

(Short Paper) Evidence Collection and Preservation System with Virtual Machine Monitoring

Toru Nakamura^{1,2}[0000–0003–2530–648X], Hiroshi Ito³, Shinsaku Kiyomoto¹, and
Toshihiro Yamauchi^{3,2}[0000–0001–6226–5715]

¹ KDDI Research, Inc., 2-1-15 Ohara, Fujimino-shi, Saitama, 356-8502, Japan.

² Advanced Telecommunications Research Institute International,
2-2-2 Hikaridai, Seika-cho, Soraku-gun, Kyoto, 619-0288, Japan.

³ Graduate School of Natural Science and Technology, Okayama University,
3-1-1 Tsushima-naka, Kita-ku, Okayama, 700-8530, Japan.

Abstract. In a system audit and verification, it is important to securely collect and preserve evidence of execution environments, execution processes, and program execution results. Evidence-based verification of program processes ensures their authenticity; for example, the processes include no altered/infected program library. This paper proposes a solution for collection of evidence on program libraries based on Virtual Machine Monitor (VMM). The solution can solve semantic gap by obtaining library file path names. This paper also shows a way to obtain hash values of library files from a guest OS. Furthermore, this paper provides examples of evidence on program execution and the overhead of the solution.

Keywords: Virtual machine introspection · Forensics · OS security

1 Introduction

1.1 Background

Audit and verification based on log files as digital evidence of program execution environments, execution processes, and execution results are important for trustworthiness. In particular, a secure collection and preservation mechanism for the evidence should be implemented on the system in order to ensure the validity of the evidence. There are two types of logs for verifying execution; logs of application programs (APs) such as a web server program and “system level” logs such as those of operating systems. Sufficient logs for verification might not be always collected due to the design of the APs. Thus, collection of system level logs of execution processes is required for amending evidence as well.

1.2 Attack Model

Our target is a system based on virtual machine (VM). The threat considered in this paper is defined as executing illegal processing while concealing the wrongdoing. Our assumed adversary is a user of a guest OS, who has neither an administrator account of the guest OS nor any account of the host OS, while we assume that an administrator of host OS is trusted. One example of an attack scenario is to run a program with malicious and different library files from those usually used by employing an illegal and malicious method. For example, it is possible for a user by modifying LD_PRELOAD. In this case, we subsequently need to obtain the path names of the library files and the hash values for proof that such attack has occurred. From existing studies, it is not clear if this is possible.

1.3 Related Work

Linux Audit [1] is a mechanism for collecting system level logs in the Linux system. In Linux Audit, a collection mechanism in kernel space collects information about events occurring on a system based on a preliminary set of rules, and an audit daemon process in the user space stores the information in log files [2, 3]. Pfoh et al. [4] proposed a hardware-based system call tracing function for VMs. The function on the virtual hardware of the guest VM obtains information from the guest VM using Virtual Machine Monitor (VMM) and monitors system calls. Yan et al. [5] presented a method for recording events on the VM and reproducing in order to analyze malware behaviors. The method records the initial state and processes of state changes in an environment, reproducing them in another environment. The method includes both hardware virtualization support technology and software emulation technology. In addition, the initial state is recorded and changing states are processed by an instruction unit of the CPU and a program function unit. Hassan et al. [6] proposed a method of generating provenance graphs based on logs by APs and an operating system. The graphs are useful for analyzing malicious behaviors perpetrated by an attacker.

1.4 Our Contribution

The goal of our study is to realize a system, called an Evidence Collection and Preservation System (ECoPS), for collecting and preserving evidence without any restrictions of information on collecting evidence for program execution. In this paper, we focus on collection of evidence on program libraries as a first step of our study. From the viewpoint of security, it is more desirable to obtain the evidence from a VMM than from a guest OS. To obtain evidence from a VMM the issue of semantic gap needs to be solved. We show a solution to this issue by obtaining the library file path names. We also show how to obtain the hash values of library files from a guest OS. We evaluate the solution by performing an implementation, and provide examples of evidence on program execution for proof of concept. We also evaluate the overhead of system calls incurred by the solution.

2 Issues on Existing Evidence Collection Systems

Pfoh *et al.* [4] proposed a method whereby information on system call can be obtained via Virtual Machine Introspection (VMI) [7]. VMI is a monitoring mechanism for the guest OS that works by obtaining a memory statement on virtual hardware through VMM. In their scheme, the information related to the guest OS is not stored in the guest OS, but in the host OS, and no adversary can gain influence over VMM at all from the guest OS. Their scheme suffers from the semantic gap issue [8]; VMM does not interpret data on memory in the virtual hardware of the guest OS due to the lack of information on data structure in the guest OS. Pfoh *et al.* considered a solution only on standard inputs/outputs and command line arguments; it does not include the path name of executable files, or executed library files because it is not for auditing.

In this paper, we propose an effective method to fill semantic gaps not only on standard inputs/outputs and command line arguments, but also on path name of executed files, environment variables, and OS type and its version for collecting evidence for auditing. Furthermore, our method calculates hash values in guest OS, after that they are stored to host OS, in order to collect hash values of executable files, executed library files, and accessed files.

3 Evidence Collection and Preservation System (ECoPS)

3.1 Requirements

The requirements for ECoPS are shown as follows.

System requirement 1: *prevents alteration of evidence.*

System requirement 2: *prevents attacks from infiltrating monitoring functions.*

System requirement 3: *collects appropriate information related to program execution.*

For system requirements 1 and 2, we adopt a VMM based architecture for ECoPS. Regarding system requirement 3, we need to identify the type of program execution for extracting the list of information related to program execution for use as evidence. We show the assumed program process flow in Fig. 1. We assume that a program is executed in a shell. The user first inputs a path name and arguments of the execution file to a shell. A shell process creates a duplicate process of itself via a `fork` system call. A child process issues an `execve` system call, loads the specified program, and initializes the state of the process. Next, it runs the program process. In addition, program libraries are mapped to a virtual memory space before the process begin in the case of a dynamically linked program. In this paper, we regard program processes as processes with a file, standard input and output, and standard error output.

For the assumed program processes previously shown, an evidence collection system should obtain the following information.

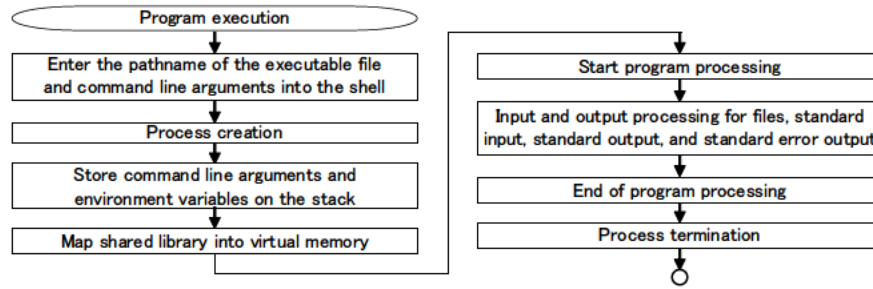


Fig. 1. Program process flow

- Information to identify the executable files
 1. Path name of an executable file
 2. Hash value of an executable file
- Information to identify the executed library files
 3. Path names of executed library files
 4. Hash values of executed library files
- Information to identify the user’s inputs and outputs
 5. Contents of standard input, standard output and standard error output
- Information to identify used files
 6. Path names of used files
 7. Hash values of used files
- Information to identify the execution environment
 8. Command line arguments
 9. Environment variables
 10. OS type and its version

3.2 Design Criteria for ECoPS

An overview of ECoPS is shown in Fig. 2. In ECoPS, we assume that the monitored OS is a guest OS on the VM. We adopt a Kernel-based Virtual Machine (KVM) that supports full virtualization as the VMM because if using full virtualization VMM, the source codes of the guest OS do not need to be modified. Note that we assume that the VMM is secure and the administrator of the VMM does not perform any illegal actions.

In ECoPS, both the monitoring function in the guest OS and that in the KVM of the host OS obtain information by executing a hooking system call from a user process in the guest OS. The reason for using two different monitoring functions is due to the issue of semantic gap. It is comparatively easy to obtain OS information from the monitoring function in the guest OS. However, the monitoring function is not so tolerant. The processes involved in obtaining information from the KVM can also obtain OS information by referring virtual hardware such as the memory of the guest OS directly. The monitoring function

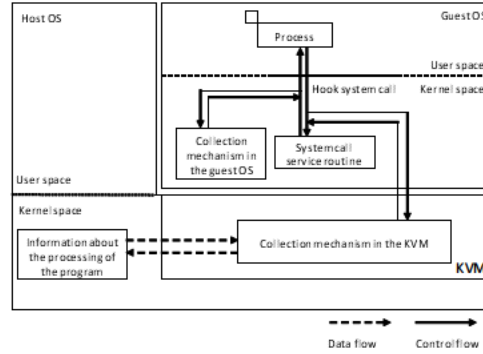


Fig. 2. Overview of ECoPS

in the KVM is tolerant to attacks against the OS because they are isolated from the guest OS. However, the data structure of the guest OS needs to be analyzed to solve the semantic gap. Therefore, to minimize the security risk, we decided that the only information for which it is difficult to solve the semantic gap should be obtained from guest OS.

The monitoring function in the guest OS is realized by implementing a kernel module. The monitoring function in the KVM obtains information by executing a hooking system call and tracing variables on the memory in the guest OS and members of structures. We use a system call hook method described in [9] for the hooking system call in the guest OS with the KVM. This method uses a hardware breakpoint. ECoPS can obtain information about the guest OS by setting a hardware breakpoint and executing a hooking system call. In the case of x86-64 architecture, we use hardware debug registers for setting the breakpoint. ECoPS can set the breakpoint to an instruction which is run just before the system call process by setting the address of the instruction into the hardware debug register. If the instruction, which exists in the breakpoint address, is executed, a debug exception occurs. Therefore, a system call in the guest OS can be hooked because the KVM can capture the occurrence of VMexit by the debug exception.

The following design criteria were considered in the design of ECoPS.

- Criterion 1:** Specific methods to obtain information on a system call with taking semantic gap into consideration
- Criterion 2:** Specific methods for securely transferring information obtained from the guest OS to the KVM
- Criterion 3:** Specific methods for securely storing the information in the KVM

In the later sections, we focus on only the Criterion 1 and regard the Criteria 2 and 3 as outside the scope. We provide the specific method for obtaining the information in the case of identifying executed libraries as an example in the next section. In particular, we present a method for obtaining the path name and the hash values of libraries.

4 Evidence Collection Mechanism

In this section, we describe the method for obtaining the information on executed library files. The following information are required for auditing;

- (a) Library file path name
- (b) Hash values of library files

In many Linux distributions, the mechanism of dynamic link is widely used because it contributes to reducing the amount of memory. In using dynamic links, linked libraries are mapped on a virtual memory space for program execution. Storing the path names of linked libraries is not sufficient to regard them as evidence of program execution because we cannot determine whether the linked libraries have replaced with illegal ones. Therefore, we need not only the path names of library files, but also the hash values of the files. `openat` and `mmap` are system calls for dynamically linked libraries. The `openat` system call opens a library file that is dynamically linked. The `mmap` system call maps the library file on a virtual memory space. We adopt the use of the `openat` system call for hooking because this system call is used earlier than `mmap`. It helps to reduce the possibility of statements on memory being altered by adversaries. We show methods for identifying the path names and the hash values of libraries. We assume that both the guest OS and host OS are Linux 4.15.18 and a KVM is used as the virtualization environment.

4.1 Library File Path Names

We use the `dentry` structure for obtaining the library file path names because although the arguments of the `openat` system call include the file path names, they may be a relative path or symbolic links by which it is difficult to identify the absolute path. The `dentry` structure is a data structure holding file names, directory names and the address of the parent `dentry` structure in virtual file systems. The relationships among structures related to file path names are shown in Fig. 3. To obtain a path name of the target library, we need to refer to the `d_name` in the `dentry` structure. Though the `d_name` stores a file name or directory name, we need to trace the parents' `dentry` structures recursively in order to obtain the absolute path of the target.

First, we explain the method used to identify a `dentry` structure of the target library. We assume that a CPU based on the x86-64 architecture is used. We obtain the contents of IA32 GS BASE register, which is an MSR (Model Specific Register) in the architecture. MSR is a group of registers for CPU control. The register stores the initial addresses of CPU variables, including the `current_task` variable. The `current_task` variable stores the initial address of the `task_struct` structure which stores the `files` variable. The `files` variable stores the initial address of the `files_struct` structure which stores the `fdt` variable. The `fdt` variable stores the initial address of the `fdtable` structure which stores the `fd` variable. The `fd` variable stores the initial address of

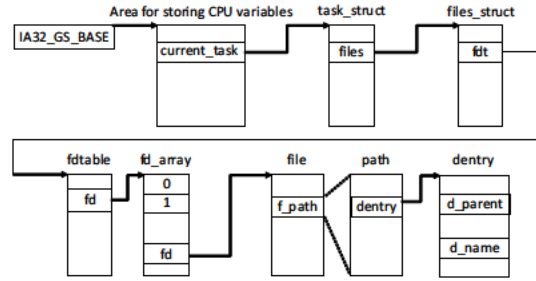


Fig. 3. Relations of structures from MSR to **dentry** structure

the **fd_array**. The **fd_array** stores the initial addresses of the **file** structures which stores the **f_path** variable. In **openat** system call, the file descriptor is received as the return value. The file descriptor is the index of the **fd_array** for the target library. Therefore, we identify the initial address of the **file** structure of the target library by the file descriptor. The **f_path** variable is the **path** structure, which stores the initial **dentry** address. We identify the **dentry** structure of the target from the address.

Next, the method used to obtain the absolute path of a target library will be explained. By the above method, we identify the **dentry** structure of the target library. The **dentry** structure stores the **d_name** variable and **d_parent** variable. The **d_parent** variable stores the address of the **dentry** structure of the parent directory. We repeatedly refer to the **dentry** structure of the parent and obtain the directory name until the root directory is reached. Finally, we construct the absolute path of the target library from the file name and directory names.

4.2 Hash Values of Library Files

In this paper, we show a way to obtain the hash values of libraries not in KVM, but in the guest OS. As mentioned in Section 3, it is desirable to obtain the hash values in KVM if this is possible. However, we need to append additional routines to analyze the file system of the guest OS to solve the semantic gap. We must implement different routines for each file system. This is not practical because it causes an increase in the cost for the modification as well as an increased risk of vulnerabilities. Therefore, we install a module into the guest OS to obtain the hash values of libraries.

The module is installed into a guest Linux kernel as a Linux Kernel Modules (LKM). It also hooks into **openat** system calls to obtain the hash value of a library file specified with the argument of **openat**. It uses Integrity Measurement Architecture (hereinafter called IMA) [10], which is one of the integrity verification frameworks that are adopted in the Linux kernel, to calculate the hash values of libraries. IMA has the **ima_calc_file_hash** function to calculate file hash values. We can also use Linux Kernel Crypto API [11] instead of IMA.

```
[ 1527.161376] i_ino = 136731
[ 1527.161377] full_path = /lib/x86_64 linux gnu/libc-2.27.so
```

Fig. 4. Logs of processes of getting a path-name of a file

4.3 Consideration of ways to collect other information

We discuss mechanisms to obtain other information as follows;

Standard inputs, standard outputs, and standard error outputs. These are obtained by hooking `write/read` system call and referring to the argument of the system call storing the address of the buffer storing the contents. In a similar way, we may also be able to obtain evidence on command line arguments and environment variables with `execve` system call.

Path names of files and OS types. These are obtained using the same approach described in Section 4.1, to trace `dentry` structures. For the hash values of executable files and executed files, these may be obtainable using the way described in Section 4.2, to calculate hash values.

Path names of executable files. Though not difficult to obtain from a guest OS, it is desirable to obtain these from KVM for reasons of security. A candidate of system call for obtaining path names of executable files is `execve` system call. However, the returned variable is different from `openat`; hence we cannot directly apply the concept proposed in this paper. This is still an open issue.

5 Evaluation

The evaluation environment consists of Intel(R) Core(TM) i5-6500 @ 3.20 GHz, 4 GB memory for the Guest OS, 32 GB memory for the Host OS, and Ubuntu 18.04 LTS (Linux 4.15.18, 64bit) for both the Guest OS and Host OS. In the implementations, we assigned one vCPU to a monitored guest OS. We show a part of the kernel logs in the host OS related to the process of obtaining the path name of a library in Fig. 4. We can find that an `inode` number of a target library and the corresponding path name of the target library. Fig. 5 shows the kernel logs in the guest OS related to the processes of obtaining the hash value of a library. We used SHA-256 as the hash algorithm. We can find that an `inode` number of a target library, the corresponding path name, and the corresponding hash value of the target library. We used the Linux `sha256sum` command for validating the operation, and confirmed that the hash value obtained from our implementation matched the hash value with the Linux `sha256sum`.

We evaluated the overhead of system calls incurred by the collection mechanism for path names. We measured the number of clock cycles for repeating `openat` system calls and `getpid` system calls 100 times, and repeated this trial 10 times. Table 1 shows the result. From this result, the overhead for `openat` system call was 121,880 clock cycles per system call and that in another system call such as `getpid` was 67,235 clock cycles. The substance between these overheads of 54,645 clock cycles is a result of obtaining the path names in the `openat`

```
[ 1562.289421] filename = /lib/x86_64 linux gnu/libc 2.27.so
[ 1562.305725] i_ino = 136731
[ 1527.161377] digest = f0ad9639b2530741046e06c96270b25da2339b6c15a7ae46de8fb021b3c4f529
```

Fig. 5. Logs of processes of getting a hash value of a file**Table 1.** Result of evaluating `openat` and `getpid` system calls (clock)

	openat				getpid			
	Max.	Min.	Ave.	Med.	Max.	Min.	Ave.	Med.
Normal	310,976	3,688	9,756	9,313	3,632	1,312	2,348	2,504
Ours	13,409,152	66,596	131,636	67,438	9,445,120	34,324	69,583	34,528

system call. If it is assumed that a 3.0 GHz processor is used, the overhead of system calls without `openat` is about 22.4 μ s. The overhead of an applications is increased in proportion to the number of system calls which the application issues. Therefore, the influence on performance from the overhead caused by the collection mechanism is acceptable unless the application issues system calls too frequently. Note that the overhead is not included the calculations of hash values, hence it remains as future work.

Table 2 shows a comparison with our perspective of ECoPS and other existing techniques. Note that it is just our perspective as described in Section 4.3, we have not implemented and evaluated all collection functions. While existing techniques did not provide any way to collect the path names and hash values of executed libraries. In our perspective, ECoPS is capable of collecting all the information required for evidence of program executions as shown in Section 3.

6 Conclusion

In this paper, we showed a solution to solve the semantic gap in obtaining the library file path names as a first step of realizing ECoPS. We also showed a way to obtain hash values of library files from a guest OS. We evaluated the solution by performing an implementation, and provided examples of evidence on program execution for proof of concept. We also evaluated the overhead of system calls incurred by the solution and showed that the overhead can be acceptable. In future work, we will try to find a way of obtaining the path names of executable files in VMM. Though the Criteria 2 and 3 shown in Section 3 are regarded as outside the scope, we will discuss and attempt to realize a method capable of securely transferring evidence from the guest OS to KVM and a method to securely store evidence in KVM.

Acknowledgement. This work was partially supported by JSPS KAKENHI Grant Numbers 19H04109 and 19H05579.

Table 2. Comparison with our perspective of ECoPS and existing work

Collected evidence	ECoPS (this paper)	Linux Audit [1]	Pfoh <i>et al.</i> [4]
Path names of executable files	✓ (Guest OS)	✓	
Hash values of executable files	✓ (Guest OS)		
Path names of executed libraries	✓		
Hash values of executed libraries	✓ (Guest OS)		
Standard I/O and error output	✓	✓	✓
Path names of used files	✓		
Hash values of used files	✓ (Guest OS)		
Command line arguments	✓	✓	✓
Environment variables	✓	✓	
OS Type and its version	✓		

References

1. Linux Audit, available from (<https://people.redhat.com/sgrubb/audit/>) (accessed 2020-12-02).
2. Latzo, T. and Freiling, F.: Characterizing the Limitations of Forensic Event Reconstruction Based on Log Files, *Proc. 2019 18th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/13th IEEE International Conference On Big Data Science And Engineering (Trust-Com/BigDataSE)*, pp.466–475 (2019).
3. Ma, S., Zhai, J., Kwon, Y., et al.: Kernel-Supported Cost-Effective Audit Logging for Causality Tracking, *Proc. 2018 USENIX Annual Technical Conference (USENIX ATC’18)*, pp.241–253 (2018).
4. Pfoh, J., Schneider, C. and Eckert, C: Nitro: Hardware-Based System Call Tracing for Virtual Machines, *Proc. 6th International conference on Advances in Information and Computer Security*, pp.96–112 (2011).
5. Yan, L. K., Jayachandra, M., Zhang, M. and Yin, H.: V2E: combining hardware virtualization and software emulation for transparent and extensible malware analysis, *Proc. 8th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE’12)*, pp.227–237, 2012.
6. Wajih Ul Hassan, Mohammad A. Nouredine, Pubali Datta, Adam Bates, Omega-Log: High-Fidelity Attack Investigation via Transparent Multi-layer Log Analysis, *Proceedings of the Network and Distributed System Security Symposium (NDSS 2020)*, pp.1-16, 2020.
7. Garfinkel, T. and Rosenblum, M.: A Virtual Machine Introspection Based Architecture for Intrusion Detection, *Proc. Network and Distributed Systems Security Symposium (NDSS 2003)*, (2003).
8. Chen, P. M. and Noble, B. D.: When Virtual Is Better Than Real, *Proc. 8th Workshop on Hot Topics in Operating Systems*, pp.133–138 (2001).
9. Fujii, S., Sato, M., Yamauchi, T. and Taniguchi, H.: Evaluation and Design of Function for Tracing Diffusion of Classified Information for File Operations with KVM, *The Journal of Supercomputing*, Vol.72, No.5, pp.1841–1861 (2016).
10. Linux Integrity Subsystem, available from (<http://linux-ima.sourceforge.net/>) (accessed 2020-12-04).
11. Linux Kernel Crypto API, available from (<https://www.kernel.org/doc/html/v4.15/crypto/index.html>) (accessed 2020-12-04).