Design and implementation of hiding method for file manipulation of essential services by system call proxy using virtual machine monitor

Masaya Sato*

Graduate School of Natural Science and Technology, Okayama University,
Okayama, JAPAN
E-mail: sato@cs.okayama-u.ac.jp
* Corresponding author

Hideo Taniguchi

Graduate School of Natural Science and Technology, Okayama University,
Okayama, JAPAN
E-mail: tani@cs.okayama-u.ac.jp

Toshihiro Yamauchi

Graduate School of Natural Science and Technology, Okayama University,
Okayama, JAPAN
E-mail: yamauchi@cs.okayama-u.ac.jp

Abstract: Security or system management software is essential for keeping systems secure. If these essential services are stopped or disabled by attackers, damages to the system increase. Therefore, protecting essential services is crucial for preventing and mitigating attacks. To deter attacks on essential services, hiding information related to essential services is helpful. This paper describes design and implementation of a method to make files invisible to all services except their corresponding essential services and provides access methods to those files in a virtual machine (VM) environment. The proposed method consists of interposition and proxy execution of the system call function. In the proposed method, the virtual machine monitor (VMM) monitors system calls invoked in a protection target VM. If an essential process invokes system calls related to file manipulation, the VMM interposes the system call and collects information from the protection target VM. If the file is an essential file, the VMM requests proxy execution to the proxy VM on another VM. After proxy-execution of the system call, the proxy process returns the result to the VMM. Finally, the VMM returns the result and skips the execution of the original system call on the protection target VM. Thus, access to essential files by the essential service is skipped on the protection target VM, but the essential service can access the file content. With this mechanism, it is difficult for attackers to monitor access to essential files. In this paper, we describe the design, implementation, and evaluation of the proposed method.

Keywords: virtual machine monitor; file manipulation; system call proxy; essential services

Reference to this paper should be made as follows: Masaya Sato, Hideo Taniguchi, and Toshihiro Yamauchi. (2019) ‘Design and Implementation of Hiding Method for File Manipulation of Essential Services by System Call Proxy using Virtual Machine Monitor’, *International Journal of Space-Based and Situated Computing*, Vol. x, No. x, pp.xxx–xxx.

Biographical notes: 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 the Graduate School of Natural Science and Technology at Okayama University. His research interests include computer security and virtualization technology. He is a member of IEICE and IPSJ.

Hideo Taniguchi received a B.E. degree in 1978, a M.E. degree in 1980 and a Ph.D. degree in 1991, all from Kyushu University, Fukuoka, Japan. In 1980, he joined NTT Electrical Communication Laboratories. In 1988, he moved to Research and Development Headquarters, NTT DATA Communications Systems Corporation. He has been an Associate Professor of Computer Science at Kyushu University since 1993 and a Professor of the Faculty of Engineering at Okayama University since 2003. He has been a Dean of Faculty of Engineering from April 2010 to March 2014 and a Vice President from April 2014 to March 2017 at Okayama University. His research interests include operating systems, real-time processing and distributed processing. He is a fellow of IPSJ. He is a member of IEICE and ACM.

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 the 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.

1 Introduction

Security software plays an important role in computer system operation. Malicious software, or malware, disables functionality of the system or may cause leaks of classified information. Stopping a system or leaks of sensitive information, including personal information, cause serious damage to services and companies. Therefore, security software is one of the most important tools in computer system operation. Logging tools are also important for system operation because they record logs of systems, which are beneficial for efficient and secure operation of the system. Monitoring tools are critical because they detect illegal behavior within the system to keep it stable. These tools are also used for system security. Detecting illegal activities may lead to detection of attacks, and immediate attack detection prevents the damage to the system. In this paper, we define these software tools as essential services. As stated by Min et al. (2014) and Gregio et al. (2015), protecting essential services is crucial to preventing attacks and limiting the spread of damage caused by malware.

Attackers target essential services and prepare the attack during a reconnaissance phase. Although some methods to protect essential software have been proposed by Hsu et al. (2012), Garfinkel et al. (2003), and Srinivasan et al. (2011), none consider the identification of essential services. This paper asserts that if the identification of essential services were more difficult, then attacking essential services would also become more difficult.

To identify an essential service, process, file, and communication information may be used. Process information includes process name, process ID (PID), or other information managed by the operating system (OS). File information includes files and file manipulation information. If an attacker can monitor file access, file manipulations will provide strong clues to identify what service is provided by the process. For example, an attacker can discover that a process is an essential service if it accesses configuration files or a database related to essential services. Communication information includes host name, IP addresses, port numbers, and communication content. An attacker could determine that a process is security software if that process accesses to a vendor's servers to update their signature for malware detection.

Sato et al. (2015) previously proposed method to complicate the identification of essential services based on process information. Thr prior method involves monitoring context switches and replacing process information while the process is not running. Then, other processes cannot identify the process as an essential service because they cannot view the original information. The other processes view the process providing the essential service (essential process) as simply providing other dummy services.

We propose a hiding method for file manipulation by system call proxy. Though the prior method makes process information invisible from attackers, files and communication remain visible. The proposed method hides files related to essential services (essential files) and access to essential files by the processes providing the essential services. With the proposed method, essential files are only visible to essential processes, so other processes cannot find essential files. The proposed method employs a virtual machine monitor (VMM) and another proxy virtual machine (VM) which provides proxy access to essential files. The VMM monitors system calls related to file manipulation on a target VM. If the process that invokes a system call is an essential process, the VMM copies information of the system call from the target VM and transfers it to the proxy VM. The proxy process on the proxy VM invokes the system call and returns the result to the VMM. The VMM sets the result and returns processing to the starting point of the function to return to user mode. With this protocol, the essential process resumes as if it had accessed the required file in local storage, but other processes cannot interpose or monitor system calls. As a result, it is much more difficult for attackers to identify essential services by monitoring file access.

This paper describes design, implementation, and evaluation of the proposed method. Design and implementation assume Linux as a guest OS and Xen as a VMM. We also evaluated the overhead of the proposed method for basic system calls and security software.

2 Related Work

Protecting security software is crucial, therefore researchers have proposed protection methods. Hsu et al. (2012) analyzed attacks on security software and proposed an attack-tolerant method for protecting security software called Antivirus Software Shield (ANSS). This method prevents termination of security software by monitoring and controlling system calls that attempt to terminate the security software. ANSS is implemented as a kernel mode driver of Windows OS. Placing protectoin inside the OS is more secure than implementing as an application program. However, certain malware, including rootkits, exploit kernels to achieve their goal. In contrast, the proposed method uses a VMM and another VM. Implementing a security mechanism at a VMM is even more secure than implementing it within the OS kernel.

Next, malware analysis methods using a VM are reviewed and compared with the proposed mthod. Garfinkel et al. (2003) proposed virtual machine introspection (VMI) to monitor inside the VM from the outside. In VMI, security software on one VM monitors the other VM from the outside. Dinaburg et al. (2008) proposed Ether, which is a method that analyzes the behavior of malware on one VM from another VM. Ether monitors system calls and memory access on the monitored VM. Ether requires no modification to the guest OS and thus can analyze Linux and Windows. Srinivasan et al. (2011) proposed the process out-grafting as a method to analyze malware while keeping the monitoring environment secure. This method extracts the user mode execution of the target process to a security monitoring VM. The security monitoring applications run on the security monitoring VM and analyzes user mode execution. These methods focus on the secure analysis of malware, where we focus on protection of security software on a VM that attackers want to compromise. Because information can be collected inside the VM, this method is more efficient for analysis and protection than performed outside the VM. We do not focus on moving security mechanisms from inside the VM to outside, but on protecting security mechanisms remaining inside the VM. System calls of the monitoring target process are returned back to the original VM, so the behavior of the process does not affect the behavior of processes on the security monitoring VM.

Wang et al. (2012) proposed Filesafe, which is a method to protect files by a VMM. Filesafe controls file access according to a policy held by the VMM. Filesafe is similar to the proposed method in terms of access control. However, the proposed method also hides the existence of files used by essential services. Hegarty et al. (2015) proposed XDet, a method for extrusion detection of illegal files in cloud-based systems. Hegarty et al. pointed out privacy and trust issues for cloud providers in that the files in the cloud environment can be related to malicious activities or geographical legalities. Both XDet and our approach focus on protection of environment, but the approach to files is different. Hegarty et al. focused on detection of problematic files in the cloud network environment. In contrast, we focus on the hiding of specific files from attackers in order to hide essential services.

Deception technique has also been applied to malware analysis. A method using deception to analyze and protect attacks was proposed by Almeshekah et al. (2014). Specifically, this method uses deception to inpel attackers toward honeypots, that is, analysis environments that mimic the production environment. Araujo et al. (2014) proposed an advanced analysis method targeting attacks aiming at vulnerabilities. This method responds as if the attack were successful and transfers the request to a honeypot to monitor and analyze advanced attacks. Our approach is similar to the deception technique, however the proposed method uses avoidance to hide the security software to protect systems from attack.

3 Attacks to Essential Services

3.1 Essential Services

As previously mentioned, we define security software and system management tools as essential services. Correspondingly, a process providing an essential service is defined as an essential process and a file holding sensitive information used by an essential process is an essential file. Security software is an example of a typical essential service. Per the preceding definitions, each process run by security software is an essential process, and the configuration files or white-list files used by the service are considered to be essential files. If an attacker modifies configuration files or white-list files, the security software cannot detect malware because these files are not scanned for malware. This compromises the computer, making the system vulnerable to further attacks. Furthermore, if files for security software are visible to attackers, attackers may be able to identify the security software and invoke additional exploits. Because attackers often start by detecting and preparing attacks based on essential services, hiding essential services from attackers reduces the possibility of attacks. Although hiding the existence of a process reduces the possibility of the discovery of essential services, the existence of essential files remains a problem. Because attackers can discover the existence and extent of essential services from the essential files, hiding these files and the processes that manipulate them would provide a valuable enhancement to system security.

3.2 Existing Protection Method of Security Software

As stated in Section 2, Hsu et al. (2012) proposed the ANSS, which protects security software by monitoring and controlling application programming interfaces (API) on Windows. This method adds a kernel-mode driver into Windows and monitors API calls. If an API call attempts to terminate the security software, the driver fails the call and protects the security software from termination by attackers.

Another method for protecting security software uses virtualization technology. Srinivasan et al. (2011) proposed a method to monitor the behavior of a process inside a VM from the outside. This method relies on the assumption that the VM and the VMM are isolated from each other. The security software runs on one VM and monitors the other VM. Because each VM is isolated, the security software is safe from attacks on the other VMs. However, integrating the existing security software with the outside VM causes a semantic gap, which is the difference in perspective between the inside and outside of the VM.

Wang et al. (2012) proposed Filesafe to protect classified files using virtualization technology. Filesafe is implemented as a VMM and enforces security policy to the guest OS. If Filesafe restricts access to specific files, processes cannot access the file even though it is permitted by the guest OS. Because Filesafe is resistant to attacks on the OS, it can prevent leaks of sensitive information.

3.3 Problems

Many of the previously discussed methods introduce security mechanisms into the OS kernel. This can make the system vulnerable to attackers who have kernel data access privilege. Sophisticated attackers exploit kernel vulnerabilities to disable functionalities in order to take further control. VMs are widely used in currently, so utilizing virtualization technologies for security is a practical approach. Some methods are implemented using a VMM but do not focus on protecting the software inside the VM. This approach also requires integration of security software the outside the VM. To utilize existing security software, integration and modification to existing security software is undesirable.

We also consider the visibility of files related to essential services. Visibility is defined as the possibility of attackers discovering the files, so preventing attackers from knowing the names of essential files can impede attacks. Once essential files are identified, attackers can discover the existence of essential processes by monitoring essential files and their manipulation. Thus, hiding the existence of those files may reduce the detection of essential services. Although Filesafe effectively protects files from unauthorized access, but the files remain visible to attackers. If these files and their manipulation are visible to attackers, a process accessing those files could be flagged as a potential essential process. Unlike existing access control approaches, the proposed method considers visibility.

4 File Manipulation Hiding

4.1 Purpose

The purpose of this paper is to complicate identification based on file information of essential services from attackers. Although a hiding method for process information, which includes process ID or process name, is proposed, file information is still visible to attackers. The file information includes the files themselves and file manipulation. For example, even though access to files related to essential services is restricted, an attacker can conclude that a process which accesses them is a part of essential services. Thus, monitoring file access enables attackers to identify the essential service. One example of an essential file is a white-list for antivirus software. Antivirus software scans files and compares them with signatures to detect known malicious files. However, a white-list is used to exclude specific directories or files from scanning because of performance and operational issues. If an attacker exploits this functionality, he can identify a process which read a white-list file as an essential service. Therefore, file manipulations also need to be invisible to attackers. The Linux kernel provides some methods to monitor file access by a process from other processes. `ptrace` is one example of an access monitoring method provided by Linux.

To address these problems, we propose a method to prevent the identification of an essential service by monitoring essential files. This approach makes files invisible to attackers and only allows essential processes access to the files. The proposed method executes without modifying the essential service. This attribute addresses the problem in existing methods which require the modification or integration of existing security software.

4.2 Requirements

The followings requirements are defined to address the previously mentioned problems:

R1 Essential files are made invisible to attackers.

R2 The method that satisfies R1 is also invisible to normal processes or kernel modules, and thus invisible to attackers as well.

If the method were detected, attackers could discover the essential service by monitoring the detected method. We assume that attackers insert their programs into normal processes. A normal process is any process other than an essential process or kernel modules. Therefore, hiding the method from normal processes and kernel modules is an important aspect of R2.

4.3 Hiding Method for File Manipulation

4.3.1 Challenges

To fulfill the requirements, the following challenges should be addressed:

C1 Interposition of file access related to essential services.

C2 Control of interposed file access.

C3 Addressing C1 and C2 is undetectable by normal processes or kernel modules.

C1 and C2 are required satisfy R1. In addition, to satisfy R2, hiding the method addressing C1 and C2 from attackers is required.

4.3.2 Interposition of file access for essential files

The following methods are considered for the approach of interposing file access: library call, system call, the processing of a file system and the processing of device access. In a normal file access procedure, a process calls library functions, the library functions request processing to the kernel by invoking system calls, and finally the kernel accesses the files.

We monitored file access by interposing system calls. To hide essential files from attackers, interposing library calls is ideal. This is because we can reduce the chance of file manipulation monitoring as fast as we can interpose. However, it is difficult to interpose all library calls related to file access. Libraries may be dynamically or statically linked to a program. If a library is dynamically linked, we can interpose file access by monitoring function calls to specified memory regions. However, if a library is statically linked, the relevant memory regions differ for each program. Identifying all the library calls to monitor of all programs is impractical. In contrast, it is possible to monitor all access by monitoring file systems or device drivers. However, these procedures occur in the latter part of file access. If an attacker monitors or modifies a file manipulation before it reaches the latter part, essential services may be identified. For this reason, our method interposes system calls for file access.

To handle C3, we implement the solution using a VMM. The VMM and VMs are isolated from each other. Thus, non-intrusively interposing file access of a VM from a VMM helps hide the mechanism itself from attackers on the VM.

4.3.3 Control of file access

As the previous section stated, the proposed method interposes system calls to hide files and file manipulations. This section details the control method. File access is made by file handlers, and processes use file handlers to manipulate files. Accordingly, file access is regulated by controlling the acquisition of file handlers. The proposed method interposes a system call to obtain a file handler. If the target file of the system call is an essential file, the proposed method checks the process. If the process is an essential process, the proposed method returns a file handler. If the process is a normal process, the proposed method returns a failure response. To continue the file access related to the returned file handlers, the proposed method controls the file access via the handlers. Thus, only an essential process can access essential files.

4.3.4 File placement on the outside the VM

To address C3, we employ a VM. While essential processes run on the OS on one VM, essential files are placed on another VM. Because essential files are placed on the other VM, it is difficult to identify or modify the files even if an attacker has root privilege. In addition, an attacker cannot find the essential files by raw device access because the files are not located on the same virtual disk. In this architecture, an essential process cannot access essential files. To enable an essential process to access the essential files, we propose a system call proxy, which is a method to transfer file access to the other VM. Section 5.5 details the system call proxy mechanism. Because we assume that essential files are files that are only required for essential services, hiding these files from normal processes does not obstruct the normal processes.

5 Implementation

5.1 Environment

We used Xen proposed by Barham et al. (2003) as a VMM and the VMs were fully virtualized with Intel VT-x. We assumed Linux as a guest OS and system calls were invoked with SYSENTER instruction in the 32bit and SYSCALL instruction in the 64bit environments. Figure 1 provides an overview of the proposed architecture. The VMM monitors system calls invoked in the protection target VM. If the system call invoked by the essential process is related to file manipulation, the VMM transfers the information to the proxy process on the proxy VM. The proxy process executes the system call based on the information and returns the result to the VMM. Finally, the VMM returns the processing to the protection target VM.

5.2 System Call Interposition

We used a debug exception to detect invocation of system calls in the VM from the VMM. In the proposed method, the VMM set a hardware breakpoint at the

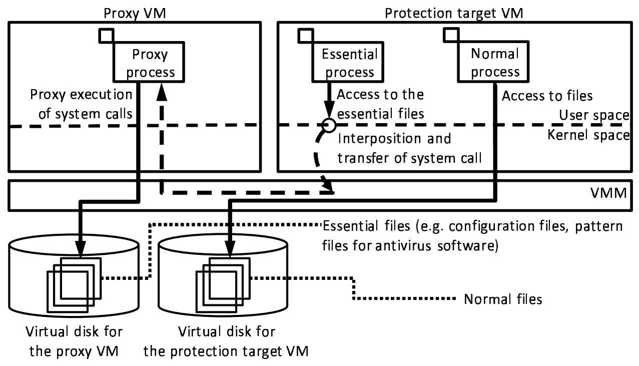
Proxy VM
Protection target VM
Proxy process
Essential process
Normal process
Proxy execution of system calls
Access to the essential files
Access to files
User space
Kernel space
Interposition and transfer of system call
VMM
Essential files (e.g. configuration files, pattern files for antivirus software)
Normal files
Virtual disk for the proxy VM
Virtual disk for the protection target VM

Figure 1 Overview of the hiding method of file manipulation by system call proxy.

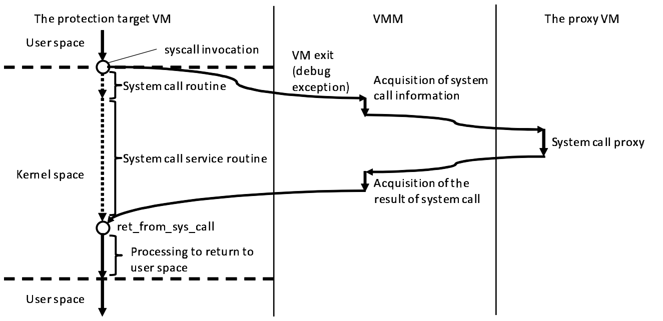
The protection target VM
VMM
The proxy VM
User space
syscall invocation
VM exit (debug exception)
System call routine
Acquisition of system call information
System call proxy
Kernel space
System call service routine
Acquisition of the result of system call
ret_from_sys_call
Processing to return to user space
User space

Figure 2 The flow and mode transition in the proxy execution of system calls.

starting address of the system call routine. In addition, the VMM manipulated the VM execution control field to cause VM exit when a debug exception occurred. In this case, when an instruction at the address was executed, a debug exception caused a VM exit. The VMM interposed the system call by managing the VM exit at the designated address.

In addition to the mechanism for system call interposition, classification of system calls is required. The VMM interposes all system calls invoked in a VM, however the proposed method requires knowledge of the system calls related to file manipulation. To classify the system calls, the VMM chooses system calls by a system call number. The VMM acquires the system call number from the RAX register. Accordingly, the VMM can interposes system calls of the protection target VM that are specifically related to file manipulation.

5.3 Control of System Calls Related to File Manipulation

Figure 2 shows the flow and mode transition in the proxy execution of system calls. When system calls are invoked by processes on the protection target VM, VM exit occurs. If it needs to request proxy execution, the VMM requests proxy execution to the proxy process on the proxy VM, and mode transition occurs. The proxy process invokes system calls as proxy and returns the result to the VMM. The VMM returns the processing to the starting point of the returning system call routine in the kernel of the protection target VM. This causes the original system call routine to be bypassed and return value and buffers are passed to the essential process.

Figure 3 shows the detailed flow of VMM to control system calls and request proxy execution. The following steps lay out thte control flow of the interposed system call.

1. The VMM detects system call invocation on the protection target VM by checking VM exit reason and register states. If the VM exit occurred by a debug exception and the address is the address previously set by the VMM, the VM exit was caused by system call invocation.
2. The VMM distinguishes the current process as an essential process or not. The VMM holds a table for managing essential services, and the table contains the value of CR3 register for each essential process. At start time, the manager of the proxy VM designates which process is an essential process. Then, the value of CR3 is stored into the table.
3. The VMM classifies the interposed system call. If the system call is related to file manipulation, the VMM proceeds to Step (4). If not, the VMM moves to Step (14). Because this paper focuses on file manipulation, network communication is omitted from the flow. Network communication issues will be addresses in our future work.
4. The VMM gets arguments of the system call from the protection target VM to the VMM. In this step, if the arguments contain pointers, the VMM needs to copy the data from the pointed area of the VM to the VMM.
5. The flow branches based on the type of file manipulation: open (Step (6)) or read/write/close (Step (7)).
6. The VMM distinguishes the files to open as an essential file or not. Because the VMM holds the table of essential files, the file and the entries of the table are compared. If the file is not an essential file, the VMM moves to Step (14).
7. If the file descriptor (FD) is already registered to the FD list held by the VMM, the VMM proceeds to Step (8). If not, the VMM moves to Step (14).
8. The VMM requests proxy execution of the interposed system call to the proxy. Figure 6 details proxy execution.
9. The VMM receives the result of the proxy execution list from the proxy process.
10. If the result is successful, the VMM proceeds to Step (11). If not, the VMM moves to Step (15).
11. The flow branches based on file manipulation: open (Step (12)), read/write (skipping FD managements and moves to Step (15)), and close (Step (13)).

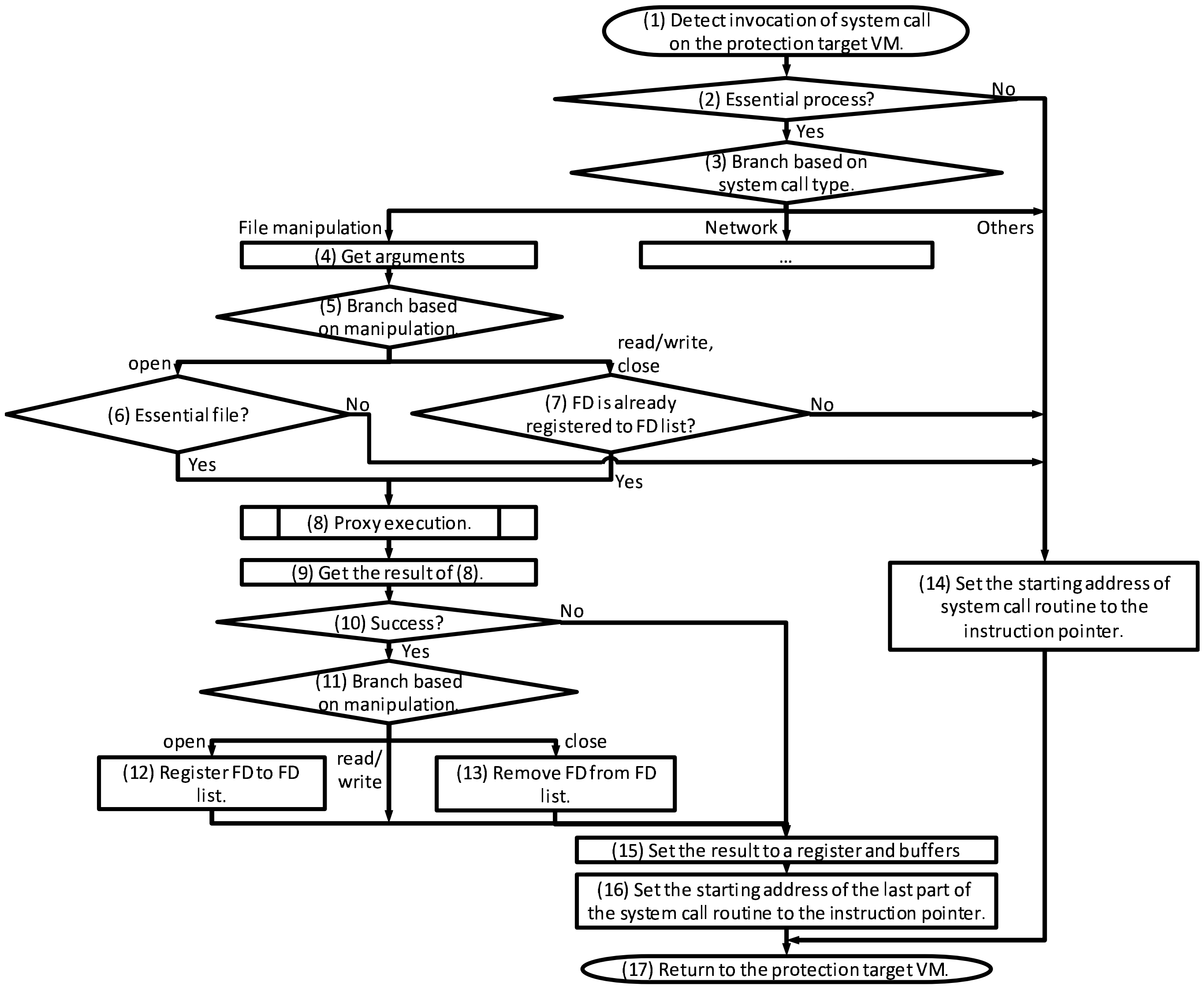
(1) Detect invocation of system call on the protection target VM.
(2) Essential process?
No
Yes
(3) Branch based on system call type.
File manipulation
Network
Others
(4) Get arguments
...
(5) Branch based on manipulation.
open
read/write, close
(6) Essential file?
No
Yes
(7) FD is already registered to FD list?
No
Yes
(8) Proxy execution.
(9) Get the result of (8).
(10) Success?
No
Yes
(11) Branch based on manipulation.
open
read/write
close
(12) Register FD to FD list.
(13) Remove FD from FD list.
(14) Set the starting address of system call routine to the instruction pointer.
(15) Set the result to a register and buffers
(16) Set the starting address of the last part of the system call routine to the instruction pointer.
(17) Return to the protection target VM.

Figure 3 The flow of VMM to control system calls and requesting proxy execution.

12. The VMM registers the FD of the opened file to the FD list held by the VMM. Then, the VMM moves to Step (15).

13. The VMM removes the FD of the closed file from the FD list held by the VMM. Then, the VMM moves to Step (15).

14. The VMM sets the starting address of the system call routine to the instruction pointer. This is needed to restart the original system call routine as if no interposition occurred in the protection target VM.

15. The VMM sets the starting address of the last part of the system call routine to the instruction pointer. This operation enables skipping of the original system call routine. This skip is key for hiding file manipulation from attackers. Because the original system call routine is bypassed with this step, file access monitoring in the kernel is also bypassed. For example, with hooks for `ptrace` located in the original system call routine, malicious processes cannot detect invocation of proxy execution and invoked system calls interposed by the proposed method.

16. Finally, the VMM returns the processing, and the protection target VM resumes.

After the proxy execution, the proxy process must return the result of proxy execution. To return the result from the proxy process to the essential process, we added a new hypercall to the Xen hypervisor. Hypercall is an interface to request processing from a VM to a VMM. The proxy process returns the result of the proxy execution by the new hypercall. The VMM receives the hypercall and manipulates the register of the protection target VM to set the return value of the original system call. Finally, the VMM manipulates the instruction pointer of the protection target VM to skip original system calls.

Table 1 Required information for proxy execution.

Name	Description
VMID	VMID is required to identify the VM
Address of PD	Address of PD is used to identify the essential process
Current directory	Current directory is used to construct the full path of the target file when a relative path is passed by the system calls
System call number	System call number is used to determine whether the system call is related to file manipulation or not
Arguments	Arguments of the system call are required to request the proxy execution of the system call. If the arguments include the address of the buffer, the VMM copies the buffer between the VM and the VMM

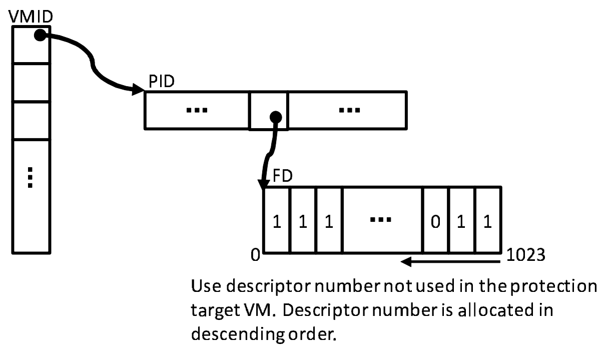
VMID
PID
FD
1 1 1 ... 0 1 1
0
1023
Use descriptor number not used in the protection target VM. Descriptor number is allocated in descending order.

Figure 4 A table for managing file descriptors of essential files.

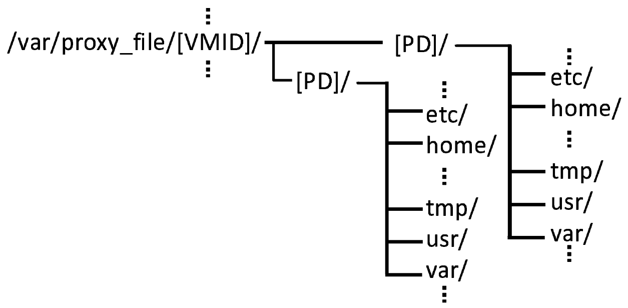

Figure 5 Directory structure in Proxy VM.

5.4 Managing of Essential Files

In the proposed method, the essential processes manipulates normal files and essential files. To use the existing programs as an essential service with no modification, the proposed method must transparently provide normal files and essential files to the essential processes. To address this requirement, we changed the allocation rule of file descriptors (FD). It is ideal to split the table for FD, however this approach requires modifications to the OS of the protection target VM. Therefore, the VMM allocates FDs for essential files in descending order. Figure 4 shows the table layout for managing FDs in the VMM. FD numbers for normal files are allocated by the OS in ascending order (0, 1, 2, . . .). Conversely, the VMM allocates the FD numbers for essential files in descending order (1023, 1022, 1021, . . .). If the descriptor number for a normal file is within the range of the numbers allocated to essential files, the VMM aborts the process. Thus, normal files and essential files are distinguished with no modification to the essential service or the OS. However, we note that the maximum number for the FD is reduced.

5.5 Proxy Execution of System Call

In the proposed approach, a proxy process is allocated to each protection target VM. A proxy execution must address the following two points: the information to be passed to the proxy process and the structure of the directory tree. Table 1 lists the information required for proxy execution. The virtual machine ID (VMID) and the address of page directory (PD) are required to classify the VM and the process. Because the address of PD is unique to the process, and the value is already saved in VMM when VM exit occurred, we can classify the process without copying its PID. By classifying the process by PD, we can reduce the overhead for copying PID from the protection target VM. The current path of the essential process is required to construct the full path of the essential files. The system call number is required to execute the system call, and the arguments are simply passed to the system call. Only the path information is changed by the proxy process, as the structure of the directory tree differs from that of the protection target VM.

In the proxy VM, essential files are stored as shown in Figure 5. All essential files are stored under the `/var/proxy_file/` in proxy VM. To distinguish files of the proxy VM and file of the protection target VMs, proxy processes create directories, which are named by VMID for each VM. In the same manner, a proxy process creates directories for each essential process. Directories for essential processes are distinguished by the address of page directory (PD).

The proxy process executes the system call requested from the VMM. Figure 6 shows the flow of proxy execution by a proxy process.

1. The proxy process listens to the event channel for proxy execution. When the VMM requests proxy execution, an event is delivered to the channel. The proxy process starts the proxy execution by receiving the event.

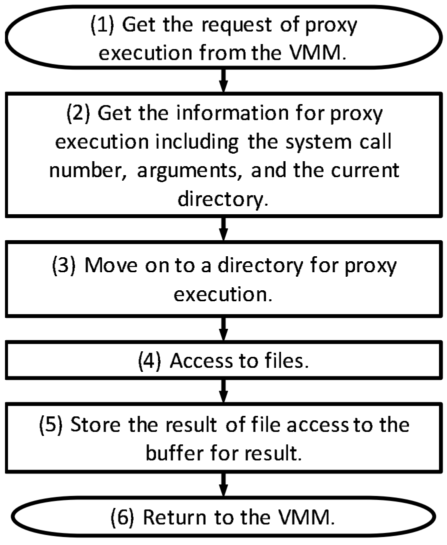
(1) Get the request of proxy execution from the VMM.
(2) Get the information for proxy execution including the system call number, arguments, and the current directory.
(3) Move on to a directory for proxy execution.
(4) Access to files.
(5) Store the result of file access to the buffer for result.
(6) Return to the VMM.

Figure 6 The flow of proxy execution by a proxy process.

2. The proxy process gets the information for proxy execution by invoking a hypercall to the VMM. The information is detailed in Table 1.

3. The proxy process moves on to a directory for proxy execution. To manage multiple protection target VMs and essential processes, the proposed method allocates directories for each VM and process. Thus, the proxy process first moves to the corresponding directory.

4. The proxy process accesses the requested file.

5. The proxy process sets the results to a data structure for proxy execution. The data structure is copied by the VMM after processing is returned to the VMM. The return value and affected buffers are stored in this area.

6. The proxy process returns processing to the VMM. Then, the VMM start resuming the phase of the original system call.

The VMM classifies whether the file is an essential file or not, so it must hold the list of essential files. However, it is difficult to determine which file is an essential file because files are manipulated dynamically. We addressed this problem using a policy. At first, the VMM holds the list of essential files. The list includes files which may be classified as static essential files. Then, any file newly created by an essential process is handled as an essential file because the file probably includes information identifying the process as an essential process. Finally, a file overwritten by the essential process is also handled as an essential file using the same logic applied to newly created files.

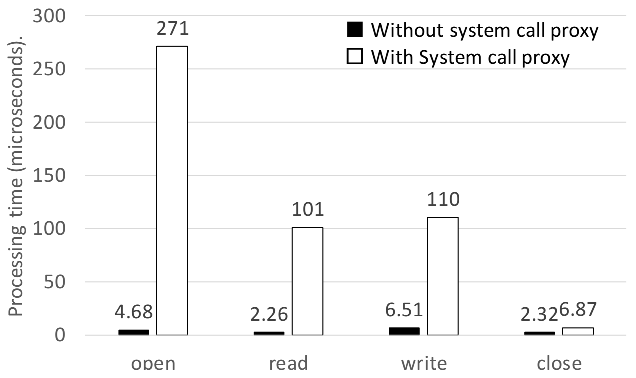

Figure 7 Performance overhead by system call proxy in open, read, write, and close system calls.

6 Evaluation

6.1 Purpose and environment

To evaluate the performance of the proposed method, we measured the processing time for system calls related to file manipulation. Additionally, we also measured the performance of the security software: ClamAV. Although ClamAV has various functionalities for security, we used `clamscan` command for evaluation. The computer used had an Intel Core i7-2600 (3.4 GHz, 4 cores) and 16 GB RAM. The protection target VM had one virtual CPU (VCPU) and 1 GB RAM. The proxy VM comprised three VCPUs and 15 GB RAM. The VCPUs of VMs were pinned to separate physical CPUs.

6.2 Basic Performance

The evaluation results for basic performance are provided in Figure 7. We compared the performance of open, read, write, and close system calls to a VMM without interposition to those made to a VMM with the system call proxy. In this evaluation, to clarify the overhead of system calls that involved a system call proxy, the essential process always accessed the essential files. The size of buffer for both read and write was 10 bytes.

The performance degradation in open, read, and write was greater than that of close. We observed substantial degradation in the performance of open. We suggest that the reason for the degradation was memory copy. Read and write have buffers to read or write. These buffers are copied from the protection target VM to the proxy VM, and this copy causes large overhead. The performance of open system call was substantially degraded because of directory creation. The proxy process creates directories for each protection target VM and its essential processes. Thus, the first access to the essential file by an essential process requires the creation of a directory for the essential process in the proxy VM. Also, to check whether the directory already exists, additional system calls are invoked by the proxy process. These additional system calls lead to the large overhead. Thus, preserving the state of

Table 2 Processing time for clamscan in seconds.

w/o System Call Proxy	w/ System Call Proxy	
	No Essential File	One Essential File
12.24	12.25	12.31

directory creation effectively improved the performance of proxy execution. By preserving the states, it was possible to reduce the number of the additional system calls by the proxy process. Additionally, reducing the number of buffer copies between the VMM and VMs also reduced the overhead. These performance improvements will make up our future work.

6.3 ClamAV

The proposed method primarily focuses on the protection of security software. Therefore, this section describes performance evaluation of the proposed method with ClamAV, which is a famous antivirus software. Because the implementation of the proposed method is currently focusing on Linux, we evaluated the performance of clamscan, which is a command within ClamAV that detects malicious files by comparing files with signatures on Linux. To evaluate the performance of the proposed method, we measured the performance of clamscan with an essential file (the signature file) and one hundred normal files (scan target). The size of the signature file was 113 MB and of the size each normal file was 4 KB. All normal files were stored in a directory, and we measured the time for scanning the above directory.

Table 2 shows the result of scanning time of clamscan. By comparing the time with and without the proposed method, we can observe that performance overhead with the proposed method is less than 1%. The results also show that the performance changes if the manipulated files include essential files. However, performance degradation with an essential file is small in this measurement. This is due to the workflow of clamscan. Clamscan first loads the signature and then scans target files. Because performance will degrade only for loading the signature, overall performance was not significantly degraded.

7 Conclusion

We have described the design, implementation, and evaluation of a hiding method for the file manipulation of essential services. The proposed method exploits virtual machine monitor (VMM) for interposition and proxy execution. Interposed system calls by essential services on the protection target VM are executed by another VM to hide files and file manipulation from attackers. Because essential files are stored in the other VM and inaccessible from normal processes on the protection target VM, it is difficult for attackers exploiting normal processes to identify essential services by monitoring files and file manipulation. Therefore, the proposed method is more resistant to attack than a mechanism inside the VM. In addition, the proposed method is designed to use a VMM and another VM without modification to software on the protection target VM. With a design requiring no modification to the protection target VM, conventional security software is safe from attacks. The evaluation results show that the performance of system calls related to file manipulation is largely degraded. However, performance evaluation of clamscan showed that the performance degradation is less than 1% in a standard case.

Acknowledgement

This work was partially supported by KAKENHI Grant Numbers JP18K18051 and JP16H02829.

References

Min, B., Varadharajan, V., Tupakula, U. and Hitchens, M. (2014) 'Antivirus security: naked during updates', *Softw. Pract. Exp.*, Vol. 44, No. 10, pp.1201–1222.

Gregio, A., Afonso, V., Filho, D., Geus, P. and Jino, M. (2015) 'Toward a taxonomy of malware behaviors', *Comput. J.*, Vol. 58, No. 10, pp.2758–2777.

Hsu, F.H., Wu, M.H., Tso, C.K., Hsu, C.H. and Chen, C.W. (2012) 'Antivirus software shield against antivirus terminators', *IEEE Trans. Inf. Forensics Secur.*, Vol. 7, No. 5, pp.1439–1447.

Garfinkel, T. and Rosenblum, M. (2003) 'A virtual machine introspection based architecture for intrusion detection', in *Network and Distributed Systems Security Symposium*, Vol. 3, pp.191–206.

Srinivasan, D., Wang, Z., Jiang, X. and Xu, D. (2011) 'Porcess out-grafting: an efficient "out-of-VM" approach for fine-grained process execution monitoring', in *Proceedings of 18th ACM Conference on Computer and Communications Security*, pp.363–374.

Sato, M., Yamauchi, T. and Taniguchi, H. (2015) 'Process hiding by virtual machine monitor for attack avoidance', *J. Inf. Process.*, Vol. 23, No. 5, pp.673–682.

Dinaburg, A., Royal, P., Sharif, M. and Lee, W. (2008) 'Ether: malware analysis via hardware virtualization extensions', in *Proceedings of 15th ACM Conference on Computer and Communications Security*, pp.51–62.

Hegarty, R. and Haggerty, J. (2015) 'Extrusion detection of illegal files in cloud-based systems', *IJSSC*, Vol. 5, No. 3, pp.150–158

Wang, J., Yu, M., Li, B., Qi, Z. and Guan, H. (2012) 'Hypervisor-based protection of sensitive files in a compromised system', in *Proceedings of 27th Annual ACM Symposium on Applied Computing*, pp.1765–1770.

Almeshekah M.H. and Spafford, E.H. (2014) 'Planning and integrating deception into computer security defenses', in *Proceedings of 2014 Workshop on New Security Paradigms Workshop*, pp.127–138.

Araujo, F., Hamlen, K.W., Biedermann, S. and Katzenbeisser, S. (2014) 'From patches to honey-patches: lightweight attacker misdirection, deception, and disinformation', in *Proceedings of 21st ACM Conference on Computer and Communications Security*, pp.942–953.

Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I. and Warfield, A. (2003) 'Xen and the art of virtualization', *SIGOPS Oper. Syst. Rev.*, Vol. 37, No. 5, pp.164–177.