International Journal of Networking and Computing – www.ijnc.org, ISSN 2185-2847
Volume 9, Number 2, pages 217–238, July 2019

Implementation and Evaluation of Communication-Hiding Method by System Call Proxy

Yuuki Okuda, Masaya Sato, Hideo Taniguchi
Graduate School of Natural Science and Technology, Okayama University,
3-1-1 Tsushima-naka, Kita-ku Okayama-city, Okayama, 700-8530, Japan
Email: okuda2@swlab.cs.okayama-u.ac.jp, {sato, tani}@cs.okayama-u.ac.jp

Received: February 15, 2019
Revised: May 4, 2019
Accepted: June 11, 2019
Communicated by Toru Nakanishi

Abstract

Essential services, such as security software or logging software, are considered important because of an increase in attacks on computers. These essential services are provided by processes that sometimes involve file manipulation and communication. Moreover, these essential services can be a target of attacks and become disabled, as they can be an obstacle to attackers. Attackers can speculate essential services by monitoring the behavior of the processes. To avoid such attacks on essential services, methods for hiding their behavior are proposed. The methods use a virtual machine (VM) monitor to make it difficult for attackers to identify essential services by hiding process information and file manipulation.

However, the communication information remains visible to attackers. To address this problem, this study proposes a method for hiding the communication of essential services by using a system call proxy. We assume that a process providing essential services (essential process) runs on a protection target VM and a proxy process runs on a proxy VM. In the proposed method, the system calls in the communication invoked by the essential process are executed by the proxy process. The system calls invoked by the proxy process are not executed on the protection target VM; therefore, attackers cannot identify the communication of essential services by monitoring their communication. This paper presents the design, implementation, and evaluation of the proposed method.

Keywords: Attack Avoidance, Virtual Machine, Communication Hiding

1 Introduction

For preventing and mitigating damages caused by attackers, several services are provided on computers. Security software prevents attacks from malicious software (malware). Logging or management tools are also used to detect and analyze attacks and damages caused by malware. In this study, we define security software and system management tools as essential services. Protecting these essential services is an important challenge for preventing and mitigating damages by attackers. Following are some of the proposed methods for protecting essential software: ANSS prevents termination of security software by monitoring and controlling system calls that attempt to terminate the security software [4]. VMI analyzes malware on a VM by monitoring from a VM monitor (VMM) [3]. Process out-grafting also analyzes malware by keeping the monitoring environment secure [9]. When an attacker wants to terminate an essential service, he identifies it and prepares to attack it beforehand.

However, these methods do not focus on the complicated process of the identification of essential services. If the identification of essential services is made more difficult, then attacking these services becomes difficult.

Process information and process behavior are useful for identifying an essential service. The process information such as PID or name of execution command is given to identify a process. The process behavior involves file information and communication information. The file information such as configuration files or whitelists for security software can be a clue to detect an essential service. The file information includes file entity and file manipulation; therefore, not only file entities such as configuration files and whitelists for security software, but also access to these files can be a clue to identify essential services. The communication information such as communication contents or a hostname of destination can also be a clue to detect an essential service.

We already proposed methods for complicating the identification of essential services by hiding process information or file information. In the following, a process providing an essential service is defined as an essential process. In the process information-hiding method [8], the VMM detects the process switching in the guest OS and replaces the process information of the essential process with dummy information while other processes run. In Linux, we applied the proposed method to make the task_struct, which contains the process information, invisible to other processes. As a result, it becomes difficult for other processes to identify the essential process based on the task_struct. Additionally, we proposed a file manipulation-hiding method [7] to hide files related to essential services and manipulation to the files. The method provides the files invisible to processes other than essential processes by placing the files on another VM. The method also provides a method for essential processes to access those files in a VM environment. In this environment, this method makes it difficult to identify essential services based on file entity or file manipulation of essential services.

However, communication information of essential process is still visible to attackers. Although the process information and the file information are made invisible by previous methods, in fact, an essential process runs. Even if the process information and the file information are hidden, the communication information such as communication contents or a hostname of destination can be a clue to detect an essential service. Attackers can acquire communication information by using monitoring methods such as packet capture. Moreover, by setting breakpoints or tampering an entry of the system call table, they can acquire system call arguments which include communication information from the memory of the essential processes. Therefore, it is necessary to prevent attackers from identifying essential services based on communication information of essential services.

In this study, we propose a hiding method for essential services' communication for complicating the identification of essential services. In the proposed method, system calls in the communication invoked by essential services are executed by a proxy process. The proxy process must be isolated from an operating system (OS) hosting the essential service for hiding the proxy execution of system calls. We utilize a VMM for isolating the proxy process. The VMM interposes system calls in the communication invoked by the essential services and transfers their execution to the proxy process running on another VM. The essential service resumes its processing as if the system calls are executed in the local kernel. Because communication is interposed by the VMM and executed by the proxy process on another VM, attackers cannot identify the communication of essential services by monitoring their communication. The proposed method is designed to use Linux 3.2.0 as the guest OS and Xen 4.2.3 as the VMM. Additionally, we evaluated the performance of the proposed method.

The contributions made in this paper are as follows:

1. We propose a communication-hiding method for complicating the identification of essential services by using a system call proxy. In this method, a VMM interposes system calls in the communication invoked by the essential process running on a protection target VM. Then, the system calls are executed by a proxy process on the proxy VM and their results are returned to the essential process. Therefore, it is possible to hide the communication of essential processes from attackers in the protection target VM because the system calls are not executed on the VM. Additionally, the method is implemented by a VMM and a proxy process on the proxy

VM; therefore, no modification to the essential services and the OSs in the protection target VM is required.

2. In the evaluation, we showed that the communication of essential processes can be made invisible by the proposed method, even if when using monitoring tools such as strace or tcpdump. In addition, we evaluated the behavior of the proxy processes when multiple protection target VMs run on the VMM. The evaluation results showed that even the execution of proxy system calls requires a sufficient interval (larger than 0) to check a proxy execution request. Furthermore, we showed that the performance overhead of the sendto() system call is approximately 2.3 μs at the least and increases in proportion to the size of the sending data. Finally, we showed that the performance of the sendto() system call when applying the proposed method is better than that when executing the system call on the VM as-is.

2 Attacks on Essential Services

We assume security software and system management tools as essential services. We define a process providing an essential service as an essential process. Essential services can be a target of attacks and be disabled because they can be an obstacle to attackers. It is reported that security software has certain drawbacks which can be exploited by malware [5]. The adore-ng, which is a kind of rootkit, invalidates syslog, which is a logging software, and makes it difficult to detect the intrusion into the system by malware [10]. If the security software is disabled, it cannot prevent attacks from malware, and therefore considerable damage to the system may occur. If system management tools such as logging software are disabled, then they cannot collect logs such as malware installation, and therefore the damage to the system worsens.

Before making attacks, attackers first identify an essential service to be targeted. To identify an essential service, process, file, and communication information are useful. The process information and file information has already been invisible from attackers by past methods. The process information-hiding method [8] replaces the process information of the essential service with dummy information while other processes run. The file information-hiding method [7] provides the files invisible to processes other than essential services. In addition to this, a method to access those files in a VM environment is proposed. Therefore, other processes cannot identify a process as an essential service by monitoring process information or file information. However, communication information is still visible to attackers. Therefore, it is necessary to prevent attackers from identifying essential services based on communication information of essential services.

3 Communication Hiding

3.1 Purpose

The purpose of our research is to avoid the identification of essential services based on their communication. As described in Section 2, it is possible to monitor the communication using monitoring methods and to identify essential services from communication information. To address the problem of the identification, we propose a communication-hiding method for avoiding the identification of essential services.

3.2 Requirements

Requirements for addressing the abovementioned problem are as follows:

Requirement 1: To make communication of essential service invisible to attackers

Requirement 2: To make the method satisfying Requirement 1 invisible to normal processes or kernel modules.

It is required to hide the communication of essential services from attackers for avoiding the identification of the essential services (Requirement 1). Additionally, a method satisfying Requirement 1 should not be detected by attackers (Requirement 2). If such a method is detected, attackers can speculate the essential services by monitoring the detected method. We assume that attackers have root privilege and insert their malicious programs into normal processes or kernel modules. The normal processes are processes excluding essential services. Therefore, it is required to hide the method from normal processes and kernel modules.

3.3 Challenges

In order to fulfill the abovementioned requirements, the followings should be addressed:

Challenge 1: Interposition of communication by essential services

Challenge 2: Control of interposed communication

Challenge 3: Method of dealing with Challenges 1 and 2 is not detected by normal processes or kernel modules.

Challenges 1 and 2 are required to satisfy Requirement 1. In order to make communication of essential services invisible to attackers, the interposition of communication of essential services and controlling the interposed communication are required. Additionally, for satisfying Requirement 2, the method of dealing with Challenges 1 and 2 should be invisible to attackers.

3.4 Hiding Method for Communication by System Call Proxy

To address the abovementioned challenges, we propose a method that proxies the communication of essential processes by using a system call proxy. The method addresses the challenges as follows:

3.4.1 Dealing with Challenge 1

We trust a VMM and use the VMM for our proposed method. A VMM is useful for monitoring the operation of VMs; it is used in malware analysis [2] and file protection [12]. In this method, the VMM interposes system calls [6] for monitoring the communication of essential services. A VMM and VMs are isolated from each other, therefore interposing communication between a VM and a VMM in a non-intrusive manner helps the method to hide itself from attackers on the VM.

3.4.2 Dealing with Challenge 2

As stated in the previous section, the VMM interposes system calls invoked by the essential processes. Further, the interposed system call needs to be executed in another environment. Owing to this reason, this method requires a proxy process on another VM for executing a proxy system call. Additionally, the result of the proxy execution is returned to the essential process via the VMM. Therefore, a system call invoked by an essential process is executed on another VM, and its result is returned to the essential process.

3.4.3 Dealing with Challenge 3

To handle Challenge 3, the proposed method uses a VMM and VMs, which are isolated from each other. In this method, essential processes run on an OS on a VM, and system calls invoked by the essential processes are interposed by the VMM and executed on another VM. It is difficult to detect this method even if an attacker gets root privilege on the VM because it is implemented by a VMM and a proxy process on another VM.

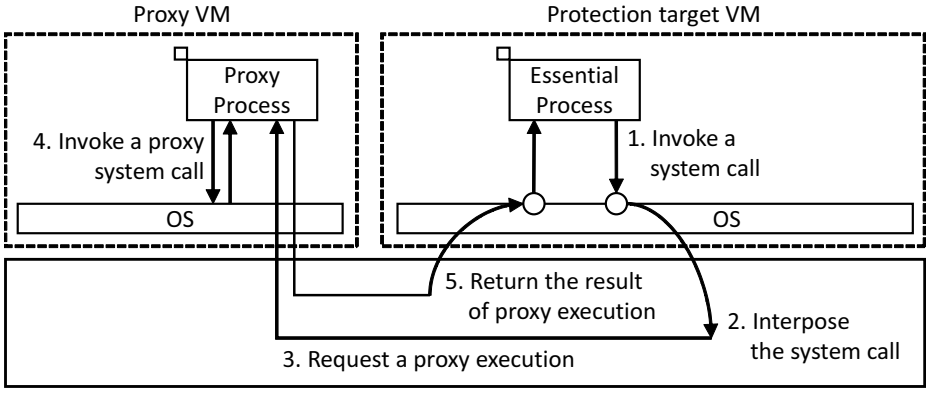

Figure 1: Design of the proposed method.

4 Implementation

4.1 Environment

We use Xen [1] as a VMM. VMs running on top of a VMM are fully virtualized with Intel VT-x. We assume Linux to be the guest OS and system calls to be invoked by a syscall instruction. Figure 1 shows the design of the proposed method. In this case, two VMs run on the VMM: a protection target VM and a proxy VM. An essential process that provides essential services runs on the protection target VM. A proxy process that executes a proxy system call runs on the proxy VM.

In such an environment, the VMM interposes a system call invoked by the essential process and requests the proxy process to execute a proxy system call. After confirming the request, the proxy process invokes a system call as a proxy execution. Following the execution, its result is returned to the VMM. Finally, the VMM returns the result to the essential process. The details of this method are described in the latter part of this section.

4.2 Overall Flow of the System Call Proxy

Figure 2 shows the overall flow of the proposed method. The method is implemented by the VMM and the proxy process. The VMM performs the following functions:

1. Detect the invocation of a system call on the protection target VM.
2. Acquire the information for a proxy execution.
3. Request the proxy process for a proxy execution.
4. Check the completion of the proxy execution.
5. Set the result of the proxy execution in registers.
6. Return the processing to the protection target VM.

The proxy process also performs the following functions:

1. Confirm the request for a proxy execution.
2. Invoke a system call as a proxy execution.
3. Return the result of the proxy execution.

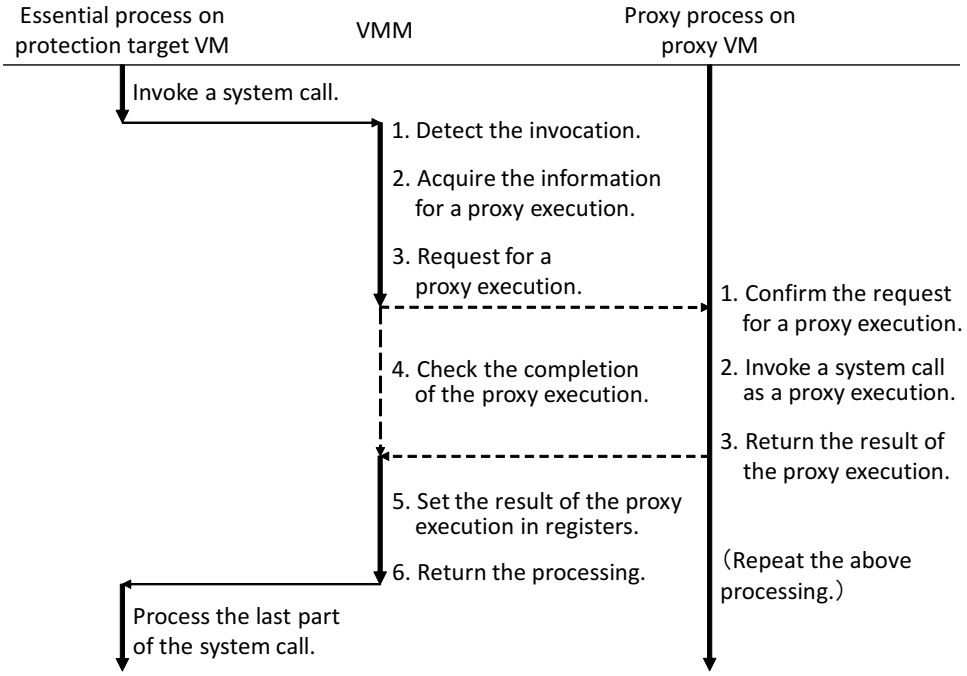

Figure 2: Overall flow of the proposed method.

When a system call is invoked on the protection target VM, the VMM interposes it and determines whether the system call is invoked by an essential process and is related to communication. In this case, the VMM acquires the information for a proxy execution such as the arguments of the system call (described in Section 4.6). Further, the VMM requests the proxy process to execute a proxy system call. Then, the VMM checks the completion of the proxy execution with the polling method.

The proxy process checks the request by invoking hypercalls. A hypercall is an interface for requesting processing to a VMM. We added new hypercalls to the Xen hypervisor. During a request, the new hypercall fetches information for the proxy execution from the VMM. Further, the proxy process invokes a proxy system call based on the fetched information. After the proxy execution, the proxy process returns the result of the proxy execution to the VMM by invoking another new hypercall. This hypercall returns the result of the proxy execution and notifies the VMM of the completion of proxy execution.

Notified by the proxy process, the VMM manipulates the register of the protection target VM for returning the result of the proxy system call. Additionally, the VMM sets the instruction pointer to the last part of the system call processing for skipping the execution of the original system call. Finally, the VMM returns the processing to the protection target VM.

According to the above method, system calls in the communication invoked by an essential process are executed by a proxy process on another VM. The details of this flow are described in the following parts of this section.

4.3 System Call Interposition

We used a debug exception to detect invocation of system calls in the protection target VM by the VMM. In the proposed method, the VMM manipulates debug registers for setting a hardware breakpoint at the starting address of the system call routine. Additionally, the VMM manipulates the VM-execution control field for a protection target VM to cause a VM exit when a debug exception occurs. In this case, a debug exception occurs before the execution of the instruction at the address;

the VM exit occurs and the processing transitions to the VMM as a result.

The VMM detects an invocation of the system call when the reason for the VM exit is a debug exception, and the address where the debug exception occurs is the starting address of a system call routine. Thereby, the VMM interposes system calls invoked in the protection target VM.

We assume that the proposed method is used for the operational environment that does not use debug registers. For this reason, it is no problem using the debug registers for implementing the proposed method. Additionally, accesses to the debug registers can be interposed by the VMM. Therefore, manipulating access to the registers and disguising the contents of the registers by the VMM, it is possible to prevent the tampering of the debug registers and the detection of the proposed method from attackers.

4.4 Flow of the VMM

In order to make the system calls invisible on the protection target VM, it is necessary to execute the system call on another VM and return the result to the essential process. Figure 3 shows the flow of the VMM. The details of the flow are as follows:

1. The VMM detects a system call invoked in the protection target VM by the abovementioned method.
2. The VMM determines whether the system call is invoked by the essential process. If the process invoking the system call is an essential process, the VMM goes to step 3, else to step 15.
3. The VMM determines whether the system call is related to communication. If the system call is related to communication, the VMM goes to step 4, else to step 15.
4. The VMM gets the arguments of the system call. If the argument is the address of the buffer, the VMM copies the buffer to the VMM region.
5. The processing branches on system call type. If the system call is read(), write(), or close(), the VMM goes to step 6, else to step 7.
6. If the file descriptor, which is the first argument of read(), write(), or close(), is registered to hiding list, the VMM goes to step 7, else to step 15. The hiding list is a list which reserves file descriptors used for communication.
7. The VMM requests the proxy process to execute a proxy system call.
8. Post the request, the VMM checks the state of proxy execution with the polling method until the execution is completed.
9. If the result of proxy execution is successful, the VMM goes to step 10, else to step 13.
10. The processing branches on system call type. If the system call is socket(), close(), or others, the VMM goes to step 11, 12, or 13, respectively.
11. The VMM registers a file descriptor, which is a return value of the socket() system call, to the hiding list.
12. The VMM deletes a file descriptor, which is the first argument of the close() system call, from the hiding list. Attributed to 11 and 12, the VMM can determine whether the read(), write(), or close() system call is used for communication or not by referring to the list.
13. Set the result of proxy execution in the registers. If the argument is the address of the buffer, the VMM copies the buffer to the memory assigned to the protection target VM.
14. The VMM set the instruction pointer of the protection target VM to the last part of the system call processing for skipping the original system call.

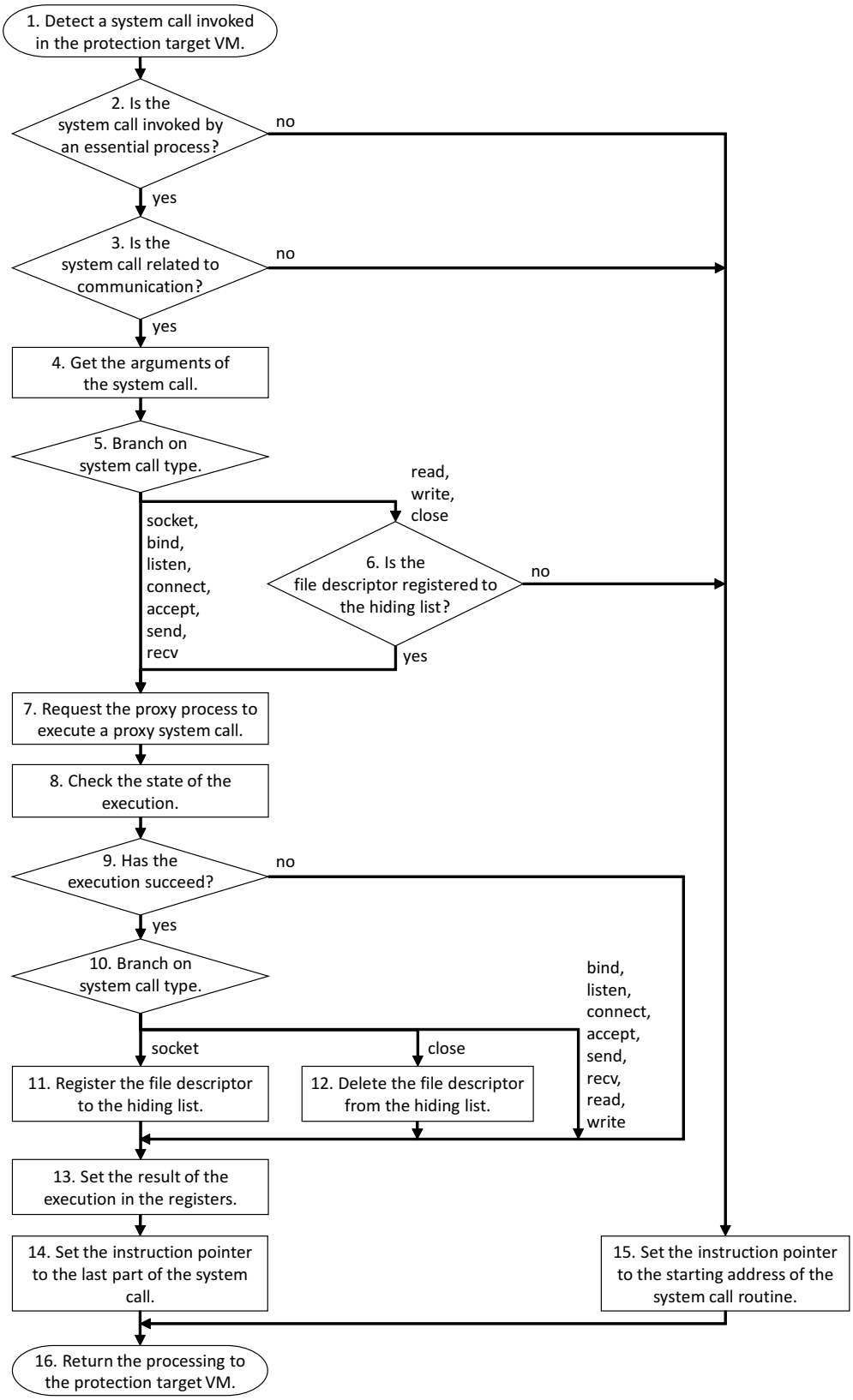

Figure 3: Flow of the VMM.

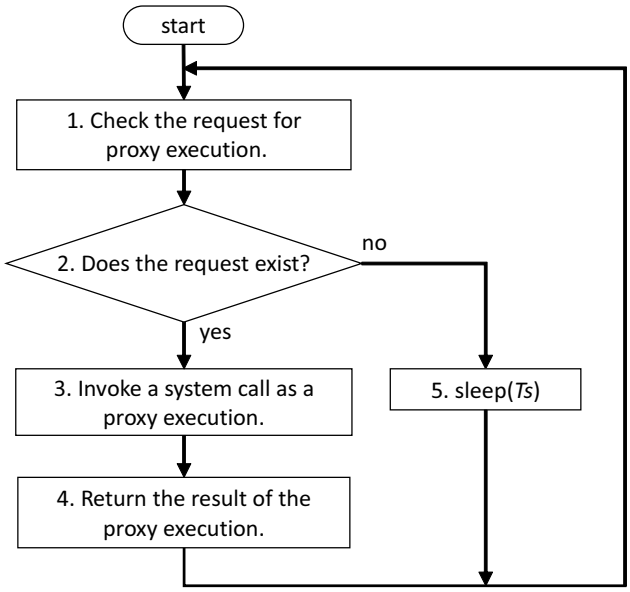

Figure 4: Flow of the proxy process.

15. The VMM set the instruction pointer of the protection target VM to the starting address of the system call routine for executing the system call in the protection target VM.

16. The VMM returns the processing to the protection target VM. Thereby, the essential process can be restarted from the last part of the system call processing.

In this way, the VMM bypasses the system calls related to the communication invoked by an essential process. As a result, system calls invoked by the essential process are not executed on the protection target VM, so that it is possible to avoid monitoring the communication of the essential process on the protection target VM.

4.5 Flow of the Proxy Process

Figure 4 shows the flow of the proxy process. The details of the flow are described as follows:

1. The proxy process checks the existence of a request for proxy execution by invoking a hypercall which is introduced by us. The hypercall gets the information for proxy execution from the VMM if the request exists. In this case, the hypercall returns 1. If the request does not exist, the hypercall returns 0.

2. Branch on the return value of the hypercall. If the request exists, the proxy process goes to step 3, else to step 5.

3. Using the information for proxy execution, the proxy process invokes a system call as a proxy execution.

4. After the execution, the proxy process returns the execution of the system call to the VMM. To return the result, the proxy process invokes another hypercall, which is introduced by us. The hypercall returns the result of proxy execution to the VMM and notifies the VMM of completion of proxy execution.

5. If the request does not exist, then the proxy process sleeps for T_s. Thereby, the proxy process can check the existence of requests with a constant period T_s. By setting this T_s to be larger

Table 1: Information required for the proxy execution.

	Name	Description
1.	VMID	VMID is required to identify the protection target VM.
2.	Address of the page directory	The starting address of the page directory is required to identify the essential process.
3.	System call number	The system call number is used to determine whether the system call is related to communication or not.
4.	System call arguments	The system call arguments are required for requesting the proxy execution of the system call. If the argument is the address of the buffer, the VMM copies the buffer to the VMM region.

than 0, it is possible to lower the CPU usage rate on the proxy VM. Therefore, when a plurality of proxy processes operates, they can be executed evenly.

In the proposed method, the proxy process confirms the request for proxy execution by periodically invoking hypercalls. The event channel, which is a mechanism realizing asynchronous notification in Xen, is not used in the proposed method because of the performance issue. The event channel uses a method of invoking several hypercalls to confirm the existence of an event. In addition to the hypercalls, when it is applied to the proposed method, the proxy process needs to invoke another hypercall to get information for proxy execution. Therefore, the overhead for acquiring the request of proxy execution increases. To reduce the overhead, we added the new hypercall which performs confirmation of the request and acquisition of the information for proxy execution at once.

If the proxy VM is a domain 0, which is a privileged VM in Xen, the system call overhead can be reduced by mapping pages of the protection target VM directly. For example, when executing a system call that requires data transfer such as sendto() system call, the proxy process can directly access the sending data by mapping the page of the protected VM directly. As a result, because the data transfer time can be reduced, system call overhead can be reduced. However, direct mapping of pages increases the risk of buffer corruption and makes the system security lower. For this reason, we considered that it is better to copy the buffer without using direct mapping.

4.6 Information for Proxy Execution

In order to execute a proxy system call in a proxy process, information which is shown in Table 1 is required. The details of the information are described as follows:

1. We assume that multiple VMs run on the VMM. In such an environment, it is necessary to determine whether the VM invoking the system call is the protection target VM or not. In order to identify a protection target VM, the VMM gets a VMID, which is uniquely allocated to each VM. A VMID can be acquired from the VM control structures.

2. It is necessary to identify the essential process because multiple processes run on most OSes. In order to identify an essential process, the VMM uses the address of the page directory. The starting address of page directory is unique to each process and can be acquired from the CR3 register. The VMM can classify essential processes without copying the PID because the registers are saved to the VMM when a VM exit occurs. We can reduce the overhead for copying PID from the memory of the protection target VM by using the page directory for classifying essential processes.

3. It is necessary to identify whether the interposed system call is related to communication or not. In order to identify a system call, the VMM uses a system call number. The system call number can be acquired from RAX register.

Table 2: System calls related to communication.

	System call	Description
1.	socket()	Create an endpoint for communication.
2.	bind()	Bind a name to a socket.
3.	listen()	Listen for connection on a socket.
4.	connect()	Initiate a connection on a socket.
5.	accept()	Accept a connection on a socket.
6.	sendto()	Send a message on a socket.
7.	recvfrom()	Receive a message from a socket.
8.	read()	Read from a file descriptor.
9.	write()	Write from a file descriptor.
10.	close()	Close a file descriptor.

4. In order to execute a proxy system call, system call arguments are required. There are up to six arguments, which can be acquired from RDI, RSI, RDX, R10, R8, and R9 registers in x86_64 architecture. If the argument is the address of a buffer, the VMM copies the buffer to the VMM region.

Which service to be monitored is specified by an user at first. The user can specify a target service to the VMM by starting a proxy process with a VMID and a program name as arguments. The VMM monitors system calls invoked on a VM with the specified VMID. When detecting a system call on the VM, the VMM gets task_struct of the current process from the memory of the protection target VM. Then, the VMM lookups the program name of the process starting from the task_struct. After that, the VMM determines whether the program name is the same as that specified by the user. If that, the VMM acquires the page directory of the process and uses it to identify essential processes from then on. Based on the VMID and the page directory, VMM and proxy processes can identify proxy execution.

4.7 System Calls Related to Communication

The target system calls of the proposed method are shown in Table 2. In order to implement the proposed method, we focus on basic system calls used for socket-based communication. It is possible to extend the proposed method to adapt to other system calls.

If sending or receiving of messages is monitored by attackers, then the essential services may be identified by the messages. In order to avoid identifying essential services, system calls for sending or receiving messages such as sendto(), recvfrom(), read(), and write() are targeted for proxy execution. Additionally, for executing these system calls, it is necessary to establish a connection with the communication partner beforehand. Owing to this, system calls such as socket(), bind(), listen(), connect(), accept(), and close() are also targeted for proxy execution. Furthermore, system calls such as read(), write(), and close() are targeted only when they are used for communication, but not when used for file manipulation.

System calls in Table 2 invoked by essential processes are executed by the proxy processes on the proxy VM. Therefore, the file descriptor and the source IP address of the proxy VM, not the protection target VM, are used for proxy executions. The proxy execution in this method is not complete emulation. This may be a problem because some services can interfere with their operation. For example, when an essential service operates with an awareness of the IP address of the machine on which it runs, the essential service may not be operated properly. Although it may be not necessary to resolve the problem depending on essential services, if it was resolved, we can apply the proposed method to more essential services. To resolve this problem, it is considered to realize a mechanism to convert the IP address. However, it is necessary to investigate the implementation around network driver and consider the mechanism. For this reason, the realization of complete emulation is future work.

4.8 Limitation

4.8.1 Limitation on Implementation

The implementation of the proposed method has the following limitations.

1. Communication of essential processes needs to be based on system calls and detectable by the VMM. This is because system calls invoked by an essential process are interposed by the VMM in the proposed method. For example, it does not support the Data Plane Development Kit (DPDK), which uses a communication method that directly manipulates hardware without going through system calls.

2. The guest OS in the protection target VM needs to be open source to decide the address at which to interpose and return the system call processing. This is because it is unknown whether the guest OS will operate without problems when interposing system calls by the VMM.

3. The system call interface of the proxy VM needs to be compatible with that of the protection target VMs for executing proxy system calls in the proxy VM. If it is a compatible interface, the system calls can be executed as a proxy by converting them via the VMM.

4.8.2 Avoidable Attacks

The following methods are conceivable as a method of monitoring the communication of essential processes in order to identify the existence of the essential processes:

1. Library call hook

2. System call hook

3. Packet capture

When the proposed method is applied, the instructions of target system calls in the protection target VM between the start address of the system call starting routine and the start address of the system call returning routine are not executed. Therefore, the proposed method can prevent attacks on this section. For example, it is possible to avoid an attack of monitoring communication using monitoring methods such as a packet capture or a system call hook tool.

However, as the execution before the system call starting routine and after the system call returning routine is monitored, it is not possible to avoid an attack of tracing the execution of essential processes by using a library call hook tool. To make the essential services less visible, it is necessary to deal with such circumstances. The visibility test is described in Section 5.2.

5 Evaluation

5.1 Purpose, Environment, and Method

In order to evaluate the proposed method, we analyzed the following five items:

1. Visibility test of target system calls.

2. The behavior of the proxy processes regarding intervals for acquiring requests and the overhead of system calls.

3. The behavior of the proxy processes regarding the time of the proxy execution.

4. The system call overhead regarding the size of the data transferred between VMs.

5. The performance when system calls are actually executed.

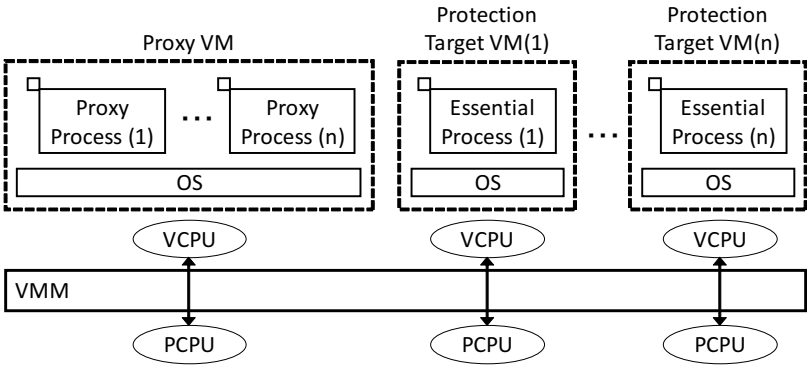

Figure 5: Model of the evaluation.

Table 3: Environment used for the evaluation.

Machine 1		Machine 2	
CPU	Intel Core i7-2600 (3.4 GHz, 4 cores)	CPU	Intel Core i7-4790 (3.6 GHz, 4 cores)
Memory	8 GB	Memory	8 GB
VMM	Xen 4.2.3	OS	Debian 9.3 (Linux 4.9.30 64-bit)
· Proxy VM			
VCPU	1 core		
Memory	5 GB		
OS	Debian 7.3 (Linux 3.2.0 64-bit)		
· Protection target VM			
VCPU	1 core		
Memory	1 GB		
OS	Debian 7.3 (Linux 3.2.0 64-bit)		

We used the model shown in Figure 5 for analyzing the above items. The description of the model is given below:

1. n protection target VMs and a proxy VM run on the VMM.
2. The number of essential processes running in each protection target VM is one.
3. n proxy processes run on the proxy VM, which is the same as the number of essential processes.
4. The proxy execution of a system call invoked by the essential process i ($1 \leq i \leq n$) is performed only by the proxy process i.
5. One virtual CPU (VCPU) is assigned to each VM and one unique physical CPU (PCPU) core is allocated to the VCPU respectively.

The environment used for evaluation is shown in Table 3. Machine 1 had an Intel Core i7-2600 (3.4 GHz, 4 cores) and 8 GB RAM. We used Xen 4.2.3 as a VMM on this machine. Furthermore, the proxy VM had one VCPU and 5 GB RAM. The protection target VM had one VCPU and 1 GB RAM. The number of cores is 4; therefore, n has a maximum value of 3. Debian 7.3 (Linux 3.2.0 64-bit) runs on the protection target VMs and proxy VM. Machine 2 had an Intel Core i7-4790 (3.6 GHz, 4 cores) and 8 GB RAM. Debian 9.3 (Linux 4.9.30 64-bit) runs on this machine.

Figure 6 shows the flow of the proxy process used for the evaluation. The difference with the flow of Figure 4 is step 3, 4, and 6. The proxy process checks the existence of requests with a

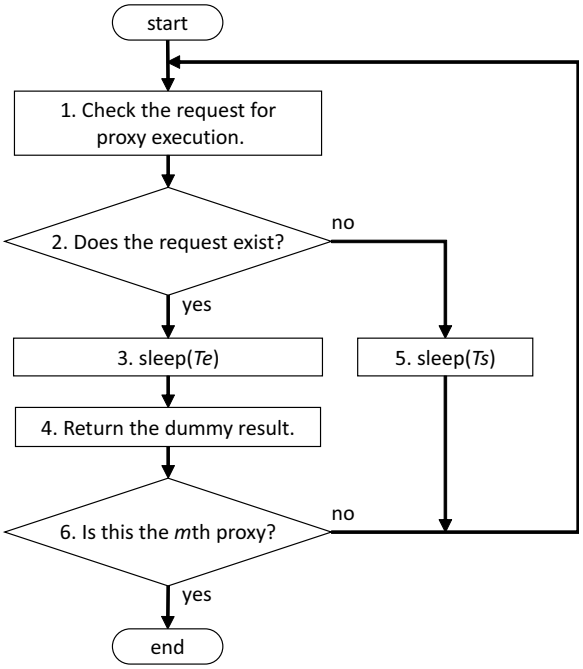

Figure 6: Flow of the proxy processes used for the evaluation.

constant period T_s. If such a request exists, the proxy process executes a sleep processing for T_e instead of executing the proxy system call. When setting T_s and T_e as 0, the system call overhead can be acquired by measuring the execution time of the system call on the essential process. After the sleep, the proxy process returns the dummy result to the essential process. Finally, the proxy process terminates its processing when the proxy execution is performed m times.

5.2 Visibility Test of Target System Calls

We tested whether the communication of an essential process is made invisible by the proposed method. In the test, we run an essential process on the protection target VM that communicates with a process on machine 2. From the result of monitoring the behavior of the essential process, we confirmed whether the communication is invisible on the protection target VM. In order to monitor the communication, we used the following tools:

1. ltrace: A tracing tool for monitoring library calls invoked by a specific process.
2. strace: A tracing tool for monitoring system calls invoked by a specific process.
3. tcpdump: A packet capture tool for monitoring TCP/IP network packets on the computer.

The essential process used in this evaluation performs simple communication that connects to the process on machine 2 and sends a message. The essential process invokes the following system calls related to communication:

1. socket(): Open a file descriptor.
2. connect(): Connect to the process on machine 2.
3. sendto(): Send a message to the process.
4. close(): Close the file descriptor.

Table 4: Result of the visibility test.

	Monitoring tool	Original	Proposed method
1.	ltrace	×	×
2.	strace	×	√
3.	tcpdump	×	√
		× Visible	√ Invisible

The result of this test is shown in Table 4. In the original system, communications of the essential process are visible with any tools. When applying the proposed method, the result of the test using strace or tcpdump is invisible. It is invisible to tcpdump because no packet is generated, as no system call is being executed on the protection target VM. It is invisible to strace because a breakpoint of the proposed method is set at the address before tracing the system call. Thus, it was confirmed that the communication of essential processes could be made invisible by the proposed method.

When applying the proposed method, the result of the test using ltrace is visible. It is visible to ltrace because the address where ltrace traces is set prior to the breakpoint set by the proposed method. In other words, although the system call is executed as a proxy, the result is visible because it is already monitored before the interposition. To avoid monitoring using library hook tools, it is required to set a breakpoint prior to the address where the library hook is set. If libraries are dynamically linked, we can set a breakpoint at the specific address. For example, when the executable file format is ELF, the address can be obtained based on the information in the .plt or .got section. However, because ltrace detects library calls by setting breakpoints in .plt section, it is necessary to set a breakpoint for each process. Additionally, if libraries are statically linked, we need to set a breakpoint for each program because library functions are already linked with executable files. Thus, there are several challenges to interpose the library calls before being detected by ltrace.

5.3 Behavior of the Proxy Processes Related to Intervals for Acquiring Requests and the Overhead of System Calls

The proxy process sleeps for the time T_s when a request for proxy execution does not exist. This means the proxy process tries to get the requests for proxy execution at the interval of T_s. We analyzed the behavior of the proxy processes by changing the T_s. Particularly, when three proxy processes execute proxy system calls ($n = 3$), we created graphs by superimposing timestamps acquired from essential processes and proxy processes for each case when $T_s = 1$ ms and $T_s = 0$ ms. Additionally, for evaluating the overhead of system calls, we set execution time (T_e) to 0 ms.

The essential process used for the evaluation repeats invoking socket() and close() system call alternately m times. Moreover, to analyze the behavior, the essential process gets timestamps t_a, t_b, and t_c in places, before invoking socket() call, between socket() and close() call, and after invoking close() call, respectively. The proxy process also gets timestamps (t_p) in a place before invoking the proxy system call. The overhead of socket() is calculated from the difference between the acquisition times of t_a and t_b, and the overhead of close() is calculated from the difference between the acquisition times of t_b and t_c. We calculated the overheads of the system calls when $n = 1, 2, 3$ respectively.

Figure 7 and Figure 8 show the behavior of the proxy processes when $T_s = 1$ ms and $T_s = 0$ ms, respectively. In both the graphs, timestamps t_a, t_b, t_c, and t_p shown in Figure 6 are plotted as $\bigcirc$, $\triangle$, $\times$, and $+$ in chronological order. The vertical axis indicates the index of essential processes and proxy processes; and the horizontal axis indicates the elapsed times since acquiring the first timestamp. The series of markers ($\bigcirc, +, \triangle$) or ($\triangle, +, \times$) indicate execution of socket() or close() by using the proposed method, respectively. In order to facilitate the analysis, m is set to 40 for $T_s = 1$ ms and to 1,000 for $T_s = 0$ ms. In Figure 7, proxy execution of socket() and close() are alternately repeated 40 times in each process. In Figure 8, because the proxy execution of socket() and close() are alternately repeated 1,000 times in each process, many markers are overlapped. These figures illustrate the followings:

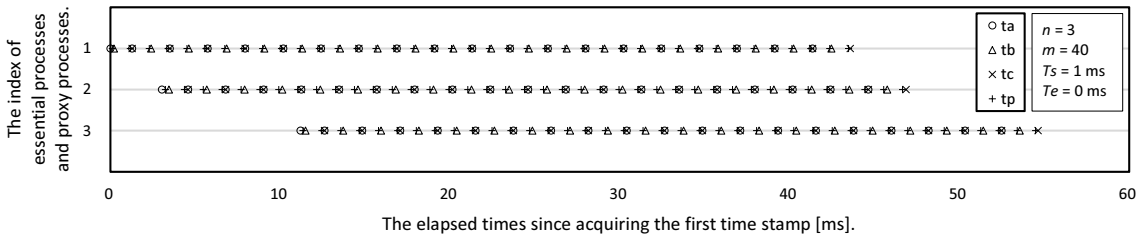

Figure 7: Behavior of essential processes and proxy processes. ($T_s = 1$ ms, $T_e = 0$ ms)

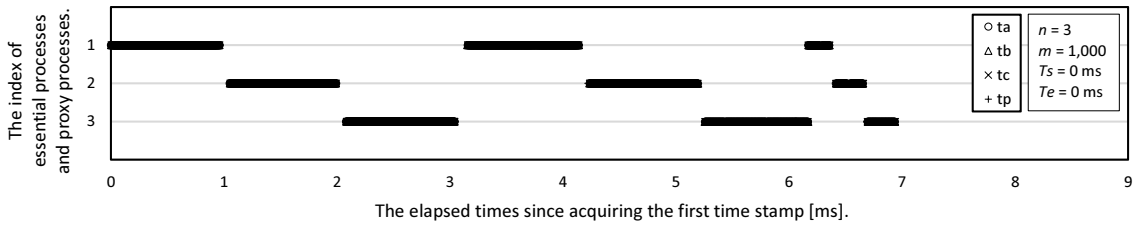

Figure 8: Behavior of essential processes and proxy processes. ($T_s = 0$ ms, $T_e = 0$ ms)

1. Because timestamps are arranged at regular intervals in Figure 7, the proxy execution for each essential process is alternately performed one at a time. This is owing to the reason that a timer interruption for process switching in a proxy VM and acquisition of request for the proxy execution occur alternately.

2. Unlike Figure 7, Figure 8 shows that the proxy executions are not performed alternately one at a time. Many proxy executions are performed at one time in each proxy process at high frequency during an interval of 1 ms. This is owing to the reason that the proxy process cannot be switched until a timer interrupt occurs.

Figure 9, Figure 10, and Figure 11 show the overhead of the socket() and close() calls when $T_s = 2$ ms, $T_s = 1$ ms, and $T_s = 0$ ms, respectively. These figures show the followings:

3. Figure 9 and Figure 10 show the overhead of a system call to be approximately 2,100 μs and 1,100 μs, respectively. This is due to the influence of sleep processing for T_s.

4. Figure 11 shows the least overhead of a system call to be approximately 2.1 μs.

5. All figures show that the changes in the overhead are small even if the number of proxy processes (n) increases. This is because there is no significant change in the height of each graph.

6. All figures also show that the influence on the overhead due to the difference between socket() and close() calls is small. This is because the shapes of the graphs (A) and (B) are almost the same.

It can be seen from 1 to 6 above, that when $T_s = 1$ ms and $T_e = 0$ ms, proxy system calls can be evenly executed for each essential process. However, the overheads of socket() and close() calls are influenced by sleep processing for T_s, which is approximately 1,100 μs for $T_s = 1$ ms and 2,100 μs for $T_s = 2$ ms. On the other hand, the overheads are approximately 2.1 μs when $T_s = 0$ ms and $T_e = 0$ ms. However, proxy system calls cannot be executed evenly for each essential process.

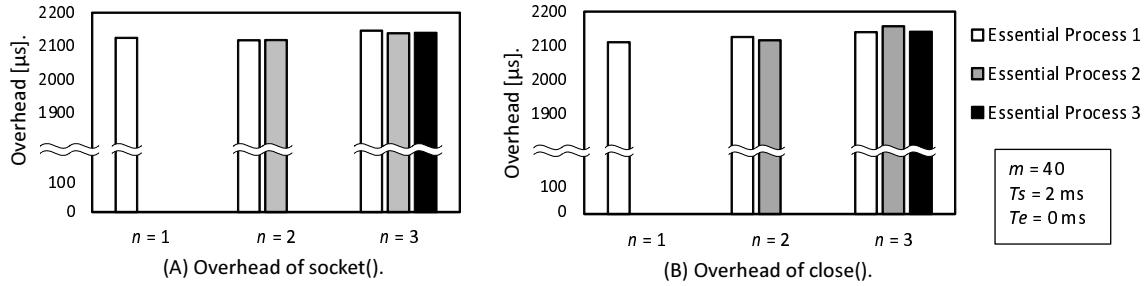

Figure 9: Overhead of socket() and close(). ($T_s = 2$ ms, $T_e = 0$ ms)

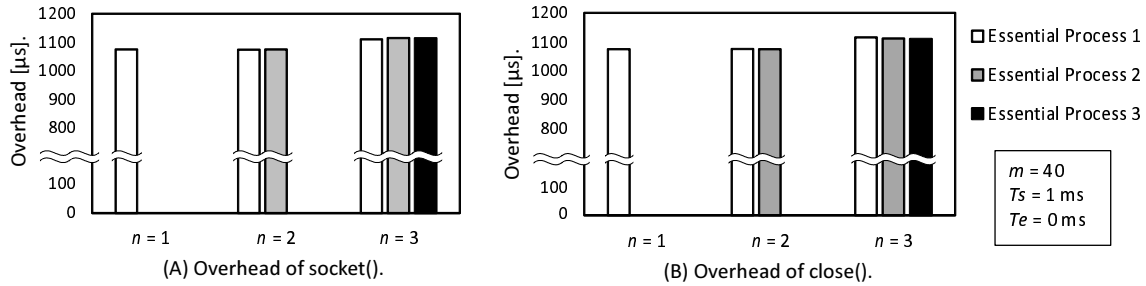

Figure 10: Overhead of socket() and close(). ($T_s = 1$ ms, $T_e = 0$ ms)

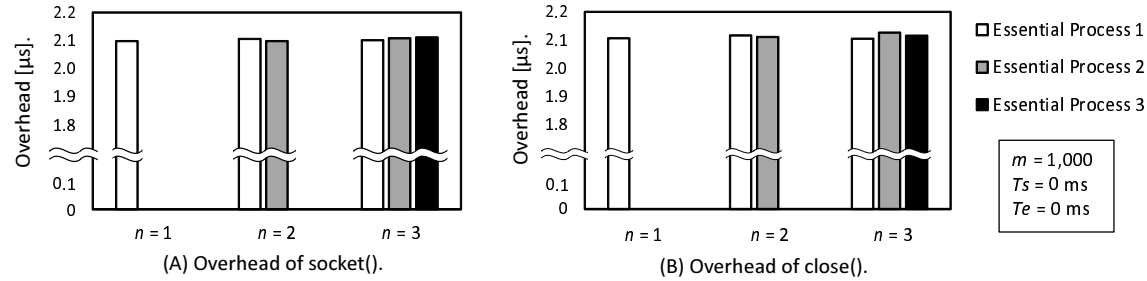

Figure 11: Overhead of socket() and close(). ($T_s = 0$ ms, $T_e = 0$ ms)

5.4 Behavior of the Proxy Processes Related to the Time of the Proxy Execution

In the previous section, we described the behavior of proxy processes by changing the sleeping time (T_s). In this section, we describe the behavior of the proxy processes by changing the execution time (T_e). Specifically, we set $T_s = 0$ ms and created graphs in the same way as before for each case when $T_e = 1$ ms and $T_e = 0$ ms.

In this evaluation, we consider that there are two types of system call processing that cause a context switch and that require CPU processing. In order to simulate these system call executions, we used a system call for sleep and a dummy system call for a busy loop as proxy execution performed by the proxy process.

Figure 12 and Figure 13 show the behavior of the proxy processes when $T_e = 1$ ms and $T_s = 0$ ms. In Figure 12, we used sleep() for T_e instead of invoking a system call as a proxy. On the other hand in Figure 13, we used a dummy system call which performs a busy loop for T_e. Because the graph for $T_e = 0$ ms has been already shown, we use Figure 8 again. These figures illustrate the followings:

1. In Figure 8, proxy executions are performed continuously until process switching owing to the

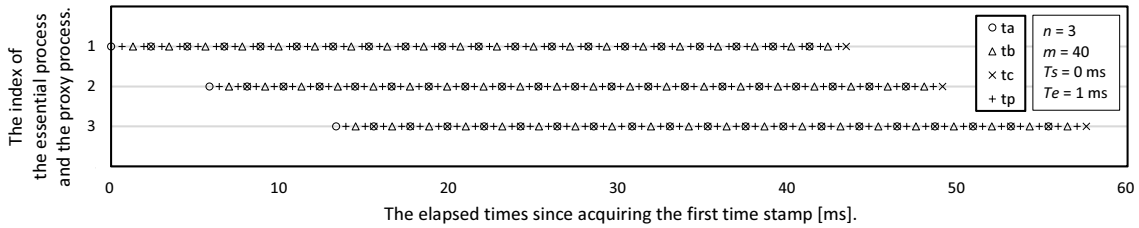

Figure 12: Behavior of essential processes and proxy processes. ($T_s = 0$ ms, $T_e = 1$ ms, Using sleep() instead of invoking a system call as a proxy.)

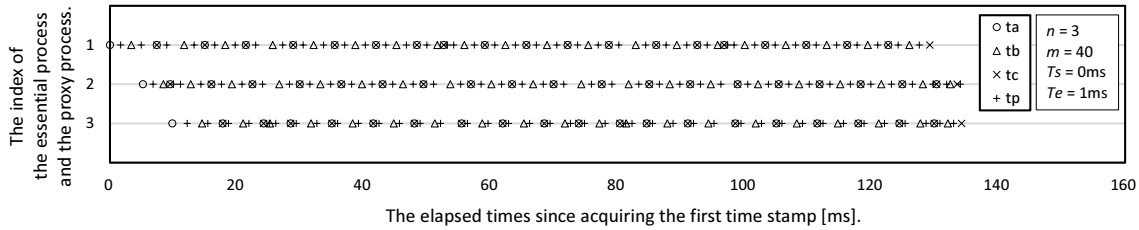

Figure 13: Behavior of essential processes and proxy processes. ($T_s = 0$ ms, $T_e = 1$ ms, Using a dummy system call (busy loop) instead of invoking a system call as a proxy.)

timer interrupt occurs; therefore, it is impossible to execute the proxy system calls evenly for each essential process.

2. Both Figure 12 and Figure 13 show that the proxy executions are alternately performed one at a time, as in Figure 7. This is also because a timer interruption for process switching in a proxy VM and acquisition of request for the proxy execution occur alternately. Therefore, even if a system call causes a context switch or requires CPU processing, multiple requests for proxy execution can be handled evenly.

From points 1 and 2 above, when $T_s = 0$ ms and $T_e = 1$ ms, proxy system calls can be executed evenly for each essential process. However, this cannot be done when $T_s = 0$ ms and $T_e = 0$ ms.

In summary, to execute proxy system calls evenly for each essential process, it is necessary to set T_s or T_e greater than 0. However, because the execution times of system calls are actually different, T_e can be close to 0. Therefore, it is necessary to set T_s greater than 0. Additionally, because there is a trade-off between the value of T_s and the equality of proxy execution, it is the best to make the overhead as small as possible and to be able to execute system calls evenly for each proxy request. The timer-interrupt frequency of the Linux kernel we used for the evaluation is 1 ms, it cannot sleep for less than 1 ms. Therefore, $T_s = 1$ ms is optimal in this environment.

5.5 System Call Overhead Regarding the Size of the Data Transferred Between VMs

To analyze the performance of the proposed method, we evaluated the overhead of the sendto() system call regarding its sending data size. In the proposed method, the number and arguments of the system call invoked by an essential process are transferred to a proxy process. When transferring data, a memory copy will occur twice: once from the protection target VM to the VMM and once from the VMM to the proxy VM. The processing time for these copies increases as the size of arguments increases. Therefore, we evaluate how the size of the arguments affects the overhead of the system call. We used sendto() as a system call whose argument size can be changed. The size of the sending data is to be 1 B, 1 KB, 2 KB, 3 KB, and 4 KB. To remove the unnecessary processing time for the evaluation of the overhead, we set T_s and T_e as 0. Additionally, to evaluate the load when the number of the protection target VMs increases, we set $n = 1, 2, 3$.

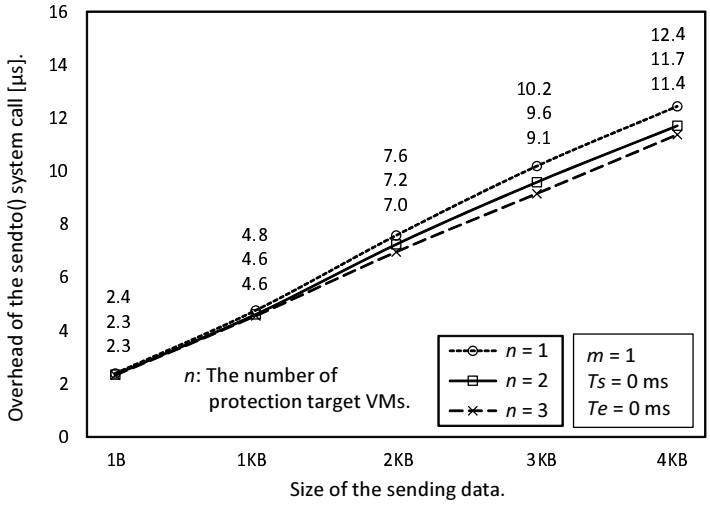

Figure 14: Overhead of the sendto() system call.

Figure 14 shows the overhead of the sendto() system call. For each n, the overhead increases in proportion to the size of the sending data. Because the overhead is 2.3 μs when the size of the sending data is 1 B, the overhead of the sendto() system call has a certain overhead of approximately 2.3 μs. In addition to the specific overhead, time is needed to copy the sending data, so the overhead of the sendto() system call has the following relationship:

$$O \approx 2.3[\mu s] + c \times (s - 1). \tag{1}$$

O: Overhead of the sendto() system call
c : Constant
s : Size of the sending data

The constant c corresponds to the slope of the graph. From the graph, the constant increases as n increases. This is because the waiting time until the CPU is allocated to the proxy process has increased due to an increase in the number of proxy processes.

5.6 Performance When System Calls are Actually Executed

To analyze how the system call overhead affects the communication performance when applying our proposed method, we evaluated the processing time of the sendto() system call 1,000 times. For comparison, the original system call processing time was measured on the protection target VM and proxy VM. In this evaluation, the proxy process operates with the processing flow shown in Figure 4; therefore, system calls invoked by an essential process are actually executed as a proxy. The parameter of the proxy process is set as $T_s = 0$ ms. The number of protection target VMs is 1 ($n = 1$). In machine 2, the process of repeating recvfrom() system calls runs. An essential process for this evaluation repeatedly invokes 1,000 sendto() system calls to the process. The size of the sending data when invoking the sendto() system call is 512 B, 1 KB, 1.5 KB, and 2 KB. Because the sendto() system call is invoked 1,000 times, the total size of the sending data is 1,000 times these sizes.

In the evaluation, the proxy VM is domain 0, which is a VM that manages the VMM and other VMs in Xen. Because domain 0 manages hardware drivers, all VMs access the NIC via domain 0's NIC driver. In the original protection target VM, sending data is transferred to domain 0 via the netfront driver, which is a type of Xen split drivers. Xen split drivers are a mechanism for transferring data between VMs. The sending data is transferred from the netfront driver to the

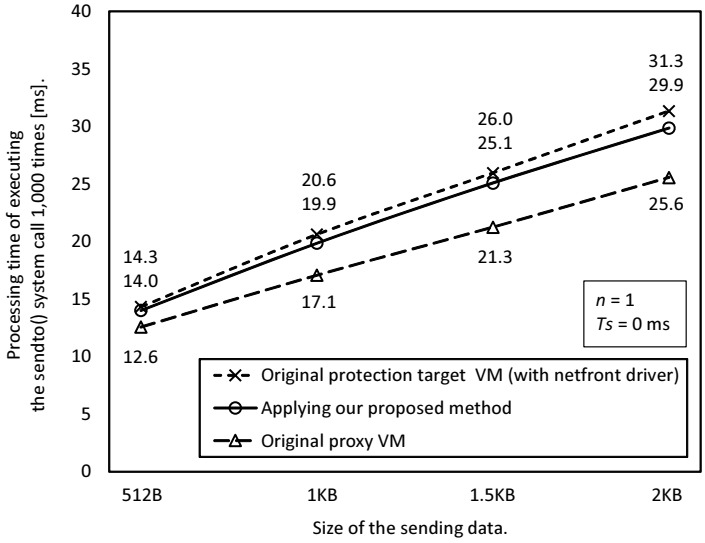

Figure 15: Processing time of executing the sendto() system call 1,000 times.

netback driver on domain 0 via the shared memory. This data transfer using the Xen split drivers is faster than device emulation by QEMU. In our proposed method, because the VMM interposes a system call and transfers the argument from the essential process on the protection target VM to the proxy process on the proxy VM, the data transfer via the Xen split drivers does not occur.

Figure 15 shows the processing time of executing the sendto() system call 1,000 times. Because the graph is a straight line, each processing time increases in proportion to the size of the sending data. The processing time of the original proxy VM is the smallest. The processing time of the original protection target VM and that of applying our proposed method are similar. The difference between the processing time of the original proxy VM and the time of the original protection target VM occurs because of the data transfer from the protection target VM to the proxy VM. On the other hand, the difference between the processing time of the original proxy VM and that of applying our proposed method is occurred by the system call overhead of the proposed method. The processing time of applying our proposed method is approximately 4% less than that of the original protection target VM. Therefore, it is faster to transfer the sending data from the protection target VM to the proxy VM using the proposed method than using the netfront driver. Furthermore, although not shown in Figure 15, we evaluated the processing time of the original protection target VM with device emulation by QEMU. The results showed that the processing time is 40.6 ms when the size of the sending data is 512 B and 142.3 ms when the size is 2 KB. Therefore, it was found that the processing time was much longer than the results in Figure 15 and the slope is also very large.

6 Related Work

A system for proxy execution is proposed. Ta-Min et al. proposed a system called Proxos [11] that allows applications to configure their trust in the OS by partitioning the system call interface into trusted and untrusted components. In the Proxos, the trusted system calls are executed in the trusted private OS, the untrusted system calls are executed in the untrusted commodity OS. Thereby, the applications can be operated safely on the private OS.

Both our proposed method and Proxos are aiming the protection of a specific process and executing system calls in another VM. In contrast, there are differences in whether it is necessary to modify the original environment. In order to realize the Proxos system, it is necessary to modify the existing guest OSes. On the other hand, our method can be realized without modifying the existing guest OSes and applications.

7 Conclusion

We proposed a communication-hiding method for avoiding the identification of essential services by using a system call proxy. The method is implemented by a VMM and a proxy process on the proxy VM. System calls in the communication invoked by the essential process running on a protection target VM are executed by the proxy process. The result of the execution is returned to the essential process; therefore, the essential process resumes its processing as if the system calls are executed in the local kernel. However, attackers in the protection target VM cannot detect the system calls because the system calls are not executed on the protection target VM. Thereby, it is possible to hide the communication of essential services from attackers. Additionally, it is difficult to detect this method even if an attacker gets root privilege on the VM because the VMM and VMs are isolated from each other.

In the evaluation, we showed that communication of essential processes can be made invisible by the proposed method, even if when using monitoring tools such as strace or tcpdump. In addition, we evaluated the performance overhead of the proposed method. The evaluated result showed that the interval to check the request for proxy execution needs to be larger than 0 in order to execute the proxy system calls evenly for each essential process. In addition, we showed that the performance overhead of the sendto() system call is approximately 2.3 μs at the least and increases in proportion to the size of the sending data. The performance of the sendto() system call in the proposed method is better than executing system call on VM as-is.

We have two future works as described below. First, complete emulation of communication in the proxy process needs to be realized. The file descriptor and the source IP address of the proxy VM, not the protection target VM, are used for proxy executions. Second, to make the essential services less visible, it is necessary to deal with a case where the system calls are monitored at places other than bypassed by the proposed method.

Acknowledgment

This work was partially supported by KAKENHI grant numbers JP18K18051.

References

[1] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 164–177, New York, NY, USA, 2003.

[2] Artem Dinaburg, Paul Royal, Monirul Sharif, and Wenke Lee. Ether: Malware analysis via hardware virtualization extensions. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, CCS '08, pages 51–62, New York, NY, USA, 2008.

[3] Tal Garfinkel and Mendel Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Network and Distributed Systems Security Symposium*, volume 3, pages 191–206, 5 2003.

[4] Fu-Hau Hsu, Min-Hao Wu, Chang-Kuo Tso, Chi-Hsien Hsu, and Chieh-Wen Chen. Antivirus software shield against antivirus terminators. In *IEEE Transactions on Information Forensics and Security*, volume 7, pages 1439–1447, 2012.

[5] Byungho Min and Vijay Varadharajan. A novel malware for subversion of self-protection in anti-virus. In *Softw. Pract. Exper.*, volume 46, pages 361–379, New York, NY, USA, 5 2016.

[6] Jonas Pfoh, Christian Schneider, and Claudia Eckert. Nitro: Hardware-based system call tracing for virtual machines. In *Advances in Information and Computer Security*, pages 96–112, Berlin, Heidelberg, 2011.

[7] Masaya Sato, Hideo Taniguchi, and Toshihiro Yamauchi. Hiding file manipulation of essential services by system call proxy. In *Lecture Notes on Data Engineering and Communications Technologies*, volume 22, pages 853–863. The 7th International Workshop on Advances in Data Engineering and Mobile Computing (DEMoC-2018), 9 2018.

[8] Masaya Sato, Toshihiro Yamauchi, and Hideo Taniguchi. Process hiding by virtual machine monitor for attack avoidance. In *Journal of Information Processing*, volume 23, pages 673–682, 9 2015.

[9] Deepa Srinivasan, Zhi Wang, Xuxian Jiang, and Dongyan Xu. Process out-grafting: An efficient "out-of-vm" approach for fine-grained process execution monitoring. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, pages 363–374, New York, NY, USA, 2011.

[10] Stealth. A new adore root kit. http://lwn.net/Articles/75990/.

[11] Richard Ta-Min, Lionel Litty, and David Lie. Splitting interfaces: Making trust between applications and operating systems configurable. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 279–292, Berkeley, CA, USA, 2006.

[12] Junqing Wang, Miao Yu, Bingyu Li, Zhengwei Qi, and Haibing Guan. Hypervisor-based protection of sensitive files in a compromised system. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, SAC '12, pages 1765–1770, New York, NY, USA, 2012.