in the real environment and after the introduction of the VMM-based tracing function in the virtualized environment, we find that these items' performance is influenced by about 220–4300% and this is very large. This is because the measurement values in the real environment are small and the overhead ratio in the virtualized environment is relatively large. The actual measurement values are subsided to about 1–2 μs in terms of the impact on most items. However, the overhead of *open clos* is 5.44 μs, and this is large as compared to that of the other items. This is attributed to the fact that the open() and close() system calls are related to the diffusion of classified information and are heavily traced by the VMM-based tracing function.

Table 4 LMbench results (μs).

	Real	Virtualized environment		Overheads	
	environment	Bare	Traced	Real:Virt(Traced)	Virt (Bare):Virt (Traced)
null call	0.04	0.04	1.76	1.72 (4300.00%)	1.72 (4300.00%)
null I/O	0.09	0.09	2.33	2.24 (2488.89%)	2.24 (2488.89%)
stat	0.54	0.56	1.73	1.19 (220.37%)	1.17 (208.93%)
open clos	1.09	1.18	6.53	5.44 (499.08%)	5.35 (453.39%)
slct TCP	2.06	2.19	3.26	1.20 (58.25%)	1.07 (48.86%)
sig inst	0.09	0.12	1.15	1.06 (1177.78%)	1.03 (858.33%)
sig hndl	0.64	0.71	1.85	1.21 (189.06%)	1.14 (160.56%)
fork proc	62.2	459	561	498.8 (801.93%)	102 (22.22%)
exec proc	215	1253	1527	1312 (610.23%)	274 (21.87%)
sh proc	932	3549	4453	3521 (377.79%)	904 (25.47%)

In summary, the latency of the basic functions of an OS influenced by the VMM-based tracing function is relatively small. However, the performance of processing with system calls that are related to the diffusion of classified information declines.

5.4.4 Application

Table 5 shows the overhead rate of building bzImage in the OS-based tracing function (5-1) and the consumed time of that in the VMM-based tracing function (5-2, 3). *Managed file* : 0 and *Managed file* : 10 in Table 5 show the measurements in the case of not registering the management file and of registering the 10 management files. *Overheads* are calculated by following formula: (measurement after the introduction of the function − measurement before the introduction of the function).

As shown in Tables 5-1 and 5-3, the overhead rate of the OS-based tracing function is 0.46% and that of the VMM-based tracing function is 14.2%, which is about 30 times longer than that of the OS-based tracing function. Therefore, we suspect that building bzImage includes a large number of read() and write() system calls that have a large overhead, as shown in Table 3-3. Moreover, the overhead ratio of the system time is larger than that of the user time. The VMM-based tracing function hooks the SYSRET of the guest OS for collecting the OS information (e.g., inode number). Due to the SYSRET processing on the kernel land, the overhead of the system time is large. This also means that the overhead increases depending on the number of system call invocations.

It can be seen that the processing time in the case of registering the 10 management files is larger by about 15 s than that without the management files, as shown in Table 5-3. The VMM-based tracing function audits whether the system call treats the classified file for each system call invocation. To achieve the above audit, the VMM-based tracing function scans the list of classified files. This scan

Table 5 Overhead and time for building bzImage in each environment.

5-1 Overhead rate of OS-based tracing function (%).

	Managed file: 0	Managed file: 10
Real time	0.29	0.46
User time	0.14	-0.02
System time	2.5	3.7

5-2 Results in a real environment (s).

Real time	462.406
User time	413.260
System time	41.358

5-3 Results in a virtualized environment (s).

	Bare	Traced		Overheads	
		Managed file: 0	Managed file: 10	Managed file: 0	Managed file: 10
Real time	579.159	660.566	675.056	81.407 (14.1%)	95.897 (14.2%)
User time	473.940	489.827	498.350	15.887 (3.4%)	24.410 (4.9%)
System time	85.853	114.133	131.380	28.280 (32.9%)	45.527 (34.7%)

causes the lengthening of the processing time. Further, the VMM-based tracing function appends all the files that have the potential to be classified files to the list of classified files. Therefore, the false positive will generated. If the list of classified files is bloated by the false positive, the processing time will increase. Thus, reducing the false positive is the research task for performance improvement in the future.

In conclusion, the VMM-based tracing function largely affects the processing of AP. Hence, suppressing the overhead is a significant task for practical use.

6 Related Work

The prevention of information leaks and the tracing diffusion of classified information have been researched and become an important challenge in computer security. Our purpose includes the tracing diffusion of classified information and, logging this information to inform us of leakage. Thus, we introduce approaches that aim to trace or prevent the classified information from cloud computing bases to smartphones and compare them with the VMM-based tracing function. In addition, we compare the attributes of the VMM-based tracing function and the encryption of classified information, which play an important role in the protection of classified information.

Tightlip [5] is a privacy management system that swaps an original process for a dummy process, called " Doppelgangers, " when a process that has sensitive data attempts to write the data to the network. This protects the sensitive data from leakage because Doppelgangers does not itself contain sensitive data. Aquifer [6] prevents the unintended leak of information by limiting the application that can handle the sensitive data by using a policy that restricts host exportation. In addition, there are some methods to prevent the leakage of sensitive information through the use of virtualization technology. Filesafe [8] protects sensitive files by using VMM. The user sets the security policy, such as read only or not accessible, for the sensitive files beforehand. By enforcing the security policies using hypervisors, Filesafe can prevent sensitive files from unauthorized access. SVFS [9] runs a Normal VM that runs standard applications, an Admin VM for the purpose of system administration, and a DVM to store sensitive files for the other VMs. The sensitive files can be edited only by the Admin VM. Thus, it is possible to protect the sensitive files even if the Normal OS is compromised by an attacker. In addition, VOFS [15] only permits the user to view sensitive files by using SVFS. TightLip, Aquifer, SVFS, and VOFS are necessary for modifying the structure of the OS, and hence, the operational environment is limited. In contrast, the VMM-based tracing function can be introduced in various environments owing to the lack of necessity of modification to the OS. In addition, Filesafe necessitates the setting of the policy for each file individually. This possibly causes leakage of the classified information by policy misconfiguration. In contrast, the VMM-based tracing function automatically traces the diffusion of classified information, and therefore, the risk of information leakage by policy misconfiguration is low.

TaintEraser [16] is a method for tracing the diffusion of classified information by using a dynamic taint analysis (DTA). DTA tracks information that has been tainted by other data. Subsequently, if the tainted data are written to another location in the memory, this destination is marked as tainted. Thus, we can follow the classified information DTA. TaintDroid [17] uses a similar method. TaintDroid is implemented for smartphones and traces the diffusion of sensitive information within a mobile terminal. Further, when external leakage of information is detected, the user receives a notification. Taint-exchange [18] is a method of cross-host taint tracking. It is achieved by injecting taint information in data transfers. To mark taint-tag, a DTA needs additional storage, called shadow memory. Therefore, nontrivial additional memory and disk space is required. In contrast, the VMM-based tracing function can trace the diffusion of classified information without the additional memory and disk space. However, the VMM-based tracing function's tracing granularity is coarser than that of DTA because the VMM-based tracing function is based on the probability of information diffusion caused by the system calls.

Moreover, the opportunities for dealing with sensitive information on smartphones are increasing with the increasing popularity of smartphones. Therefore, smartphone security is being studied widely. AppIntent [19] detects the transmission of sensitive data by an Android application and notifies it to a user. Subsequently, in the case of an unintentional operation for the user, the application that executes the operation is judged to be the malicious one. DroidTrack [20] traces the diffusion of classified information by hooking the information-gathering API. When DroidTrack detects the possibility of information leakage, it notifies the user. Finally, if the user disallows the operation, information leakage is prevented by terminating the operation as an error. DroidSafe [21] provides a static information flow analysis framework. DroidSafe analyzes the information flow from the API, which has the potential to read sensitive data to the API, which has the potential to leak the sensitive data. From this analysis, we can verify whether an Android application has the potential to leak the sensitive data. Although these studies are targeted toward smartphones, the purpose of study is the same from the viewpoint of the protection of sensitive data.

When the sensitive data are leaked, the assurance for log integrity is important for analyzing the cause. The method of [7] gathers the logs by using Linux Security Module (LSM). In addition, the log integrity is guaranteed by using mandatory access control. The method of [22] and [23] gathers the logging information generated by the OS and the APs working on the target VM by the VMM. To gather the logging information by the VMM, the VMM hooks the system call that was invoked for sending logs from the user process to the syslog daemon. This method makes it difficult to tamper with a log by isolating it from the VM. NIGELOG [24] provides multiple backup of the log files. It enables the log files to restore by using backup files even if the original file is altered or deleted by intruders. LISS [25] backs up the log files by using a mirroring technique. Then, LISS verifies the integrity of the log files by comparing the hash value between the original file and the backup file. Similar to [22], the VMM-based tracing function ensures the reliability of the monitoring information by sheltering the tracing function from the target OS. Further, it seems that the reliability of the monitoring information can be improved by combining the VMM-based tracing function with the methods of [7][24][25].

Data encryption is one of the methods used for protecting sensitive information. Mimesis-Aegis [26] proposed a transparent encryption method for all applications. Mimesis-Aegis interposes between the application and the user. Then, Mimesis-Aegis showed the decoded data to the user and sent the encrypted data to the applications. The method of [27] is an encryption technique in the cloud storage environment that encrypts an index that represents whether the data are stored in any chunk server. PPS-RTBF [28] determines an access authority to privacy data on the Web. Thus, an unauthorized user cannot access the privacy data. In addition, PPS-RTBF ensures the security of the information by encryption. By encrypting the information, even if the sensitive information is leaked, it can reduce the possibility of the contents being read by an unauthorized person. However, modification of the existing software or introduction of encryption software is necessary for the encryption. However, it is not necessary to minutely track the file to be protected. In contrast, while the tracing processing is needed, the VMM-based tracing function can be implemented by modifying only the VMM.

7 Conclusion

In this paper, we described the design and implementation of a VMM-based tracing function for file operations and child process creation with a KVM. The VMM-based tracing function traces the diffusion of classified information from outside the OS. To trace the diffusion of classified information, hooking the system call to the guest OS and collecting the required information are necessary. Moreover, by obtaining the system call's return value, we can improve the tracing accuracy. The VMM-based tracing function is implemented without modifying the OS's source code. Therefore, we expect that the VMM-based tracing function can be introduced in various environments. Moreover, it is difficult to attack the function directly, owing to the isolation of the VMM from the OS. Furthermore, even if the kernel version is updated, the VMM-based tracing function will continue to be available, provided that the system call specifications and the data structure remain unchanged. In summary, the VMM-based tracing function resolves the problems of the existing method including the OS-based tracing function and can trace the diffusion of classified information.

We implemented and evaluated the prototype of the VMM-based tracing function. Then, we verified the traceability of the VMM-based tracing function. Moreover, we demonstrated that the VMM-based tracing function can be implemented only by slight addition and localizing the range of modification as compared to the OS-based tracing function. In the evaluation of the overhead of the system calls, we indicated that the overhead of getpid(), which was unrelated to the diffusion of classified information, was 0.000041 μs; this was a relatively small value.

On the other hand, the overhead of system calls that were related to the diffusion of classified information was in the range of 1.96–2.57 μs (426.16–742.69%) for file operations and was 12.35 μs (12.16%) for process creation; these values were large as compared to those obtained in the case of getpid(). The evaluation using microbenchmark revealed that the overhead of functions largely affected by virtualization was relatively small and that of the other functions was about 1–2 μs. Furthermore, the evaluation using a real-world application showed the VMM-based tracing function generated about 14.2% overhead for building bzImage.

In a future work, we will implement the VMM-based tracing function for interprocess communication.

References

1. Japan Network Security Association, 2008 Information Security Incident Survey Report, `http://www.jnsa.org/result/incident/data/2008incident_survey_e_v1.0.pdf`
2. Tabata T, Hakomori S, Ohashi K, Uemura S, Yokoyama K, Taniguchi H (2009) Tracing Classified Information Diffusion for Protecting Information Leakage. IPSJ Journal 50(9), pp. 2088–2102 (in Japanese)
3. Nomura Y, Hakomori S, Ohashi K, Yokoyama K, Taniguchi H (2006) Tracing the Diffusion of Classified Information Triggered by File Open System Call. Proc. 4th Int. Conf. on Computing, Communications and Control Technologies (CCCT 2006), pp. 312–317
4. Otsubo N, Uemura S, Yamauchi T, Taniguchi H (2013) Design and Evaluation of a Diffusion Tracing Function for Classified Information Among Multiple Computers. Lecture Notes in Electrical Engineering (LNEE), vol.240, pp. 235–242
5. Yumerefendi AR, Mickle B, Cox LP (2007) Tightlip: Keeping Applications from Spilling the Beans. Proceedings of the 4th USENIX Conference on Networked Systems Design and Implementation (NSDI '07), pp. 159–172
6. Nadkarni A, Enck W (2013) Preventing Accidental Data Disclosure in Modern Operating Systems. Proceedings of the 2013 ACM SIGSAC conference on Computer and communications security (CCS'13), pp. 1029–1042
7. Isohara T, Takemori K, Miyake Y, Qu N, Perring A (2010) LSM-Based Secure System Monitoring Using Kernel Protection Schemes. International Conference on Availability, Reliability, and Security, (ARES'10) pp.591–596
8. Junqing W, Miao Y, Bingyu L, Zhengwei Q, Haibing G (2012) Hypervisor-based Protection of Sensitive Files in a Compromised System. Proceedings of the 27th Annual ACM Symposium on Applied Computing (SAC'12), pp. 1765–1770
9. Zhao X, Borders K, Prakash A (2005) Towards protecting sensitive files in a compromised system. In Proc. Third IEEE International Security in Storage Workshop (SISW'05), pp. 21–28
10. KVM, `http://www.linux-kvm.org/page/Main_Page`
11. Fujii S, Yamauchi T, Taniguchi H (2015) Design of a Function for Tracing the Diffusion of Classified Information for File Operations with a KVM. The 2015 International Symposium on Advances in Computing, Communications, Security, and Applications for Future Computing (ACSA 2015)
12. Chen PM, Noble BD (2001) When Virtual Is Better Than Real. Proceedings of the Eighth Workshop on Hot Topics in Operating Systems, pp. 133–138
13. LocMetrics, `http://www.locmetrics.com/`
14. Larry M, Carl S (1996) Lmbench: Portable Tools for Performance Analysis. Proceedings of the 1996 Annual Conference on USENIX Annual Technical Conference, pp. 279–294
15. Borders K, Zhao X, Prakash A (2006) Securing Sensitive Content in a View-only File System. Proceedings of the ACM Workshop on Digital Rights Management, pp. 27–36
16. David YZ, Jung J, Song D, Kohno T, Wetherall D (2001) TaintEraser: Protecting Sensitive Data Leaks Using Application-level Taint Tracking. SIGOPS Oper. Syst. Rev. 45(1), pp. 142–154
17. Enck W, Gilbert P, Chun B, Cox LP, Jung J, McDaniel P, Sheth AN (2010) TaintDroid: An Information-flow Tracking System for Realtime Privacy Monitoring on Smartphones. Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation, pp. 1–6
18. Zavou A, Portokalidis G, Keromytis AD (2011) Taint-exchange: A Generic System for Cross-process and Cross-host Taint Tracking. Proceedings of the 6th International Conference on Advances in Information and Computer Security (IWSEC'11), pp.113–128
19. Yang Z, Yang M, Zhang Y, Gu G, Ning P, Wang, XS (2013) AppIntent: analyzing sensitive data transmission in android for privacy leakage detection. Proceedings of the 2013 ACM SIGSAC conference on Computer and communications security (CSS'13), pp. 1043–1054
20. Sakamoto S, Okuda K, Nakatsuka R, Yamauchi T (2014) DroidTrack: Tracking and Visualizing Information Diffusion for Preventing Information Leakage on Android. Journal of Internet Services and Information Security 4(2), pp. 55-69
21. Gordon MI, Kim D, Perkins J, Gilham L, Nguyen N, Rinard M (2015) Information-Flow Analysis of Android Applications in DroidSafe. 22nd Annual Network and Distributed System Security Symposium (NDSS 2015)
22. Sato M, Yamauchi T (2012) VMM-Based Log-Tampering and Loss Detection Scheme. JIT 13(4), pp. 655–666

23. Sato M, Yamauchi T (2014) Secure and Fast Log Transfer Mechanism for Virtual Machine. Journal of Information Processing 22(4), pp. 597–608
24. Takada T, Koike H (1999) NIGELOG: protecting logging information by hiding multiple backups in directories. Proceedings. Tenth International Workshop on Database and Expert Systems Applications, pp. 874–878
25. Joo JW, Park JH, Suk SK, Lee DG (2014) LISS: Log Data Integrity Support Scheme for Reliable Log Analysis of OSP. The Journal of Convergence 5(2), pp.1–5
26. Lau B, Chung S, Song C, Jang Y, Lee W, Boldyreva A (2014) Mimesis Aegis: A Mimicry Privacy Shield–A System's Approach to Data Privacy on Public Cloud. 23rd usenix security symposium (USENIX Security 14), pp. 33–48
27. Lee SH, Lee IM (2013) A Secure Index Management Scheme for Providing Data Sharing in Cloud Storagepp. Journal of Information Processing Systems 9(2), pp. 287–300
28. Lee JD Sin CH, Park JF (2014) PPS-RTBF: Privacy Protection System for Right To Be Forgotten. The Journal of Convergence 5(3), pp. 37–40