ISSN: 2182-2069 / E-ISSN: 2182-2077

Ensuring Log Authenticity in System Audits with VMM-Based Evidence Collection

Toru Nakamura[1*], Hiroshi Ito[2], Jeuk Kang[3], Takamasa Isohara[4], and Toshihiro Yamauchi[5]

[1*]Associate Professor, Department of Computer Science, Tokyo City University, Japan KDDI Research, Inc., Japan. tonakamu@tcu.ac.jp, https://orcid.org/0000-0003-2530-648X

[2]Graduate School of Natural Science and Technology, Okayama University, Japan. ito-hiroshi@s.okayama-u.ac.jp, https://orcid.org/0009-0006-7381-295X

[3]School of Engineering, Okayama University, Japan. p6b6863u@s.okayama-u.ac.jp, https://orcid.org/0009-0007-7063-6716

[4]Group Leader, KDDI Research, Inc., Japan. ta-isohara@kddi.com, https://orcid.org/0000-0003-2371-3698

[5]Professor, Faculty of Environmental, Life, Natural Science and Technology, Okayama University, Japan. yamauchi@okayama-u.ac.jp, https://orcid.org/0000-0001-6226-5715

Received: December 02, 2024; Revised: January 14, 2025; Accepted: January 27, 2025; Published: February 28, 2025

Abstract

In system audits and verifications, it is critical to collect logs and preserve them as evidence of execution environments, processes, and program results. To effectively use logs as evidence, the authenticity of the logs, files, and other information related to program execution must be ensured. Unlike previous systems that modify guest-operating systems, this paper introduces a new system that utilizes a virtual machine (VM) monitor (VMM) to preserve evidence of program execution, extending the scope of evidence collection beyond standard system-call arguments. The proposed system uses VMM to isolate the acquired information and its acquisition mechanism from the monitored guest-operating system. This paper presents a solution to the semantic gap issue for information that could be required as evidence. Moreover, it provides examples of program-execution evidence and demonstrates that it can detect specific attacks. Furthermore, we report the results of measuring the performance overhead of benchmark programs and the kernel-based VM compilation processing. The findings demonstrate that the proposed evidence collection and preservation system effectively detects sophisticated attacks, such as CVE-2021-4034, enhances system security through tamper-resistant evidence collection, and maintains practicality with minimal overhead for applications with infrequent system calls.

Keywords: Evidence Collection, Program Execution, Virtual Machine Introspection, Virtual Machine Monitor.

1 Introduction

Cyber-physical systems (CPS) collect data from the physical world, analyze them using digital technologies in the cyber world, convert them into useful information or knowledge, and feed them back

Journal of Interner Services and Information Security (JISIS), volume: 15, number: 1 (February), pp. 288-304
DOI: *10.58346/JISIS.2025.I1.018*

*Corresponding author: Associate Professor, Department of Computer Science, Tokyo City University, Japan KDDI Research, Inc., Japan.

to the physical side to create added value (Arvinth, 2023). These systems have attracted considerable attention in recent years. Because the damage caused by cyber-attacks can also affect physical systems, security measures have become increasingly important (Kassim, 2017). As multiple institutions collaborate, if one institution generates unreliable data tampered with by an attack, it can undermine trust in the entire system. To realize a reliable CPS, ensuring the authenticity of logs related to cyberspace processes and protecting the evidence backed by these logs is essential.

In this study, we emphasized the safeguarding of evidence related to program execution. Merely acquiring command logs from program execution does not provide sufficient proof that processing is conducted correctly. For instance, if the dynamically linked program libraries were illicitly tampered, there is a risk that data generated by inappropriate processing could circulate as valid data (Diwakar & Roy, 2024). Therefore, the evidence should include not only the command logs from program execution, but also the library files used during execution and the execution environment at that time. Moreover, to ensure the authenticity of logs, a mechanism is necessary to prevent the deletion or tampering of logs and evidence, even for those with administrative permissions.

This study focused on systems using virtual machines (VMs) (Poisel et al., 2013). In recent years, it has become commonplace for clients to set up virtual servers (guest-operating systems (OSs)) on cloud servers provided by cloud service providers (Aravind et al., 2023) (host OS) and deploy services. This study assumes that the host OS is trustworthy; it would not execute unauthorized operations due to attacks, nor would malicious insiders delete or tamper with logs or evidence (Fakiha, 2024) We hypothesized that the guest operating system could be vulnerable to unauthorized actions due to security breaches or malicious users erasing or modifying logs or evidence.

Linux auditing is common in Linux-based environments for system-wide log gathering. In a Linux audit system, a kernel-level collection mechanism gathers information about system events based on predefined rules, whereas a user-space audit daemon process saves this information in log files (Latzo & Freiling, 2019; Ma et al., 2018). However, if an attacker gains administrative privileges, it can alter the stored evidence or compromise the audit daemon.

In virtualized environments, employing a VM monitor (VMM) may prevent attackers with administrative access within a guest OS from tampering with logs or evidence. This is achieved using the VMM to monitor processes in the guest OS and store logs on the host OS (Garfinkel & Rosenblum, 2003).

However, these schemes encounter a semantic gap problem (Chen & Noble, 2001): the VMM is unable to interpret memory data within the virtual hardware of the guest OS because it lacks information about the guest OS's data structures. Pfoh et al., (2011) proposed a methodology to resolve the semantic gap between standard inputs/outputs and command-line arguments. However, because their method was not intended to leave evidence indicating appropriate processing, its capability to obtain sufficient information for use remains unclear.

Our Contribution

The contributions of this study are as follows:

- We developed a new evidence collection and preservation system based on a kernel-based VM (KVM). The proposed system is equipped with a mechanism to collect data on processes and execution environments during program execution with low overhead during actual service operations (Gholamzadeh, 2019). This mechanism is realized not within the guest OS, but by obtaining information from the VMM. Thus, even if an attacker hijacks the guest OS, an abnormal

program-execution history can be collected using the proposed method on the VMM side and preserved as evidence. This allows for the analysis of abnormal processes based on evidence. The proposed system facilitates the comprehensive collection of information, such as the identification of executed programs and libraries, environmental variables that affect program execution, and processes requested by application programs to the OS (system calls). However, collecting all histories would result in enormous overhead; therefore, the system is designed to collect evidence that is limited to the information necessary for post-execution verification. The implementation of the proposed system requires the resolution of the semantic gap. Although the method used to resolve the semantic gap is based on existing techniques, its application for evidence preservation offers a novel aspect.

- We assessed the proposed solution by implementing it and presenting examples of program-execution evidence as proof of concept. In addition, we demonstrated its ability to detect a specific attack based on CVE-2021-4034 (Qualys Security Advisory, 2022). Furthermore, we evaluated the system-call overhead introduced by the proposed system and showed that it remained acceptably low for applications that do not frequently issue system calls.

2 Proposed System

In this section, we present a new evidence collection and preservation system based on a KVM. The system collects evidence of the program execution and securely stores it.

2.1. System Requirements

The requirements of the proposed system are as follows:

1. The system should prevent alteration of evidence.
2. The system should prevent attacks from infiltrating monitoring functions.
3. The system should gather relevant information pertaining to program execution.

To satisfy system requirements 1 and 2, we adopted VMM-based architecture for the proposed system. For system requirement 3, it is necessary to determine the type of program execution to compile a list of information related to program execution for evidentiary purposes.

2.2. Collected Information

Figure 1 illustrates the process flow of the proposed program. It is assumed that the program ran within the shell. Initially, the user inputs the path name and arguments of the executable file into the shell.

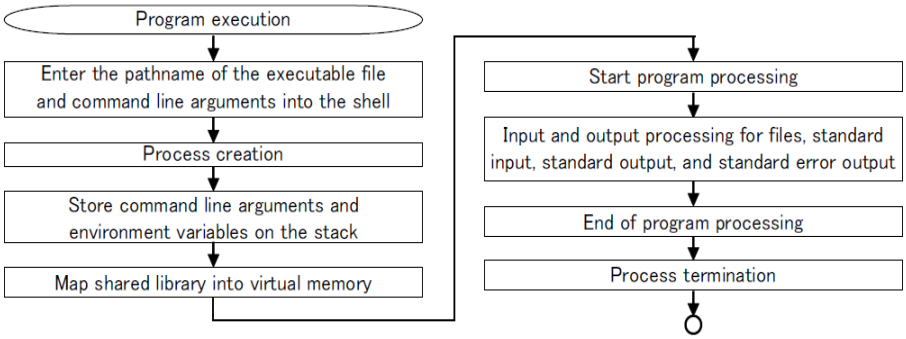

Figure 1: Program Process Flow

The shell process then uses a clone system call to create duplicate processes. The child process runs an executive system call, which loads the specified program and sets up the process state.

Subsequently, the program is executed. If the program is dynamically linked, the program libraries are mapped to virtual memory before the process starts. In this study, we considered programming processes that include a file, standard input and output, and standard error output.

Considering the overhead for real-time collection, we limited our focus to the following information shown in Table 1:

Table 1: Information to Focus on in this Paper

To identify executable files	Absolute pathname of the executable file
	Hash value of the executable file
To identify executed library files	Absolute pathnames of the executed library files
	Hash values of the executed library files
To identify the user's inputs and outputs	Standard input
	standard output
	Standard error output
To identify used files	Absolute pathnames of used files
	Hash values of used files
To identify execution environment	Command-line arguments
	Environmental variables
	OS type and version

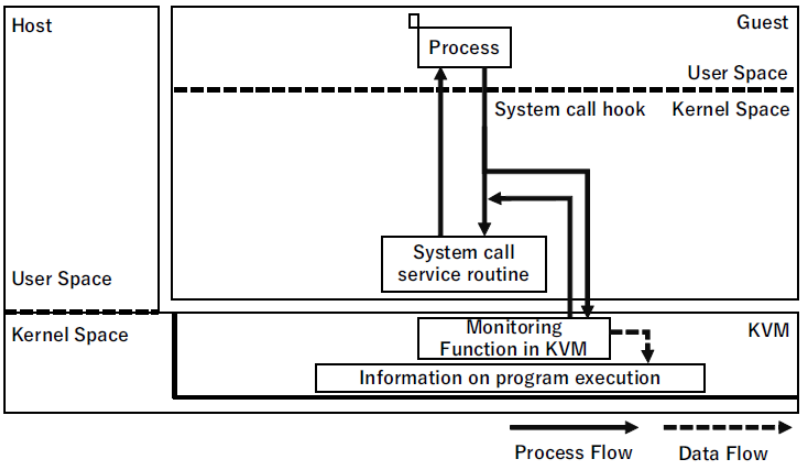

Figure 2: Overview of the Proposed System

2.3. Design Direction

Figure 2 illustrates the proposed system. In this setup, the monitored operating system operates as a guest OS within a VM. We chose KVM, which supports full virtualization, as the VMM. This selection enabled us to avoid altering the source code of the guest OS. Additionally, we assumed that the VMM is secure and that its administrator refrains from illicit activities.

We assume that the proposed system hooks system user processes as system calls in the guest OS using the KVM monitoring mechanism to obtain evidence, and hash values in the guest OS are obtained from the Linux submodules. The information required as evidence is retrieved from the value of a

register obtained by hooking a system call, and the memory in the guest OS where this information is stored is identified and read. This method is safe for the attack model assumed in this study owing to the reliability of the VMM user privileges. In particular, the KVM monitoring mechanism is more robust against attacks because it is isolated from the guest OS. Additionally, when using the KVM monitoring mechanism, the semantic gap problem can be solved independently of the OS version, type, and CPU architecture if information on the data structure is provided in advance.

3 Evidence Collection Mechanism

Our target is Linux on the x86-64 architecture of the KVM. This section presents secure acquisition opportunities and methods for each type of information listed in the previous section.

To address the semantic gap issue, our approach leverages existing techniques to interpret data structures within the guest OS, ensuring accurate and reliable evidence collection. By integrating these techniques with our VMM-based monitoring system, we bridged the gap between low-level data and high-level semantic information, thereby enhancing the clarity and effectiveness of the collected evidence.

3.1. Executable Files

In Linux systems, executing programs involve specifying an executable file and making an executive or executive at system call. Identification of an executable file is possible through its absolute pathname and hash value.

We showcase a technique for determining the absolute pathname of an executable file by examining the contents of the guest operating system's memory.

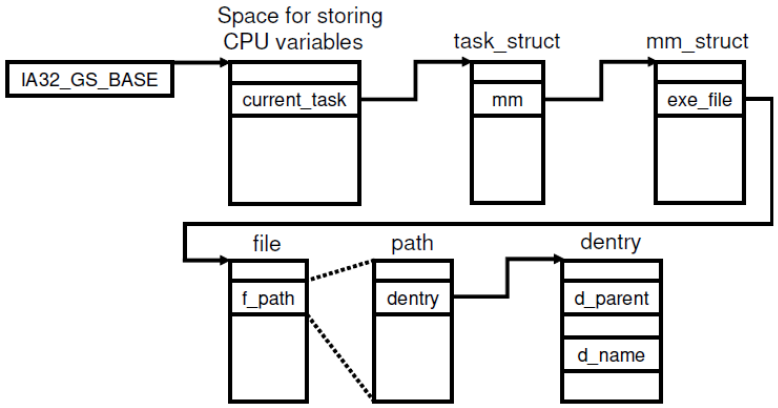

Figure 3: Relationships of Data Structures for Absolute Pathname of an Executable File

Figure 3 illustrates connections between structures involved in obtaining the absolute pathname of the executable file. By utilizing KVM to trace each component, the dentry structure associated with the executable file can be located. In the x86-64 architecture, this process involves accessing the IA32_GS_BASE register, which stores the top address of the CPU variable storage space, including the current_task variable. As depicted in Figure 3, following the pointer in the IA32_GS_BASE register, the top address of the dentry structure was identified.

We then outline the method for acquiring the executable file's hash value by extracting the hash value generated by IMA/EVM, a Linux subsystem, from memory. IMA/EVM functions as a security module within the Linux Kernel Integrity subsystem.

IMA/EVM was introduced to prevent the intentional modification of files and store hash values, including those of executables, libraries, and files used in the file system.

An overview of this process is shown in Figure 4. The series of processes in the IMA/EVM module are (1) calculation of the hash value of the file, (2) comparison with the registered hash value, and (3) return of the comparison result, which is executed when the open/exec/read system-call routine occurs. Hash values can be obtained by hooking a process that compares hash values in a series of processes in the IMA/EVM module. More precisely, immediately after the hash value comparison process and before the return instruction is executed, the address of the hash value can be checked from the register by hooking it using the debugging register and forcing VMexit to occur.

3.2. Executed Library Files

In Linux, a dynamic link mechanism was used to reduce the amount of memory required. The Openat system call is issued earlier than the mmap system call when a library file with a dynamic link is opened. Therefore, it is desirable to obtain such information when processing Openat system calls. The information used to identify a library file included its absolute pathname and hash value.

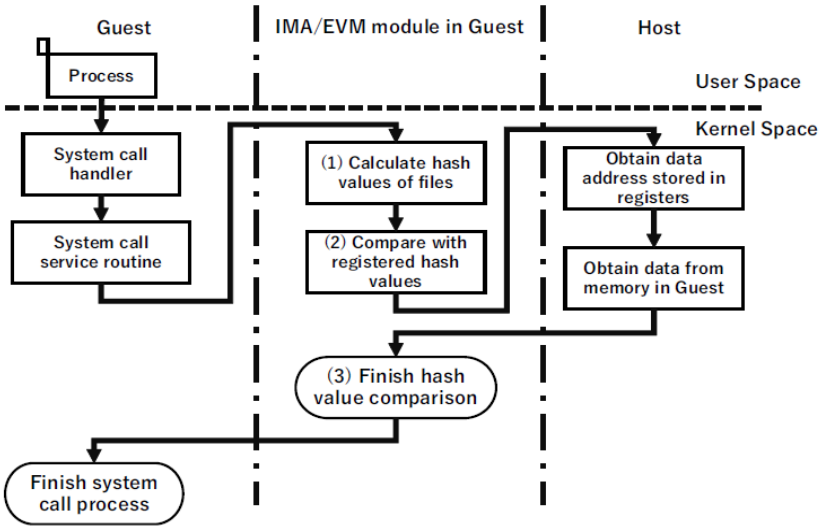

Figure 4: Process Flow of the Transfer File Hash

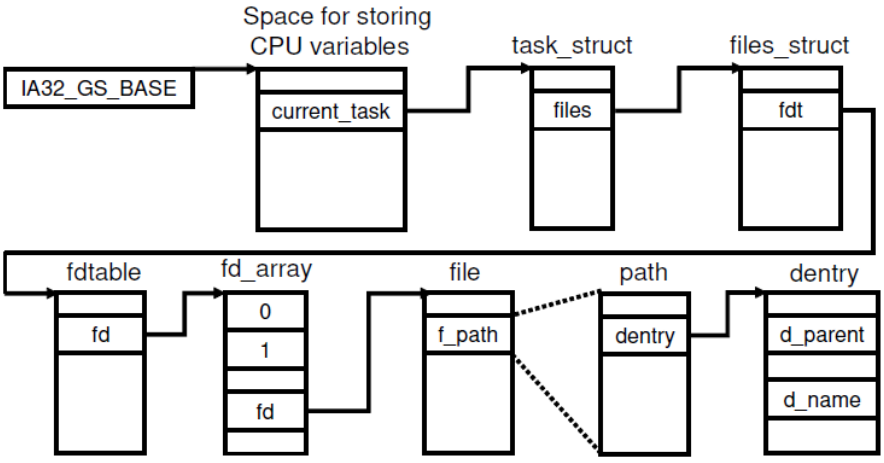

Figure 5: Relationships of Data Structures Related to Obtaining an Absolute Pathname of a File

First, we describe the retrieval of the absolute pathname of the library file. Figure 5 illustrates the relationships between the structures used for this purpose. The file and directory names can be obtained by referencing the d_name member of the entry structure. In addition, absolute pathnames can be derived from the file and directory names by tracing the parent dentry structure back to the root directory.

The hash value of the library file is obtained in KVM in the same manner as described in Section 3.1.

3.3. Inputs and Outputs

The read/write instructions to standard input, standard output, and standard error output are executed by assigning a special value to the file descriptor (fd) argument and issuing a read/write-related system call. Therefore, it is possible to determine whether the target process is related to a read/write instruction by checking the value of fd specified in the argument when a system call is issued. In Linux, the contents of the standard input, standard output, and standard error output are included in up to the sixth argument of the system call. Because up to the sixth argument is passed using registers, the contents stored in the guest's memory can be retrieved from the KVM based on the address of the buffer stored in the register.

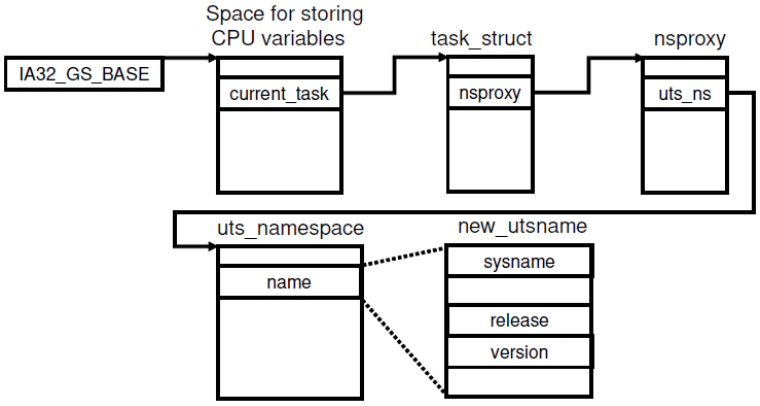

Figure 6: Relationships of Data Structures Related to Obtaining the OS Type and Version

3.4. Files used in File Operations

The information used to identify an accessed file includes its absolute path name and hash value. The absolute pathname of the file should be obtained during the initial stage of the file operation. Therefore, information should be gathered from the open, Openat, name_to_handle_at, and open_by_handle system calls, which are issued when a file is opened. The absolute pathnames and hash values of the files used were obtained in the KVM in the same manner as described in Section 3.1.

3.5. Execution Environment

The information needed to identify the program-execution environment consists of the command-line arguments, environmental variables at the time of program execution, and the OS type and version. This information should be obtained at the time of processing these system calls because they are the arguments of the execve and execveat system calls.

Command-line arguments and environmental variables can be obtained during program execution, as described in Section 3.3. The OS type and version were obtained from the KVM by tracing the variables and structures shown in Figure 6 and identifying the top address of the new_utsname structure.

Table 2: Environment for Evaluation

CPU		Intel(R) Core (TM) i5-10500K @ 3.10 GHz
OS	Guest Host	Ubuntu 20.04 LTS (Linux 5.4.159, 64 bit) Ubuntu 20.04 LTS (Linux 5.4.159, 64 bit)
Memory	Guest Host	4 GB 16 GB

Table 3: Information Required as Evidence

Item	Evidence		System call
	Purpose	Information type	
1	Identify executable file	Absolute pathname of the executable file	execve, execveat
		Hash value of the executable file	
2	Identify executed library file	Absolute pathnames of the executed library files	Openat
		Hash values of the executed library files	
3	Identify the user's inputs and outputs	Standard input	read, pread, readv, preadv, preadv2
		Standard output	write, pwrite, writev, pwritev, pwritev2
		Standard error output	
4	Identify used files	Absolute pathnames of the used files	open, open_by_handle, Openat, name_to_handle_at
		Hash values of the used files	open, Openat, name_to_handle_at, open_by_handle, write, pwrite, writev, pwritev, pwritev2
5	Identify the execution environment	Command-line arguments	execve, execveat
		Environmental variables	
		OS type and version	

4 Evaluation

In this section, we demonstrate the efficacy and practicality of the proposed system through the following three evaluations:

(1) An evaluation that confirms evidence for program execution can be obtained.
(2) A case study demonstrating the vulnerabilities exploited during program execution can be detected.
(3) Measurement of the overhead incurred by the proposed system.

The evaluation environment is presented in Table 2. One vCPUs is allocated to the guest OS.

4.1. Evidence for Program Execution

In this section, we demonstrate that the proposed scheme can correctly obtain all the required items listed in Table 3 by outputting the information to the kernel log of the host OS. The standard error output in Item 3 is omitted because it can be obtained in the same manner as the standard output.

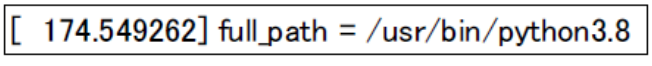

Figure 7: Logs of Processes for Obtaining the Absolute Pathname of an Executable File

```
ito@ito-Standard-PC-Q35-ICH9-2009:~$ getfattr -e hex -m security -d testopen
# file: testopen

security.evm=0x0302028540c906008083b85a96e220bd5f49594c11b27869f099067
9715d60abcd816bf49b4cb70fab4ee7d336500e8471d093e04802532b7ba5877feb3
9b00d0b49b1ee975a7a39c36cf1beac0b2cd9b7d11cbf0bb3df9dcfae39d538b39c4a
5e2cba2f70af8af871ca6000115f8a65023cb426251377e748ff19e69817e7d8f683e6f
baf492ad0c1

security.ima=0x01be74e5e5cc5a886d4d3e1cfbdc51e60af15035be
```

Figure 8: File Hash Checked in Guest OS

```
[ 3121.611008] [kang]
security.ima=0x01be74e5e5cc5a886d4d3e1cfbdc51e60af15035be
```

Figure 9: File Hash Checked by the Log of the Host OS

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

Figure 10: Logs of Processes of Obtaining the Pathname of a Library File

The absolute pathname of the file in Item 4 was omitted because it was obtained in the same manner as in Item 2. The hash values of the files in Items 2 and 4 were also omitted because they were obtained in the same manner as in Item 1.

Figure 7 shows the kernel log of the host OS for obtaining the absolute pathname of the executable file in Item 1.

Next, we present two logs that include the hash values of the executed file. Figure 8 shows the hash value registered to the guest OS with IMA/EVM, and Figure 9 shows the hash value used by the host OS to refer to the guest OS when running the file. These figures show that the hash values are identical. This indicates that the host OS can obtain hash values that are registered with the guest OS. Furthermore, it contains information that identifies the process of issuing system calls and files.

Figure 10 shows the kernel log of the host OS for obtaining the absolute pathname of the library file in Item 2.

Figure 11 shows the kernel log of the host OS for obtaining the standard input and output for Item 3 when using the cat command. The cat command outputs the standard input content to the output. Figure 11 shows that the contents of the standard input and output can be obtained.

```
[ 553.640119] stdin data: stdio test
[ 553.640127] stdout data: stdio test
```

Figure 11: Logs of Processes of Obtaining Standard Input and Standard Output Data

```
[ 143.941025] sysname = Linux
[ 143.941027] release = 5.4.159
[ 143.941036] version = #1 SMP Mon Nov 15 14:42:53 JST 2021
[ 423.882693] arg[0]: ls
[ 423.882706] arg[1]: --color=auto
[ 423.882711] arg[2]: -la
[ 423.882730] env[0]: SHELL=/bin/bash
                      :
                      :
                      :
[ 423.885023] env[49]: _=/usr/bin/ls
```

Figure 12: Logs of Processes of Obtaining Information to Identify the Program-execution Environment

Figure 12 shows the kernel logs of the host OS for obtaining the command-line arguments, environmental variables during program execution, and OS type and version in Item 5. This log is obtained when the ls command is used with the "l" and "a" options specified as command-line arguments. Figure 12 shows that the OS type and version information were obtained from the parts indicated as sysname and release, respectively. In addition, the part indicated as a version contains the date on which the kernel was built. Furthermore, the command-line argument log confirms that the options specified as aliases and those entered into the shell are obtained. Finally, 50 environmental variables were obtained from the environmental variable log.

While the current examples provide a foundational understanding of the capabilities of the proposed system, future work will focus on expanding the range of scenarios and providing more detailed case studies to further demonstrate the robustness and versatility of the system.

4.2. Detection of Attacks that Exploit Vulnerabilities in Program Execution

In this section, we confirm the effectiveness of the proposed system by demonstrating that it can detect attacks that exploit CVE-2021-4034, which has been reported to be vulnerable to program execution (Qualys Security Advisory, 2022).

Attack by CVE-2021-4034 (Qualys Security Advisory, 2022)

CVE-2021-4034 is a vulnerability of the pkexec program in polkit. Polkit is an authorization framework created to allow unprivileged programs to request privileged action. pkexec is a program that allows a non-privileged user to execute commands specified in command-line arguments as another user, such as root, according to a predefined policy. The vulnerability in CVE-2021-4034 is caused by an error where a string set as an environment variable is recognized as a command-line argument when no command-line argument is passed to pkexec. For example, if this vulnerability is exploited by rewriting the environment variable that can be set as arguments to execve, malicious code can be executed. Because pkexec is a Set User ID program that uses root privileges, it can execute arbitrary code with root privileges.

```
[ 2448.452877] env[0]: pwnkit.so:.
[ 2448.452882] env[1]: PATH=GCONV_PATH=.
[ 2448.452889] env[2]: SHELL=/lol/i/do/not/exists
[ 2448.452894] env[3]: CHARSET=PWNKIT
[ 2448.452897] env[4]: GIO_USE_VFS=
```

Figure 13: Logs of Environmental Variables at the Time of the Attack

A known attack exploiting CVE-2021-4034 involves including malicious code in an output library when making pkexec output error messages. We describe the details of a specific malicious code that uses CVE-2021-4034 vulnerability to invoke a shell with root privileges. First, the environmental variables are set in the env structure as follows, and the execve system call is issued.

- env[0]: pwnkit.so:.
- env[1]: PATH=GCONV_PATH
- env[2]: SHELL=/lol/i/do/mpt/exists
- env[3]: CHARSET=PWNKIT

Next, the pkexec command was executed with the environmental variables specified in env.

Because of the CVE-2021-4034 vulnerability, if no command-line arguments are set, the string on the first line of the environment variable is recognized as the command-line argument in pkexec. For the third line of the SHELL environment variable, an attacker generates an error by setting a path that does not exist, because if the absolute pathname of the shell set in this environment variable is not described in the /etc/shells file, an error occurs in pkexec. When an error occurs, an error message is displayed based on the CHARSET environment variable specified in Line 4. In pkexec, if the CHARSET environment variable is not UTF-8, the iconv_open() function in glibc is called to display the error message in the corresponding format. The iconvict_open() function was executed using the output library, which was determined using the configuration file gconv-modules. In this example, because PWNKIT is specified instead of UTF-8 as the CHARSET environment variable, it is processed using the output library. The directory in which this library is stored is specified by the GCONVERSION_PATH environmental variable. In this example, the PATH environment variable is specified as “GCONV_PATH,” hence the environment variable in the first line is replaced by “GCONV_PATH=./pwnkit.so:..” Thus, if the pwnkit.so:. library contains malicious code; it would be executed.

Detection Test

Figure 13 shows the kernel log of the host OS containing the environmental variables when the attack (Berardi, 2022) described in the previous section is executed. The possibility of detecting an attack from the log is evident because it contains the five environmental variables required for the attack.

Figure 14 shows the host OS kernel logs for the files used during program execution.

```
[20522.981941] full_path = /etc/shells
...
[20522.983623] full_path = /home/ito/ito/workspace/CVE-2021-4034-main/gconv-modules
...
[20522.984837] full_path = /home/ito/ito/workspace/CVE-2021-4034-main/pwnkit.so
```

Figure 14: Logs of Files Used During the Attack

The log contains the absolute pathnames of the three files used for the attack. The first line indicates that the file containing the absolute pathname of the available login shells has been opened. The second line indicates that the configuration file gconv-modules, which determines the library to output error messages, was opened. The third line indicates that the library for outputting error messages has been opened. Because the current directory path is set in the GCONV_PATH environment variable, the use of a library containing malicious code in the current directory could be confirmed.

4.3. Measurement of the Overhead

We measured the system-call overhead of the proposed system to evaluate its practicality. First, we investigated the system-call overhead using LMBench 3.0, which is a microbenchmark. We also evaluated applications (APs) to determine whether the overhead is acceptable for practical use. Here, we focus on the frequency of system-call usage. We evaluated Redis, an in-memory database, as an AP with a high frequency of system-call usage and evaluated the processing time of compiling KVM as an AP with a low frequency of system calls.

Overheads of the System Calls

The overhead of the proposed system-call hooks can be divided into two major categories: overhead of VMexit and overhead of memory search processing to obtain evidence, as shown in Section 3. First, the overhead of the getppid was measured using LMBench to separate the overhead of VMexit from that of memory search. This is because the getppid system call has no overhead due to memory searching but includes the overhead of VMexit. Next, the system-call overheads for read, write, open+close, and fork+execve were measured using LMBench. The proposed system hooked read, write, open, and execve for evidence collection. The evaluation results are presented in Table 4.

From the results of the getppid, the overhead due to VMexit was estimated to be 2.64 μs. The overhead of VMexit is the same for all system calls; however, the overhead of memory search varies between system calls.

The read results reveal that the overall overhead is 3.41 μs, of which the overhead due to memory searching was estimated to be 0.77 μs because the overhead due to VMexit was 2.64 μs.

Table 4: System-call Processing Time and Overhead (μs)

	Original (T1)	**Ours (T2)**	**T2-T1**	**T2/T1**
getppid	0.05	2.69	2.64	54.9
read	0.13	3.54	3.41	27.23
write	0.08	3.05	2.97	37.65
open+close	0.91	7.36	6.46	8.13
fork+execve	69.99	88.31	18.32	1.26

Table 5: Redis Benchmark Results (Average Processing Time for Each Request (μs))

Request	**Original (T1)**	**Ours (T2)**	**T2-T1**	**T2/T1**
set	0.869	6.234	5.366	7.175
get	0.872	6.120	5.248	7.017

Similarly, for write operations, the overall overhead is 2.97 μs, of which the overhead due to memory searching was estimated to be 0.33 μs. For open+close and fork+execve, VMexit is executed for two system calls. The overall overhead from open+close was 6.46 μs, of which the overhead from memory searching for open was estimated to be 1.17 μs. The overall overhead from fork+execve was 18.32 μs, of which the overhead from memory searching for execve was estimated to be 13.04 μs.

These results show that the system-call hooks require an overhead of several microseconds. Furthermore, the processing time of memory search by the proposed system varies for individual system calls.

Redis Benchmark

Assuming the typical use of an AP that frequently issues system calls, we measured processing time using the Redis benchmark. Redis is an in-memory database used in Web backend environments. The Redis benchmark measures the number of times a set or get instructions can be executed per unit time. When a set or receive instruction is executed, a read or write system call is called once. The results are presented in Table 5.

Table 5 shows the overhead of approximately 5.3 μs per request. As shown in Section 5.3.1, the overhead for a single system call is small, that is, approximately 3 μs. However, for applications where a small percentage of the processing time is in user mode, the impact of the system-call hook overhead is relatively large.

KVM Compile

Assuming that the user of an AP does not frequently issue system calls, we measured the processing time of a KVM compiler. Here, we measured the total compile time (real), which includes the time executed in the user mode (user) and the time executed in the kernel mode (sys). The results are presented in Table 6.

The user time of the proposed system remains nearly similar to that of the original system, whereas the sys time of our system increases by approximately 1.5 s. Hence, the total real time of our system is almost the same as that of the original system, increasing by only 5%.

Table 6: KVM Compile Time (s)

Type	Original (T1)	Ours (T2)	T2-T1	T2/T1
real	26.015	27.423	1.408	1.054
user	23.702	23.279	−0.423	0.982
sys	1.775	3.274	1.499	1.845

4.4. Discussion

We evaluated the trade-offs between security and system performance to provide a comprehensive analysis of performance overhead. Our results indicate that while the proposed system introduces some overheads owing to VMexit and memory search processing, this is a necessary trade-off to ensure robust security and reliable evidence collection. Specifically, the overhead is more pronounced in applications with a high system-call frequency, such as Redis, where the average processing time per request increased by approximately 5.3 μs. However, for applications with lower system-call frequencies, such as KVM compilation, the overall impact on performance was minimal, with only a 5% increase in total compile time. These results underscore the need to balance security measures with performance demands. Future studies will aim to enhance the system and reduce the overhead while maintaining rigorous security protocols.

4.5. Challenges and Limitations

Although the suggested system shows promise in safeguarding evidence of program execution, several potential obstacles exist. These include the semantic gap problem, increased overhead due to VMexit during system calls, and reliance on the host OS's integrity. Future research should focus on tackling these constraints to improve the resilience and real-world applicability of the system.”

5 Related Work

A technique for gathering information on a guest VM by monitoring system calls using a VMM was introduced by Pfoh et al., (2011). This approach is considered secure, owing to its independent operation from the guest OS. Su et al., (2021) developed an out-of-VM VMI method for analyzing malware, utilizing a VMM to monitor a VM in an isolated environment and employing cache memory to examine the control flow. However, these methods are limited in their ability to obtain only system-call arguments. In contrast, the proposed system can acquire system-call arguments and additional information crucial for evidence collection, such as complete file paths, environmental variables, and hash values.

Yan et al., (2012) introduced a method for recording and replaying events on a virtual machine to analyze malware. This approach integrates hardware virtualization with software-emulation techniques. It requires more detailed information than the proposed system, including the program's initial state and state change process. The proposed system focuses on information crucial for result verification and is expected to reduce the runtime overhead.

Hassan et al., (2020) developed a technique to create a history graph using application program (AP) and operating system (OS) logs. This graph examines attacker behavior using AP-generated log information and system-level data. The proposed system emphasizes gathering information related to AP execution, such as system calls and environmental variables, when the AP requests OS processes, aiming to preserve runtime evidence. Kwon et al., (2021) devised a method for reconstructing the attack execution processes for forensic analysis. This approach records resource accesses based on system-call processing and reconstructs them from these records. The proposed system collects information to identify elements involved in program execution, aiming to verify program-execution processes.

The proposed system offers distinct benefits compared with current solutions. Traditional log collection methods, operating within the guest operating system, are vulnerable to manipulation by attackers with administrative rights. In contrast, the proposed system uses VMM-based architecture to segregate and safeguard evidence-based gathering processes, thereby improving the security and reliability of the collected data. The system addresses the semantic gap challenge by utilizing existing techniques to decipher data structures within the guest operating system, ensuring precise evidence collection.

However, the proposed system has some limitations. It incurs performance overheads owing to VMexit and memory search operations, which may impact applications that frequently use system calls. The system's efficacy relies on the trustworthiness of the host operating system to prevent unauthorized actions. Future research will aim to streamline the system to reduce overhead and investigate techniques to enhance the security and dependability of evidence gathering.

6 Conclusion

Our team developed a method to safeguard program execution evidence using a VMM. This approach separates the collected data and the collection process from the guest operating system under observation. We addressed the semantic gaps for all the necessary evidentiary information. In our framework, the acquisition process is initiated when system calls are associated with the generation of information that needs to be preserved as evidence is executed. Furthermore, we demonstrated a method for obtaining the hash values of files related to program execution from a guest OS.

We assessed the solution by implementing it and presenting examples of program-execution evidence as proof of concept. Additionally, we evaluated our system with an attack that exploits vulnerability-

enabling environmental variables to be tampered with during the program runtime. The results showed the possibility to obtain evidence regarding the vulnerability exploited during program execution. Furthermore, the evaluation using LMBench showed that the overhead is significant when many VMexits occur during the information acquisition. We also demonstrated that the overhead was higher for applications that frequently issue system calls. Finally, we demonstrated that the overhead is acceptably small for applications that frequently do not issue system calls. The findings demonstrate that the proposed evidence collection and preservation system effectively detects sophisticated attacks, such as CVE-2021-4034, enhances system security through tamper-resistant evidence collection, and maintains practicality with minimal overhead for applications with infrequent system calls. Future work will focus on optimizing the system to reduce overhead and explore methods to further enhance the security and reliability of the evidence collection process.

Acknowledgments

This work was partially supported by JSPS KAKENHI Grant Numbers 23K24848, and The Okayama Foundation for Science and Technology.

References

[1] Aravind, B., Harikrishnan, S., Santhosh, G., Vijay, J. E., & Saran Suaji, T. (2023). An Efficient Privacy - Aware Authentication Framework for Mobile Cloud Computing. *International Academic Journal of Innovative Research*, *10*(1), 1–7. https://doi.org/10.9756/IAJIR/V10I1/IAJIR1001

[2] Arvinth, N. (2023). Digital Transformation and Innovation Management: A Study of How Firms Balance Exploration and Exploitation. *Global Perspectives in Management*, *1*(1), 66-77.

[3] Berardi, D. (2022). berdav/CVE-2021-4034: CVE-2021-4034 1day. GitHub. https://github.com/berdav/CVE-2021-4034

[4] Chen, P. M., & Noble, B. D. (2001, May). When virtual is better than real [operating system relocation to virtual machines]. In *Proceedings eighth workshop on hot topics in operating systems* (pp. 133-138). IEEE. https://doi.org/10.1109/HOTOS.2001.990073

[5] Diwakar, & Roy, J. (2024). The Role of Data Analytics in Digital Transformation: A Study of how Firms Leverage Data for Insights. *Indian Journal of Information Sources and Services*, *14*(4), 29–34. https://doi.org/10.51983/ijiss-2024.14.4.05

[6] Fakiha, B. (2024). Unlocking Digital Evidence: Recent Challenges and Strategies in Mobile Device Forensic Analysis. *Journal of Internet Services and Information Security, 14*(2), 68-84. https://doi.org/10.58346/JISIS.2024.I2.005

[7] Garfinkel, T., & Rosenblum, M. (2003, February). A virtual machine introspection based architecture for intrusion detection. In *Ndss* (Vol. 3, No. 2003, pp. 191-206).

[8] Gholamzadeh, A. (2019). The service - oriented architecture and service storage of establishing safe or secure communication among island systems. *International Academic Journal of Economics*, *6*(1), 80–90. https://doi.org/10.9756/IAJE/V6I1/1910006

[9] Hassan, W. U., Noureddine, M. A., Datta, P., & Bates, A. (2020, January). OmegaLog: High-fidelity attack investigation via transparent multi-layer log analysis. In *Network and distributed system security symposium*.

[10] Kassim, N. M. (2017). Effect of perceived security and perceived privacy towards trust and the influence on internet banking usage among Malaysians. *International Academic Journal of Social Sciences*, *4*(2), 26-36.

[11] Kwon, Y., Wang, W., Jung, J., Lee, K. H., & Perdisci, R. (2021, February). C2sr: Cybercrime scene reconstruction for post-mortem forensic analysis. In *Network and Distributed Systems Security (NDSS) Symposium 2021*.

[12] Latzo, T., & Freiling, F. (2019, August). Characterizing the limitations of forensic event reconstruction based on log files. In *2019 18th IEEE International Conference on Trust, Security and Privacy in Computing and Communications/13th IEEE International Conference on Big Data Science and Engineering (TrustCom/BigDataSE)* (pp. 466-475). IEEE. https://doi.org/10.1109/TrustCom/BigDataSE.2019.00069

[13] Ma, S., Zhai, J., Kwon, Y., Lee, K. H., Zhang, X., Ciocarlie, G., ... & Jha, S. (2018). {Kernel-Supported}{Cost-Effective} Audit Logging for Causality Tracking. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)* (pp. 241-254).

[14] Pfoh, J., Schneider, C., & Eckert, C. (2011, November). Nitro: Hardware-based system call tracing for virtual machines. In *international workshop on security* (pp. 96-112). Berlin, Heidelberg: Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-25141-2_7

[15] Poisel, R., Malzer, E., & Tjoa, S. (2013). Evidence and Cloud Computing: The Virtual Machine Introspection Approach. *Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications, 4*(1), 135-152.

[16] Qualys Security Advisory. (2022). pwnkit: Local privilege escalation in polkit's pkexec (CVE-2021-4034). Qualys. https://www.qualys.com/2022/01/25/cve-2021-4034/pwnkit.txt

[17] Su, C., Ding, X., & Zeng, Q. (2021). Catch you with cache: Out-of-VM introspection to trace malicious executions. In *Proceedings of the 2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)* (pp. 326-337). https://doi.org/10.1109/DSN48987.2021.00045

[18] Yan, L. K., Jayachandra, M., Zhang, M., & Yin, H. (2012, March). V2e: combining hardware virtualization and software emulation for transparent and extensible malware analysis. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments* (pp. 227-238). https://doi.org/10.1145/2151024.2151053

Authors Biography

Toru Nakamura received the B.E., M.E., and Ph.D. degrees from Kyushu University, in 2006, 2008, and 2011, respectively. In 2011, he joined KDDI and in the same year he moved to KDDI R & D Laboratories, Inc. (currently renamed KDDI Research Inc.). From 2018 to 2020, he worked as a researcher at the Advanced Telecommunications Research Institute International (ATR). From 2020 to 2024, he worked as a researcher at KDDI Research Inc. Since 2024, he has been serving as an Associate Professor with Tokyo City University. He has received the CSS2016SPT Best Paper Award. His current research interests include security and privacy, particularly privacy-enhanced technology and trust computing. He is a member of the IEICE and IPSJ.

Hiroshi Ito received his B.E. and M.E. degrees from Okayama University, Japan in 2020, 2022, respectively. His research interests include computer security.

Jeuk Kang received his B.E. degree from Okayama University, Japan in 2023. His research interests include computer security.

Takamasa Isohara received his B.E. and M.E. degree in Information and Computer Science from Keio University, Japan, in 2005 and 2007, respectively. He joined KDDI and has been engaged in research on network security, smartphone security, SIM-based IoT security, and automotive cybersecurity. He is currently a senior manager at the Usable Trust Laboratory of KDDI Research Inc. He received the IPSJ Kiyasu Special Industrial Achievement Award in 2011. He is a member of IEICE and IPSJ.

Toshihiro Yamauchi received the B.E., M.E., and Ph.D. degrees in computer science from Kyushu University, Japan, in 1998, 2000, and 2002, respectively. In 2002, he became a Research Associate with the Faculty of Information Science and Electrical Engineering, Kyushu University. In 2005, he became an Associate Professor at the Graduate School of Natural Science and Technology, Okayama University. Since 2021, he has been serving as a Professor with Okayama University. His research interests include operating systems and computer security. He is a member of the IPSJ, IEICE, ACM, IEEE, and USENIX.