Electronic Preprint for Journal of Information Processing Vol.24 No.5

Recommended Paper

Design of Function for Tracing Diffusion of Classified Information for IPC on KVM

Shota Fujii[1] Masaya Sato[1,a)] Toshihiro Yamauchi[1,b)] Hideo Taniguchi[1,c)]

Received: December 3, 2015, Accepted: June 2, 2016

Abstract: The leaking of information has increased in recent years. To address this problem, we previously proposed a function for tracing the diffusion of classified information in a guest OS using a virtual machine monitor (VMM). This function makes it possible to grasp the location of classified information and detect information leakage without modifying the source codes of the guest OS. The diffusion of classified information is caused by a file operation, child process creation, and inter-process communication (IPC). In a previous study, we implemented the proposed function for a file operation and child process creation excluding IPC using a kernel-based virtual machine (KVM). In this paper, we describe the design of the proposed function for IPC on a KVM without modifying the guest OS. The proposed function traces the local and remote IPCs inside the guest OS from the outside so as to trace the information diffusion. Because IPC with an outside computer might cause information leakage, tracing the IPCs enables the detection of such a leakage. We also report the evaluation results including the traceability and performance of the proposed function.

Keywords: information leakage prevention, inter-process communication, virtualization

1. Introduction

With the increase in the use of personal computers, the ability to process classified information using a computer has also increased. Associated with this, the leakage of classified information to an outside computer has become a serious problem. According to a personal information leakage analysis [1], a leakage often occurs as a result of inadvertence and mismanagement, which accounts for approximately 57% of all known leakage cases. To prevent information leakage, it is important for the user to grasp the situation surrounding the classified information. In addition, cyber-attacks aiming at the theft of classified information have become increasingly sophisticated. Therefore, it is difficult to completely prevent such attacks, and it has become important to reduce the amount of damage to users by detecting the transfer of classified information outside their computer [2].

To trace the status of classified information in a computer, and manage the resources that contain such information, we developed an operating system (OS) based function for tracing the diffusion of classified information [3] (particularly, an OS-based tracing function). In addition, the proposed function visualizes the diffusion using a directed graph [4], and traces the diffusion of the classified information on multiple computers [5]. Such functions are efficient for grasping the use situation and preventing the leakage of classified information.

On the other hand, an OS-based tracing function is executed within the OS, and therefore, has the potential of being detected and disabled by an adversary or a malicious user. If the OS-based tracing function is disabled, the victim cannot detect the information leakage and there is a risk of increased damage. In addition, the function cannot be installed in a closed-source OS such as Windows, because its implementation requires a modification of the source code.

Similar to an OS-based tracing function, a large number of methods for protecting sensitive files have been proposed [6], [7], [8], [9]. However, because they are implemented within the OS, there is a problem in that the operational environment is limited, and they can be detected and disabled, similar to an OS-based tracing function. To resolve the above mentioned problems, methods for protecting sensitive files from outside the OS have been proposed [10], [11]. These methods demonstrate the effectiveness of an outside OS implementation of a security system. However, it is difficult to identify the cause of information leakage because these methods are aimed solely at information leakage prevention and do not trace the diffusion of classified information.

Based on the above mentioned observations, we designed a function for tracing the diffusion of classified information in a guest OS using a virtual machine monitor (VMM) specifically, a VMM-based tracing function [12]. This VMM-based tracing function provides the guest OS with functions that are equivalent to an OS-based tracing function, without the need to modify the source code of the guest OS. It is expected that attacks specifically targeting this function will be difficult to achieve because a VMM is more robust than an OS. There are three paths of in-

1 Graduate School of Natural Science and Technology, Okayama University, Okayama 700–8530, Japan
a) sato@cs.okayama-u.ac.jp
b) yamauchi@cs.okayama-u.ac.jp
c) tani@cs.okayama-u.ac.jp

The preliminary version of this paper was published at Computer Security Symposium 2015 (CSS2015), October 2015. The paper was recommended to be submitted to Journal of Information Processing (JIP) by the chief examiner of SIGCSEC.

© 2016 Information Processing Society of Japan

formation diffusion: file operations, child process creation, and Inter-process communication (IPC). In a previous work, we implemented and evaluated the proposed VMM-based tracing function for a file operation and child process creation through a kernel-based virtual machine (KVM) [13].

IPC is an important diffusion path of classified information. However, this function for IPC is not implemented because IPC tracing is a complex process. IPC is conducted through a communication medium and includes the possibility for communication outside the computer; thus, it may be made untraceable through a method similar to a file operation and child process creation. Thus, a new tracing function for IPC is needed to grasp all diffusion paths of classified information. Additionally, IPC using a socket is applied not only for local IPC, but also for remote IPC. During remote IPC, communication with an outside computer might cause information leakage. An example of such information leakage is the improper transmission of an e-mail. At this time, when a personal information is leaked, there is the possibility of suffering damage such as an unauthorized login. Additionally, in a company, information leakage is likely to cause damages to trust. Therefore, IPC tracing has a higher level of importance. A system call monitoring method used by VMwatcher is proposed in Ref. [14]. This method can monitor all system calls invoked in VMs. However, it is insufficient to trace the diffusion of classified information because it requires a large cost for analyzing the diffusion path. Since the fast detection of information leakage is preferable, the cost for tracing information diffusion must be reduced.

This paper describes the design and implementation of a VMM-based tracing function for IPC using a socket communication through a KVM. Further, this design and implementation are tailored for a 64-bit version of Linux as a guest OS. This paper also describes an evaluation including the traceability and performance of the VMM-based tracing function.

The contributions of this paper are as follows:

- We point out the IPC-related path of diffusion and the leakage of classified information, as well as their related system calls.
- We propose a VMM-based function for tracing the diffusion and leakage of classified information through a socket communication. This has the following two functions: 1) tracing the diffusion of classified information inside the guest OS and 2) detecting a leakage outside the computer.
- We evaluate and report the traceability and performance of the proposed VMM-based tracing function for IPC. The evaluation of traceability shows that the VMM-based tracing function is able to trace both the local and remote IPC. In addition, the performance evaluation shows that the performance degradation of the application is small.

The remainder of this paper is constructed as follows: Section 2 presents an overview of the function for tracing the diffusion of classified information. Sections 3 and 4 discuss the design and implementation of the function for tracing the diffusion of classified information for IPC on a KVM, respectively. Section 5 describes an evaluation of the proposed function. Section 6 discusses previous studies related to this topic. Finally, Section 7 provides some concluding remarks regarding this research.

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

2.1 Classified Information Diffusion Path

The OS-based tracing function [3] manages any files or processes that have the potential to diffuse classified information. Classified information can be diffused through any process that involves opening a classified file, reading its contents, 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) Child process creation
(3) Inter-process communication

An OS-based tracing function traces the diffusion of classified information by monitoring the system calls related to such operations.

2.2 Purpose

As described in Section 1, the leakage of classified information often occurs as a result of inadvertent handling and mismanagement (e.g., the improper transmission of an e-mail). This is attributed to the user's inability to grasp the location of classified information. In addition, it is difficult to completely prevent the leakage of such information. Therefore, when information leakage occurs, it is important to detect the event and grasp its cause. Based on the above background, the tracing function aims at achieving the following.

(Purpose 1) Tracing the diffusion of classified information inside the computer.

(Purpose 2) Detecting the leakage of classified information to the outside of the computer and recording its cause.

2.3 Overview of the OS-based Tracing Function

We proposed an OS-based tracing function [3], an overview which is shown in **Fig. 1**. This OS-based tracing function traces the diffusion of classified information as follows:

(1) The OS-based tracing function hooks system calls that are related to the diffusion of classified information.
(2) The OS-based tracing function collects information for trac-

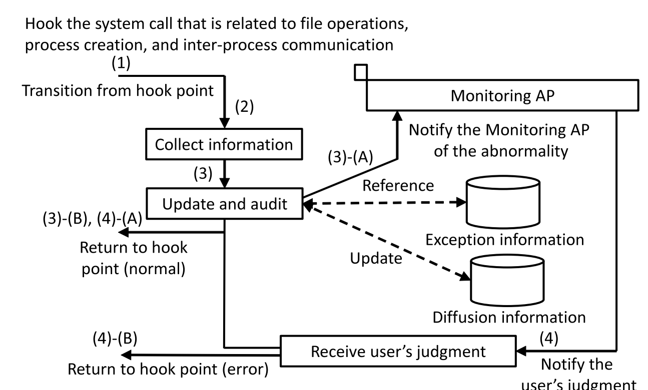


Fig. 1 Overview of the OS-based tracing function.

© 2016 Information Processing Society of Japan

ing the diffusion of classified information such as files handled by a system call or the transmission-destination process.

(3) The OS-based tracing function updates the diffusion information using the information collected during (2), and audits the potential leakage of classified information.
  (A) If the audit discovers the possibility of a leakage of classified information, the OS-based tracing function notifies such leakage to the monitoring application program (AP).
  (B) If the audit does not detect any possibility of such leakage, the OS-based tracing function returns control to the system call.
(4) After receiving the results of the user's judgment based on the monitoring AP, the OS-based tracing function controls the system call accordingly as follows:
  (A) If the user's judgment is affirmative, the system call processing is continued.
  (B) If 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.

2.4 Overview of the VMM-based Tracing Function

The OS-based tracing function meets the purpose described in Section 2.2. However, this function has certain problems as follows:

(Problem 1) 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.

(Problem 2) The OS's source code must be modified before it can be installed.
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 implemented 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.

Further, as described in Section 1, there are similar problems in the existing methods for protecting sensitive files. To resolve these problems, we proposed a VMM-based tracing function [12], which is functionally equivalent to an OS-based tracing function. In particular, the VMM-based tracing function manages any files or processes that have 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 the three operations described in Section 2.1. The user can always grasp the location of their classified information using the list of classified information stored in the VMM. Furthermore, when a diffusion of classified information is detected, the VMM-based tracing function records the pathname of the destination file, inode number, name of the command causing diffusion, and the process ID (PID). Therefore, the user can detect the information leakage using the above information and suppress the damage even if a leakage has occurred.

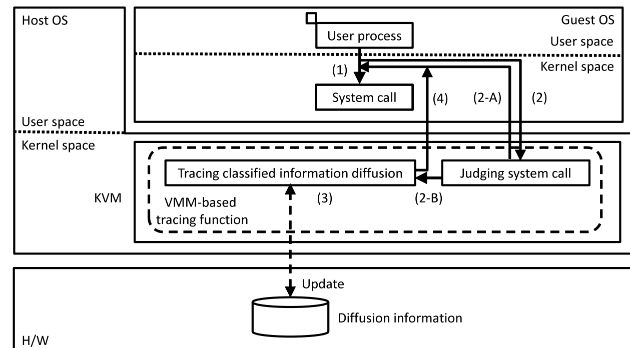


Fig. 2 Overview of the VMM-based tracing function.

Figure 2 shows an overview of the VMM-based tracing function. The VMM-based tracing function traces the diffusion of 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 processing is conducted.
  (A) When a hooked system call is unrelated to the diffusion of classified information, control is returned to the guest OS and the system call processing is continued.
  (B) When a 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) The VMM-based tracing function updates the diffusion information using the information collected in (2-B).
(4) Control is returned to the guest OS and the system call processing is continued.

Given these steps, the VMM-based tracing function provides the guest OS with functions equivalent to an OS-based tracing function. Furthermore, the user registers the inode number of the files including classified information to the VMM beforehand.

3. Tracing the Diffusion of Classified Information for IPC on KVM

3.1 Target Tracing

We described a method for tracing the diffusion of classified information using (1) a file operation and (2) child process creation [12]. Next, this paper describes a method for tracing the diffusion of classified information through (3) IPC using the VMM-based tracing function.

IPC is implemented through various methods, for example, a pipe, FIFO, message queue, shared memory, or socket. Among these, IPC using a socket is applied not only for local IPC, but also for remote IPC. Remote IPC has the possibility of classified information leaking outside the user's computer. Therefore, tracing remote IPC has a higher level of importance.

IPC tracing on KVM is more complex and more difficult than tracing file operation or child process creation because KVM

© 2016 Information Processing Society of Japan

Table 1 System call utilized for socket communication. The relevant parts are marked with "×."

Category	Name of system call	Possibility of diffusion	Possibility of leakage	Necessity for tracing
Setup	socket			
	bind			
Server	listen			
	accept			
Client	connect			
Output	sendto	×	×	×
	sendmsg			
	write			
	sendfile			
Input	recvfrom	×		×
	recvmsg			
	read			
Termination	shutdown			×
	close			

must grasp relations between sockets and processes. It requires more information than the case in file operation and process creation. File operations and child process creation are traceable by just associating source file/process to destination process/file. However, on IPC using a socket, KVM must understand which socket corresponds to which process. Finding the corresponding socket from the process control block of a current process requires the analysis of many data structures and its cost is quite high. Finding the receiver socket corresponding to the sender socket also requires a negligible data structure analysis. In addition, IP addresses and port numbers used in the socket communication must be acquired and managed by KVM. Collecting and managing the information makes it difficult to trace the information diffusion. Generally, monitoring data structures inside a VM from the outside causes a semantic gap problem. Thus, the IPC tracing on KVM is more difficult than file operation and process creation.

In this paper, we describe a method for tracing IPC using a socket.

3.2 Tasks

To detect the diffusion and leakage of classified information through IPC using the VMM-based tracing function, the following tasks are required:

(Task 1) Hooking a system call with a VMM.

(Task 2) Detecting the diffusion and leakage of classified information through socket communication.

(Task 3) Obtaining the necessary information for tracing a socket communication.

To trace a socket communication using a VMM, it is necessary to hook the system call related to the socket communication, and use it to detect the diffusion and leakage of classified information. Furthermore, tracing the socket communication from the VMM is complex and difficult when using a similar method to trace a file operation or child process creation because the socket intermediates the communication and is used for the communication with an outside computer. It is therefore necessary to manage the socket, which is used as an IPC communication path, and detects the propagation of classified information. When the VMM-based tracing function detects the socket that have the potential to diffuse classified information, it registers this socket to a list (managed socket list). The managed socket list is a list managing the sockets that have a potential to diffuse classified information. It is one of the diffusion information (shown in Fig. 2) which manages the classified information. Therefore, the VMM-based tracing function can audit whether the socket has classified information by referring to the managed socket list.

We accomplished (Task 1) in Ref. [12]; therefore, we focus on (Task 2) in the present paper. To accomplish (Task 2), the VMM-based tracing function hooks the system calls, which causes the diffusion and leakage of classified information through a socket communication. For (Task 3), we describe the solutions to each socket communication, i.e., local IPC and remote IPC.

3.3 Detecting the Diffusion and Leakage of Classified Information through a Socket Communication

Table 1 shows the system calls used for a socket communication along with its name, category, possibility of classified information diffusion and information leakage, and tracing necessity. Table 1 shows a case for Linux 3.6.10, which differs depending on the kernel version.

The diffusion and leakage of classified information do not occur during the socket creation or the establishment of a connection because the data cannot be exchanged. Therefore, from a socket creation to the establishment of a connection, the VMM-based tracing function does not trace the processed system calls.

The diffusion of classified information occurs when the data communication by an *Output*/*Input* system call is actually conducted. Thus, the VMM-based tracing function traces *Output*/*Input* system calls. In addition, an actual information leakage occurs through the transmission of data outside the computer using an *Output* system call. Therefore, the VMM-based tracing function audits the potential for leaking classified information when it hooks to an *Output* system call.

Furthermore, the terminated socket is excluded from the managed socket list because it remains unused.

3.4 Obtaining the Necessary Information for Tracing a Socket Communication

3.4.1 Local IPC

In local IPC through a socket communication, the VMM-based tracing function hooks an *Output*/*Input* system call and identi-

© 2016 Information Processing Society of Japan

fies the socket of the communication medium. At that time, the VMM-based tracing function traces the diffusion of classified information through IPC by managing whether the socket can obtain this information.

3.4.2 Remote IPC

During remote IPC using socket communication, the VMM-based tracing function hooks the *Output* system call and detects information leakage. Then, by identifying the process and computer at the destination of the communication, the VMM-based tracing function obtains its IP address and port number.

3.5 Flow of Tracing Socket Communication

Solving these tasks enables the VMM-based tracing function to trace the diffusion of classified information through a socket communication. The flow of the socket communication tracing is as follows:

(1) The VMM-based tracing function hooks the system call related to the *Output* in the guest OS from the VMM.
(2) When a process issuing the *Output* system call is managed, the following processing is conducted.
  (A) In the case of local IPC, the socket is appended to the managed socket list.
  (B) In the case of remote IPC, the possibility of information leakage is determined, and the VMM-based tracing function notifies the user as such by producing a log.
(3) Control is returned to the guest OS and the *Output* system call process is continued.
(4) The VMM-based tracing function hooks the system call related to the *Input* in the guest OS from the VMM.
(5) When the socket utilized for IPC is managed, the process issuing the *Input* system call is appended to the managed process list.
(6) Control is returned to the guest OS and the *Input* system call process is continued.

4. Implementation

4.1 Environment

In this section, we describe the implementation of the VMM-based tracing function using a KVM as the VMM and 64-bit Linux 3.6.10 as the guest OS. In addition, the system calls in the guest OS is executed through SYSCALL/SYSRET. Further, the guest OS is fully virtualized using Intel Virtualization Technology (VT).

4.2 Requirements and Tasks for Implementation

Based on the design described in Section 3, the following requirements must be implemented;

(Requirement 1) Tracing the diffusion of classified information using local IPC

(Requirement 2) Detecting the leakage of classified information using remote IPC

(Requirement 3) Tracing the diffusion and detecting the leakage of classified information using only information obtainable by the VMM

To trace the socket communication, it is necessary to trace the diffusion of classified information using local IPC and detect the information leakage through remote IPC, as described in Section 3. In addition, the VMM-based tracing function can only use the information obtainable from the VMM owing to its VMM implementation.

Under Linux, there are two ways for local IPC through a socket communication: communication of UNIX domain and that of INET domain to the loopback address or its host address. In addition, remote IPC using a socket communication is achieved through a communication with an external IP address using the INET domain.

There are two tasks required to meet the above requirements using a VMM-based tracing function:

(Implementation Task 1) Tracing the socket communication through a UNIX domain socket using a VMM

(Implementation Task 2) Tracing the socket communication through an INET domain socket using a VMM

Even though existing approaches can monitor all system calls invoked in a VM and data structures related to those system calls, it is insufficient for efficient tracing of information diffusion. To suppress the amount of information collected by a VMM and manage resources that possibly have classified information, we develop the proposed system which can manage those resources and quickly detect information leakage.

To trace IPC using a socket, it is necessary to associate two sockets first because there are sender socket and receiver socket. Subsequently, the communicated socket is made to correspond to another socket. In order to do this, after identifying the sender socket, identify the socket corresponding to receipt for each Input system call and audit whether this socket communicates with sender socket are needed. The corresponding process is also acquired by tracing data structures from the corresponding socket.

To bridge the semantic gap, first of all, the virtual address of the information to fetch must be obtained by arguments and return value, or stack pointer when the system call is hooked. Subsequently, we can obtain the target information by translating guest virtual address into host physical address using shadow page table. As shown in **Figs. 3** and **4**, there are many data structures related to the socket. When a send or a receive event occurs, the analysis of all data structures in Fig. 3 is required. In addition, when a receive event occurs, analysis of all data structures in Fig. 4 is required. This analysis enables the KVM to bridge the semantic gap.

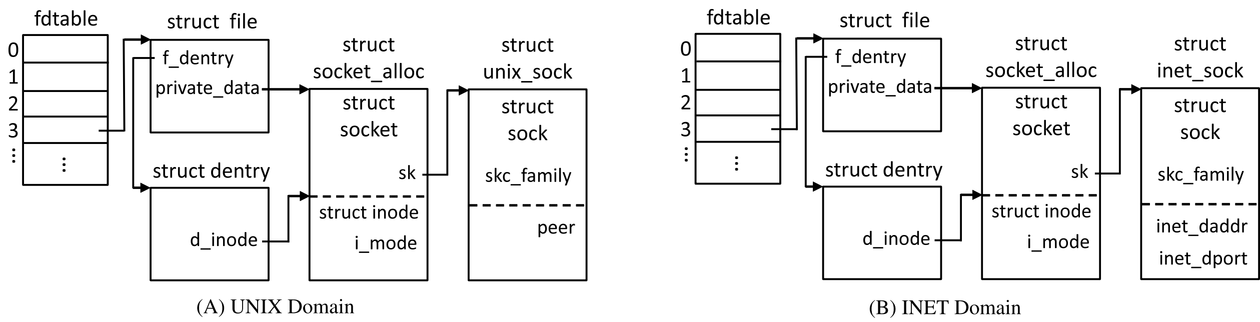
The proposed method hooks system calls of guest OSes by a method stated in the paper [12]. After system calls are hooked, all tracing functions are implemented separately from the original VMM's source codes. Thus, the insertion to existing source codes is kept to a minimum. Moreover, additional functions are modularized. For these reasons, the cost for implementing the proposed method to existing VMMs is kept low.

In Sections 4.3 and 4.4, we describe the procedures for accomplishing the above tasks, respectively.

4.3 Tracing the Socket Communication through a UNIX Domain Socket Using a VMM

As described in Section 3.4.1, to trace the local IPC, the VMM-based tracing function manages whether the socket handled by

© 2016 Information Processing Society of Japan



(A) UNIX Domain (B) INET Domain

Fig. 3 Relationship of data structures related to a socket.

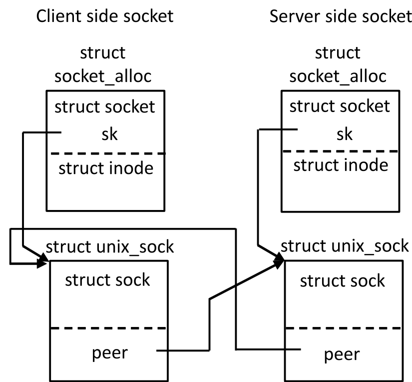


Fig. 4 Relationship between data structures of a connected UNIX domain socket.

the *Output*/*Input* system call can obtain classified information. Figure 3 (A) shows the relationship of the data structure related to the UNIX domain socket. The UNIX domain socket is implemented using the `socket` and `unix_sock` structures. As the figure shows, data of UNIX domain socket are obtainable by tracing the data structures from the file descriptor (fd). The VMM-based tracing function identifies the socket handled by a system call when it hooks the *Output*/*Input* system calls using the fd obtained from their argument.

In addition, the classified information is diffused to a destination socket through a source socket. Therefore, after identifying the socket handled by the *Output* system call, the VMM-based tracing function also identifies the destination socket. Both socket connections are established through a connection request by the client-side connect() system call to the server-side listening socket using an accept() system call. It can be seen in Fig. 4 that the destination socket is identifiable by a `peer` of the `unix_sock` structure. The VMM-based tracing function then identifies the destination socket from the `peer` of the `unix_sock` structure by tracing the socket data structure.

Further, to exclude the terminated socket from the managed socket list, the VMM-based tracing function hooks the *Termination* system call and identifies the socket handled by this call.

4.4 Tracing the Socket Communication through an INET Domain Socket Using a VMM

In the communication of the INET domain, unlike that of the UNIX domain, both sockets are not always created in the computer. Accordingly, to identify the communication destination, the VMM-based tracing function utilizes the IP address and port number of the destination socket. Figure 3 (B) shows the relationship of the data structure related to the INET domain socket. Similar to the UNIX domain socket, the INET domain socket is implemented through the `socket` and `inet_sock` structures. Therefore, they are identifiable by tracing the data structure using fd.

In addition, the IP address and port number of the destination socket are stored in the `inet_daddr` and `inet_dport` of the `inet_sock` structure, respectively. The VMM-based tracing function then obtains the values from the data structure of the socket by hooking the *Output*/*Input* system calls. When the destination IP address is a loopback address or the user's own host address, and when the destination port and source port are matched, the VMM-based tracing function judges that the communication between both sockets is achieved and traces the diffusion of classified information. When the destination of the *Output* system call is not a loopback address or own host address, the VMM-based tracing function judges it as an external communication and detects a possibility of information leakage.

5. Evaluation

5.1 Experimental Setup

We evaluated the following four items.

(1) Traceability
(2) System Call Overhead
(3) Microbenchmark
(4) Application

During the socket communication, there is the possibility of the diffusion of classified information through a local IPC and its leakage through a remote IPC. Therefore, the VMM-based tracing function is required for detecting the above two patterns. We then audit whether the VMM-based tracing function can detect them by conducting the assumed scenario in the guest OS. In addition, to audit the performance degradation of the VMM-based tracing function, we measured the performance of the system call, the microbenchmark, and the application.

Further, we evaluated the VMM-based tracing function using Core i5-3470 (3.2 GHz, 4 cores) and 4,096 MB of memory. One virtual CPU and 1,024 MB of memory are allocated to the guest OS. The prototype of the VMM-based tracing function is implemented by modifying KVM (kvm-kmod-3.6) with Linux 3.6.10 and the guest OS is the same version of the host, Linux 3.6.10. The number of modified files for implementing the VMM-based

© 2016 Information Processing Society of Japan

tracing function is seven files. In addition, the amount of modified source code is 1,458 lines. During the performance evaluation of the application in Section 5.5, we use the server and client machines as mentioned above. Additionally, the EPT is disabled.

5.2 Traceability

5.2.1 Traceability Evaluation Method

To evaluate the traceability of the VMM-based tracing function, we applied the following scenarios:

(Assumed Scenario 1) Send the classified information through local IPC using a UNIX domain socket.
Conduct local IPC through the UNIX domain socket (using sendto()/recvfrom()) from the managed client to an unmanaged server.

(Assumed Scenario 2) Send the classified information outside the computer.
Send the managed file outside the computer using a `scp` command.

(Assumed Scenario 3) Send an e-mail with an attached file which include classified information.
Send the e-mail with managed file using a `mail` command.

5.2.2 Experimental Results of Traceability

During local IPC, the classified information is diffused to a socket when the managed process sends it there. Next, the classified information is also diffused to the other process by receiving it from the managed socket. When (Assumed scenario 1) is executed, the VMM-based tracing function detects the diffusion of classified information to the socket and appends it to the managed socket list when it hooks a sendto(). In addition, the VMM-based tracing function detects the diffusion of classified information from the socket to process and append it to the managed process list when it hooks a recvfrom().

During remote IPC, the classified information is leaked when the managed process sends it outside the computer. When (Assumed scenario 2) executed by a `scp` command, the VMM-based tracing function detects the information leakage when it hooks a write() system call that sends the classified information outside the computer. **Figure 5** shows the log generated by the VMM-based tracing function when it detects information leakage. As shown in Fig. 5, the VMM-based tracing function records the destination IP addresses (`daddr`), port numbers (`dport`), command names (`comm`), and PIDs (`pid`). The user can identify the cause of the information leakage from such information. Correspondingly, the VMM-based tracing function detects the information leakage and records the equivalent information illustrated in Fig. 5 when (Assumed Scenario 3) is executed.

To summarize, we confirmed that the VMM-based tracing function is able to trace the diffusion of classified information through local IPC and detect information leakage through remote IPC.

```
write()
sensitive data is leaked.
daddr:     172.21.16.97
dport:               22
comm:               ssh
pid:               1298
```

Fig. 5 Log generated by the VMM-based tracing function in (Assumed Scenario 2).

5.3 System Call Overhead

5.3.1 Methods for Evaluating the System Call Overhead

To measure the overhead generated through the installation of the VMM-based tracing function, we measured the performance of the guest OS both before and after installation. Moreover, for comparison, we measured the performance when the process was both managed and unmanaged. In this evaluation, we measured the overhead of sendto() and recvfrom() of the UNIX domain, which is related to the diffusion of classified information. On the other hand, the VMM-based tracing function hooks all system calls even if a system call is unrelated to the diffusion of classified information. We then measured the performance of getpid() to clearly determine the impact of the performance of the system calls unrelated to the diffusion of classified information. To measure the performance of each system call, we utilized the rdtsc (Read Time-Stamp Counter) instruction, which obtains the value of timestamp counter. By comparing the value of timestamp counter obtained before and after the system call, we can calculate the time taken for executing the system call.

5.3.2 Experimental Results of the System Call Overhead

Table 2 shows the overhead of the system calls incurred through the VMM-based tracing function. In Table 2, *Bare* shows the measurement prior to the installation of the VMM-based tracing function and *Traced* shows the measurement after its installation. *Operation of managed target* and *Operation of unmanaged target* in *Traced* show the measurements conducted while operating a managed/unmanaged process or socket. In addition, *Overhead* is calculated using the following formula: (*measurement in each environment – measurement before the installation of the function*).

The overhead of sendto() and recvfrom() during the operation of an unmanaged target is 23.76 and 44.77 μs, respectively, with a relative performance level of 283.83% and 460.36%, whereas during the operation of a managed target it is 34.18 and 49.58 μs, respectively, with a relative performance level of 364.48% and 499.10%. These are relatively large values. The overhead during the operation of a managed target is greater than that of an unmanaged target because the managed process/socket list is scanned when judging whether the process/socket is managed. It seems that the overhead incurred after the installation of the VMM-

Table 2 Overhead of system calls incurred by the VMM-based tracing function (μs).

Name of system call	Bare	Traced (Relative performance)		Overhead	
		Operation of unmanaged target	Operation of managed target	Operation of unmanaged target	Operation of managed target
sendto	12.92	36.68 (283.83%)	47.10 (364.48%)	23.76	34.18
recvfrom	12.42	57.19 (460.36%)	62.00 (499.10%)	44.77	49.58
getpid	0.015	0.016 (106.86%)	-	0.0011	-

© 2016 Information Processing Society of Japan

based tracing function will be increased with the number of managed processes and sockets. Furthermore, when the process that issued the *Output* system call is managed, the VMM-based tracing function appends the destination socket to the managed socket list. In addition, when the socket handling the *Input* system call is managed, the VMM-based tracing function appends the process that issued the *Input* system call to the managed process list. It is suspected that this additional processing is the cause of the overhead when operating the managed target.

On the other hand, the overhead of getpid(), which is unrelated to the diffusion of classified information, is 0.0011 μs, with a relative performance level of 106.86%, which is a relatively small value. In the case of system calls unrelated to the diffusion of classified information, the performance impact is slight because the VMM-based tracing function only judges whether the hooked system call is related to the diffusion of classified information.

5.4 Microbenchmark

5.4.1 Evaluation Method of Microbenchmark

In this evaluation, we measure the latency of the IPC using LMbench [15] version 3. We then measured the performance of the guest OS both before and after installing the VMM-based tracing function as well as the evaluation described in Section 5.3.

5.4.2 Experimental Results of Microbenchmark

Table 3 shows the IPC latency measured by LMbench. Here, `slct TCP`, `AF UNIX`, `UDP`, `TCP`, and `TCP conn` are the total latency of the socket read/write, the communication standby time of the socket, the communication standby time through UDP/IP, the communication standby time through TCP/IP, and the total time between the socket creation and connection establishment respectively.

The overhead of `slct TCP`, `AF UNIX`, `UDP`, and `TCP` used to conduct the transmission/reception processing is within the range of 5 to 50 μs, which is a relative performance level of 327.88 to 546.55%. It seems that such overhead is caused by the process used to determine whether the process/socket is managed as by the evaluation described in Section 5.3. In addition, the overhead of `TCP conn` is 52.67 μs, which is the largest value in our analysis. It is thought that the cause of this overhead is the large number of VM-Exits incurred by issuing a system call between the socket creation and a connection establishment. On the other hand, the relative performance is 165.57%, and the ratio of increased overhead is relatively small compared to the other determined value. This is attributed to the system calls issued between the socket creation and a connection establishment being unrelated to the diffusion of classified information.

Table 3 Latency of IPC (μs).

Item	Bare	Traced (Relative performance)		Overhead
slct TCP	2.08	6.82	(327.88%)	4.74
AF UNIX	7.24	39.57	(546.55%)	32.33
UDP	13.33	63.23	(474.34%)	49.90
TCP	16.77	60.50	(360.76%)	43.73
TCP conn	80.33	133.00	(165.57%)	52.67

5.5 Web Server

5.5.1 Evaluation Method of Web Server

To evaluate the overhead on the application, we measure the average response time of a web server by using ApacheBench, version 2.3. We use the web server for the evaluation of the application since the VMM-based tracing function monitors a socket communication and a socket communication occurs frequently in the processing of a web server. In this evaluation, we use Apache 2.4.6 as the web server. This web server runs on the guest OS of the VMM-based tracing function and provides web pages with 1 Gbps network. To measure the average response time of the web server, the client machine sends an HTTP request 1,000 times to the web server by using ApacheBench and calculates its average. The ApacheBench runs on the client machine described in Section 5.1. In addition, for comparison, we measure the average response time before and after the installation of the VMM-based tracing function. In measurement after the installation, we measure the average response time both when the index.html is managed and unmanaged.

5.5.2 Experimental Results of Web Server

Table 4 shows the overhead and average response time of the Web server. As shown in Table 4, the relative performance of *Operation of unmanaged target* is within the range of 103.141 to 107.506% and that of *Operation of managed target* is within the range of 103.926 to 107.852%. In other words, the performance degradation with the VMM-based tracing function is less than about 10%. This is a very small value as compared to the overhead of the system call (maximum 499.10%) and that of the microbenchmark (maximum 546.55%). Although the overhead percentage of the system call and the microbenchmark is relatively large, that of the total processing time of the application is small. Moreover, the overhead in case *Operation of unmanaged target* is larger than that of *unmanaged target*. This is attributed to an additional processing for a managed file and a process as well as the evaluation in Section 5.3.

Furthermore, the overhead per HTTP request is within the range of 130 to 2,486 μs in *Operation of unmanaged target* and is within the range of 136 to 3,512 μs in *Operation of managed target*. These overheads may include the overhead of the system call and the microbenchmark measured in Sections 5.3 and 5.4. Although the overhead percentage of the system call and the microbenchmark is relatively large, that of the total processing time

Table 4 Overhead and average response time of Web server (ms).

File size	Bare	Traced (Relative performance)				Overhead	
		Operation of unmanaged target		Operation of managed target		Operation of unmanaged target	Operation of managed target
1 KB	1.732	1.862	(107.506%)	1.868	(107.852%)	0.130	0.136
10 KB	1.783	1.839	(103.141%)	1.853	(103.926%)	0.056	0.0700
100 KB	5.916	6.183	(104.513%)	6.371	(107.691%)	0.267	0.455
1,000 KB	48.479	50.965	(105.128%)	51.991	(107.244%)	2.486	3.512

© 2016 Information Processing Society of Japan

of the application is small.

In conclusion, the overhead generated by the installation of the VMM-based tracing function is relatively large from the viewpoint of the system call or the microbenchmark level, however, it is small from the viewpoint of the application level.

6. Discussion

6.1 Usage Scenario

We assume that the user has the privilege of both a VMM and a guest OS. We also assume that the user mainly operates in the guest OS. The proposed system mainly aims at preventing information leakage resulting from inadvertence or mismanagement in a regular operation. This assumed situation shows that the proposed system is best suited for personal use at home. Uploading files to a remote server or sending e-mails can be considered as the case of information leakage at home. The information leakages are all achieved via IPC, therefore, those are detectable by the proposed system. Further, although it is described that the proposed system is best suited for personal use at home, we assume that the proposed system can also be used in company offices or other situations if the above assumptions are matched.

6.2 Limitations

In our proposed system, the mechanism of obtaining the guest OS information (e.g., inode number, PID, socket information, etc.) is dependent on the data structure of the guest OS and the specification of the system call, and so the users cannot use any Linux they like as a guest OS. For example, the prototype in this paper is implemented for Linux 3.6.10. In this version, the prototype is dependent on the following data structures: `thread_info`, `task_struct`, `files_struct`, `file`, `dentry`, `inode`, `socket_alloc`, `unix_sock`, and `inet_sock`. Therefore, if the above data structures and system call specifications are the same as Linux 3.6.10, the tracing function is available. Even if the above conditions are not satisfied, users can use any OS by adapting the mechanism of obtaining the guest OS information for a guest OS they want to use.

In this paper, we utilize Linux as a guest OS to unify the host OS and guest OS. Also the reason why Linux is used as a guest OS is that it requires deep understanding of guest OS's data structures to implement the proposed method. This paper also considers the feasibility of the proposed method for other OS such as Windows but it is not sufficient. Windows has a large share, and so there are quite a few scenes treating the classified information with Windows. Therefore, it is a future work to apply the proposed system to Windows.

Since the proposed function is designed and implemented for one guest OS, the prototype of the proposed function cannot apply to a lot of guest OSes on one VMM. However, we thought that by distinguishing the guest OS, the proposed function can apply to a lot of guest OSes on one VMM even in the local or cloud environment. Thus, we will examine the feasibility of the proposed function for a lot of guest OSes.

The proposed method can trace the diffusion of auto-generated temporary/backup files on local drives. This is attributed to the classified information is diffused through system calls described in Table 1. Thus, it is not important the file is generated intentionally or automatically. However, the case for diffusion to files on a network drive is not able to trace but it is able to detect information leakage. Tracing diffusion of classified information on network drives is the future work.

The proposed method cannot differentiate between normal client access and malware access when a malware uses process injection techniques. Because the proposed function manages if a process has a potential of information leakage, it cannot differentiate if the process is malicious or not. However, the proposed function manages processes and files which diffuse classified information. Analyzing the information enables the manager of the VMM to detect which process leaks classified information.

7. Related Work

In this section, we compare similar works related to the proposed VMM-based tracing function. Many methods for tracing the diffusion of information within an OS have been proposed. Thus, we introduce approaches that aim to trace the classified information through both a dynamic and a static analysis. We also refer the respective method used for preventing information leakage.

TaintDroid [16] is a method for tracing the diffusion of classified information using a dynamic taint analysis (DTA). A DTA is used to track information that has been tainted by other data. Subsequently, if the tainted data are written to another memory location, the destination is marked as tainted. Thus, we can follow the classified information through a DTA. TaintEraser [17] uses a similar method. Implemented for smartphones, TaintEraser traces the diffusion of sensitive information within such a device. Further, when an external leakage of information is detected, the user receives a notification. Taint-exchange [18] is a method for cross-host taint tracking. This method is applied by injecting tainted information into a data transfer. Argos [19] is a honey pot utilizing a DTA and is implemented on QEMU [20]. Argos marks the data received from a network and tracks the tainted data. Subsequently, when the data received from a network are executed, Argos detects the data as an attack and obtains the information related to the attack (e.g., a memory dump). In Ref. [21] a taint-based protection system implemented on Xen [22] is described. This method is used to track data received from a network by tainting and preventing their execution. Thus, attacks based on a malicious code injection are prevented. To mark the data using a taint-tag, a DTA requires an additional storage, called a shadow memory. Therefore, an additional non-trivial memory and disk space are required. In contrast, the VMM-based tracing function can trace the diffusion of classified information without additional memory or disk space. However, the tracing granularity of the VMM-based tracing function is coarser than that of a DTA because the VMM-based tracing function is based on the probability of information diffusion caused by system calls.

There are also other methods for dynamically tracing the information flow. CopperDroid [23] operates Android malware on QEMU, and analyzes the behavior of this malware by hooking the system calls. An analysis of Android-specific IPC using Binder is achieved by hooking ioctl(), which sends Binder the

© 2016 Information Processing Society of Japan

data. Aquifer [7] prevents an unintended information leakage by limiting the applications that can handle sensitive data using a policy restricting the host exportation. AppIntent [24] detects the transmission of sensitive data using an Android application and notifies the transmission to the user. Subsequently, during an unintentional user operation, the application that executed the operation is judged to be malicious. The purpose of these methods is to analyze malware or prevent information leakage. On the other hand, the purpose of the VMM-based tracing function is to grasp the location of classified information and identify the cause of information leakage. Grasping the location of classified information prevents an information leakage resulting from inadvertence or mismanagement. In addition, identifying the cause of information leakage leads to the prevention of its recurrence.

Collecting information inside a VM from the outside is known as Virtual Machine Introspection (VMI) [25]. VMI is used in many researches to offload security systems to the outside of VMs. XenAccess [26] is a monitoring library for VMI and LibVMI [27] is an extended version of XenAccess. These libraries enable IDS applications on a Host OS to introspect the memory and disks of other VMs. These libraries are useful for collecting kernel data of other VMs from the Host OS. Kourai et al. proposed KVMonitor [28], which offloads legacy intrusion detection systems to KVM, for VM introspection. In addition to XenAccess and LibVMI, KVMonitor enables IDSes on Host OS to introspect network packets of other VMs. Because KVMonitor monitors network packets of VMs, information granularity is different from the VMM-based tracing function. To trace information diffusion by socket communication, it is required to reconstruct semantic views from captured network packets. Because the VMM-based tracing function detects system calls related to socket communication, the cost for semantic view reconstruction is less.

DroidSafe [29] provides a static information flow analysis framework. It analyzes an information flow that has the potential to include sensitive data. From such an analysis, it can be verified whether an Android application has the potential to leak sensitive data. DroidSafe detects the possibility of such a leakage statically. On the other hand, the proposed VMM-based tracing function traces the diffusion of classified information dynamically. Additionally, DroidSafe is specialized for Android because it detects the possibility of leakage from an intent, which is an Android-specific system. In contrast, the VMM-based tracing function has high versatility because it traces the diffusion of classified information by hooking a system call, which is widely used in various environments.

Tightlip [6] is a privacy management system that swaps an original process for a dummy process, called a "Doppelgangers," when a process that includes sensitive data attempts to write the data to a network. This protects the sensitive data from leakage because Doppelgangers do not contain sensitive data themselves. Filesafe [10] protects sensitive files on a guest OS using a 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 VMM, Filesafe can prevent sensitive files from unauthorized access. SVFS [11] operates a Normal VM running standard applications, an Admin VM for the purpose of system administration, and a DVM to store sensitive files for other VMs. These sensitive files can be edited only by the Admin VM. Thus, it is possible to protect these files even if the Normal OS is compromised by an attacker. In addition, VOFS [30] only permits the user to view sensitive files using SVFS. The method in Ref. [8] reduces the root privilege. It prevents the modification of files that exceed the authority, thereby protecting important files. The method of Ref. [31] is a provenance-aware system targeting Linux OS. It records a data provenance using LPM (Linux Provenance Module) and realizes an access control. In addition, an authenticity and integrity are guaranteed by a Linux-IMA (Linux-Integrity Measurement Architecture). TightLip, Aquifer, SVFS, and the method of Refs. [8] and [31] are necessary for modifying the structure of an OS, and hence, the operational environment is limited. In contrast, the proposed VMM-based tracing function can be installed in various environments owing to a lack of necessary modifications to the OS. In addition, Filesafe necessitates the setting of a policy for each file individually, which may cause a leakage of classified information through a policy misconfiguration. In contrast, the VMM-based tracing function automatically traces the diffusion of classified information, and therefore, the risk of information leakage through a policy misconfiguration is low.

Cashtags [32] prevents information leakage that may occur through shoulder surfing in public places. To prevent such information leakage, Cashtags are used to replace sensitive data elements with non-sensitive data elements before they are displayed on the screen. I-BOX [9] prevents information leakage by untrusted input method editor (IME) apps. It also intercepts and analyzes the user's input data. When sensitive data are included in the user's input data, I-BOX rolls back the execution of the IME app state. This prevents an information leakage from an untrusted IME app. Although these studies are targeted toward smartphones, they have the same purpose as the present study from the viewpoint of sensitive data protection.

8. Conclusion

In this paper, we describe a method for tracing the diffusion of classified information through a socket communication using a VMM-based tracing function. During local IPC, the proposed VMM-based tracing function traces the diffusion of classified information through a socket by managing whether the process and socket have classified information. During remote IPC, the VMM-based tracing function detects information leakage by detecting a managed process used to send classified information outside the computer. The VMM-based tracing function also enables identifying the cause of information leakage by recording the destination IP address, port number, command name, and PID.

We implemented and evaluated the prototype of a proposed VMM-based tracing function for a socket communication using a KVM as the VMM and 64-bit Linux as the guest OS. In our evaluation of the traceability, we confirmed that the VMM-based tracing function can trace the diffusion of classified information through local IPC and detect information leakage through remote

© 2016 Information Processing Society of Japan

IPC. During a performance evaluation, the overhead of the system calls related to the diffusion of classified information was within the range of 23.76 to 49.58 μs, with a relative performance level of 283.83% to 499.10%, which are relatively large values. On the other hand, the overhead of getpid(), which is unrelated to the diffusion of classified information, was 0.0011 μs, which is a relative performance level of 106.86%, a small value compared to those obtained for system calls related to the diffusion of classified information. In addition, the performance degradation in a web server is less than about 10%. According to this result, we can say that the overhead generated by the installation of the VMM-based tracing function is small from the viewpoint of the application level.

IPC is implemented through various methods, for example, a pipe, FIFO, message queue, shared memory, or socket. In this paper, we focus on tracing the socket communication because of its high priority. However, it also important to trace the other IPCs. Similar to the socket communication, these are conducted through a communication medium. Therefore, these are also traceable by managing whether their communication medium has classified information based on the proposed method. In our future work, we will implement the proposed VMM-based tracing function for IPC, excluding a socket communication. In addition, we will adapt the VMM-based tracing function to Windows and cloud environment.

References

[1] Japan Network Security Association: 2008 Information Security Incident, available from ⟨http://www.jnsa.org/result/incident/data/2008incident_survey_e_v1.0.pdf⟩ (accessed 2015-12-01).

[2] The Gurdian: Antivirus software is dead, says security expert at Symantec, available from ⟨http://www.theguardian.com/technology/2014/may/06/antivirus-software-fails-catch-attacks-security-expert-symantec⟩ (accessed 2015-12-01).

[3] Tabata, T., Hakomori, S., Ohashi, K., Uemura, S., Yokoyama, K. and Taniguchi, H.: Tracing Classified Information Diffusion for Protecting Information Leakage, *IPSJ Journal*, Vol.50, No.9, pp.2088–2102 (2009) (in Japanese).

[4] Nomura, Y., Hakomori, S., Yokoyama, K. and Taniguchi, H.: Tracing the Diffusion of Classified Information Triggered by File Open System Call, *Proc. 4th Int. Conf. Computing, Communications and Control Technologies* (*CCCT 2006*), pp.312–317 (2006).

[5] Otsubo, N., Uemura, S., Yamauchi, T. and Taniguchi, H.: Design and Evaluation of a Diffusion Tracing Function for Classified Information Among Multiple Computers, *7th FTRA International Conference on Multimedia and Ubiquitous Engineering* (*MUE 2013*), Lecture Notes in Electrical Engineering (LNEE), Vol.240, pp.235–242 (2013).

[6] Yumerefendi, A.R., Mickle, B. and Cox, L.P.: TightLip: Keeping Applications from Spilling the Beans, *Proc. 4th USENIX Conference on Networked Systems Design and Implementation* (*NSDI 2007*), pp.159–172 (2007).

[7] Nadkarni, A. and Enck, W.: Preventing accidental data disclosure in modern operating systems, *Proc. 2013 ACM SIGSAC Conference on Computer and Communications Security* (*CCS '13*), pp.1029–1042 (2013).

[8] Glenn, W. and Oorschot, P.C.: A Control Point for Reducing Root Abuse of File-system Privileges, *Proc. 17th ACM Conference on Computer and Communications Security* (*CCS '10*), pp.224–236 (2010).

[9] Chen, J., Chen, H., Bauman, E., Lin, Z., Zang, B. and Guan, H.: You Shouldn't Collect My Secrets: Thwarting Sensitive Keystroke Leakage in Mobile IME Apps, *24th USENIX Security Symposium* (*USENIX Security 15*), pp.657–690 (2015).

[10] Wang, J., Yu, M., Li, B., Qi, Z. and Guan, H.: Hypervisor-based Protection of Sensitive Files in a Compromised System, *Proc. 27th Annual ACM Symposium on Applied Computing* (*SAC 2012*), pp.1765–1770 (2012).

[11] Zhao, X., Borders, K. and Prakash, A.: Towards protecting sensitive files in a compromised system, *Proc. 3rd IEEE International Security in Storage Workshop* (*SISW '05*), pp.21–28 (2005).

[12] Fujii, S., Sato, M., Yamauchi, T. and Taniguchi, H.: Evaluation and Design of Function for Tracing Diffusion of Classified Information for File Operations with KVM, *The Journal of Supercomputing* (*SUPE*), Vol.72, No.5, pp.1841–1861 (2016).

[13] KVM: Main_Page, available from ⟨http://www.linux-kvm.org/page/Main_Page⟩ (accessed 2015-12-01).

[14] Xuxian, J., Xinyuan, W. and Dongyan, X.: Stealthy malware detection and monitoring through VMM-based "out-of-the-box" semantic view reconstruction, *ACM Trans. Inf. Syst. Secur.*, Vol.13, No.2, pp.1–28 (2010).

[15] McVoy, L. and Staelin, C.: lmbench: Portable tools for performance analysis, *Proc. USENIX 1996 Annual Technical Conference*, pp.279–294 (1996).

[16] Enck, W., Gilbert, P., Chun, B.-G., Cox, L.P., Jung, J., McDaniel, P. and Sheth, A.N.: TaintDroid: An Information-flow Tracking System for Realtime Privacy Monitoring on Smartphones, *Proc. 9th USENIX Conference on Operating Systems Design and Implementation* (*OSDI '10*), pp.1–6 (2010).

[17] Zhu, D.Y., Jung, J., Song, D., Kohno, T. and Wetherall, D.: Taint-Eraser: Protecting Sensitive Data Leaks Using Application-level Taint Tracking, *ACM SIGOPS Oper. Syst. Rev.*, Vol.45, No.1, pp.142–154 (2011).

[18] Zavou, A., Portokalidis, G. and Keromytis, A.D.: Taint-exchange: A Generic System for Cross-process and Cross-host Taint Tracking, *Proc. 6th International Conference on Advances in Information and Computer Security* (*IWSEC '11*), pp.113–128 (2011).

[19] Portokalidis, G., Slowinska, A. and Bos, H.: Argos: An Emulator for Fingerprinting Zero-day Attacks for Advertised Honeypots with Automatic Signature Generation, *Proc. 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006* (*EuroSys '06*), pp.15–27 (2006).

[20] Bellard, F.: QEMU, A Fast and Portable Dynamic Translator, *Proc. USENIX 2005 Annual Technical Conference*, *FREENIX Track*, pp.41–46 (2005).

[21] Ho, A., Fetterman, M., Clark, C., Warfield, A. and Hand, S.: Practical Taint-based Protection Using Demand Emulation, *Proc. 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006* (*EuroSys '06*), pp.29–41 (2006).

[22] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I. and Warfield, A.: Xen and the Art of Virtualization, *Proc. 19th ACM Symposium on Operating Systems Principles* (*SOSP '03*), pp.164–177 (2003).

[23] Tam, K., Khan, S.J., Fattori, A. and Cavallaro, L.: CopperDroid: Automatic Reconstruction of Android Malware Behaviors, *22nd Annual Network and Distributed System Security Symposium* (*NDSS 2015*) (2015).

[24] Yang, Z., Yang, M., Zhang, Y., Gu, G., Ning, P. and Wang, X.S.: AppIntent: Analyzing sensitive data transmission in Android for privacy leakage detection, *Proc. 2013 ACM SIGSAC Conference on Computer and Communications Security* (*CSS '13*), pp.1043–1054 (2013).

[25] Garfinkel, T. and Rosenblum, M.: A Virtual Machine Introspection Based Architecture for Intrusion Detection, *The 10th Annual Network and Distributed System Security Symposium* (*NDSS 2003*), pp.191–206 (2003).

[26] Payne, B.D., Carbone, M.D.P.D.A. and Lee, W.: Secure and Flexible Monitoring of Virtual Machines, *23rd Annual Computer Security Applications Conference* (*ACSAC*), pp.385–497 (2007).

[27] LibVMI Project: LibVMI, available from ⟨http://libvmi.com⟩ (accessed 2015-12-01).

[28] Kourai, K. and Nakamura, K.: Efficient VM Introspection in KVM and Performance Comparison with Xen, *2014 IEEE 20th Pacific Rim International Symposium on Dependable Computing* (*PRDC*), pp.192–202 (2014).

[29] Gordon, M.I., Kim, D., Perkins, J., Gilham, L., Nguyen, N. and Rinard, M.: Information-Flow Analysis of Android Applications in DroidSafe, *22nd Annual Network and Distributed System Security Symposium* (*NDSS 2015*) (2015).

[30] Borders, K., Zhao, X. and Prakash, A.: Securing Sensitive Content in a View-only File System, *Proc. ACM Workshop on Digital Rights Management*, pp.27–36 (2006).

[31] Bates, A., Tian, D. Butler, K.R.B. and Moyer, T.: Trustworthy Whole-System Provenance for the Linux Kernel, *24th USENIX Security Symposium* (*USENIX Security 15*), pp.319–334 (2015).

[32] Mitchell, M., Wang, A.A. and Reiher, P.: Cashtags: Protecting the Input and Display of Sensitive Data, *24th USENIX Security Symposium* (*USENIX Security 15*), pp.961–976 (2015).

Editor's Recommendation

This paper proposed the tracing mechanism against IPC's information exchange. The authors also implemented the prototype

© 2016 Information Processing Society of Japan

of proposed mechanism on a KVM (Kernel-based Virtual Machine). The proposed function is worthwhile to make grab all of the diffusion path on the KVM so that the guest OS modification are not necessary. An IPC tracing on KVM is more complex and more difficult than tracing file operation or child process creation because KVM must grasp relations between sockets and processes. The paper describes a valuable discussion of how to solve these difficulties. The paper gives insights to readers in this research field and thus is selected as a recommended paper.

(Chief examiner of SIGCSEC Satoru Torii)

Shota Fujii received his B.E. and M.E. degrees from Okayama University, Japan in 2014 and 2016 respectively. His research interests include computer security and virtualization technology. He is a member of IPSJ.

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 has been an Assistant Professor of Graduate School of Natural Science and Technology at Okayama University since 2014. His research interests include computer security and virtualization technology. He is a member of IPSJ and IEICE.

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

Hideo Taniguchi received a B.E. degree in 1978, an 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 Headequarters, NTT DATA Communications Systems Corporation. He has been an Associate Professor of Computer Science at Kyusyu University since 1993 and a Professor of the Faculty of Engineering at Okayama University since 2003. His research interests include operating systems, real-time processing and distributed processing. He is the author of Operating Systems (Shoko Publishing Co. Ltd.), etc. He is a fellow of IPSJ. He is a member of IEICE and ACM.

© 2016 Information Processing Society of Japan