Kernel-Level Malware Analysis and Behavioral Explanation Using LLMs

Narumi Yoneda, Ryo Hatano and Hiroyuki Nishiyama
Department of Industrial and Systems Engineering, Graduate School of Science and Technology, Tokyo University of Science, 2641 Yamazaki, Noda, Chiba, Japan
7423530@ed.tus.ac.jp, {r-hatano, hiroyuki}@rs.tus.ac.jp

Keywords: Dynamic Analysis, System Call, LLM, Cybersecurity.

Abstract: In this study, we collected data on malware behavior and generated explanatory descriptions using a large language model (LLM). The objective of this study is to determine whether a given malware sample truly exhibits malicious behavior. To collect detailed information, we modified the Linux kernel to build a system capable of capturing information about the arguments and return values of invoked system calls. We subsequently analyzed the data obtained from our system for indications that the malware exhibited malicious or anti-analysis behavior. Additionally, we assessed whether the LLM could interpret this data and provide an explanation of the malware behavior. This approach constitutes a shift in focus from the method of attack, which is examined in the detection of the malware family, to an evaluation of the malicious nature of the actions performed by the malware. Our inferences demonstrated that our data could represent both what the malware "attempted to do" and what it "actually did," and the LLM was able to accurately interpret this data and explain the malware behavior.

1 INTRODUCTION

1.1 Linux and Malware

The implementation of security mechanisms for Linux-based systems is critical to the effective implementation and maintenance of operating systems (OS). As an open-source operating system, Linux is widely adopted across various domains such as application in web servers, cloud infrastructures, mobile devices (Android), and embedded systems. According to (W3Techs, 2024), Linux is the dominant OS for web servers with a market share exceeding 85%. In addition, major cloud-computing platforms, such as Amazon Web Services, Google Cloud, and Microsoft Azure, offer virtual machines based on the Linux OS. However, its widespread adoption makes Linux an attractive target for attackers, and the total number of Linux malware instances is increasing (TrendMicro, 2024).

A malware is equipped with mechanisms that enable it to detect and circumvent attempts at analysis. These mechanisms can be broadly categorized into two types. Anti-virtual machine (anti-VM) techniques detect whether malware is being executed in a virtual environment. The second type is anti-debugging functionality, which detects the presence of a debugger or identifies whether the system is in an environment set up for analysis. We refer to these functionalities collectively as "anti-analysis techniques" and elucidate the subject in detail in Section 2.

1.2 System Call and Dynamic Analysis

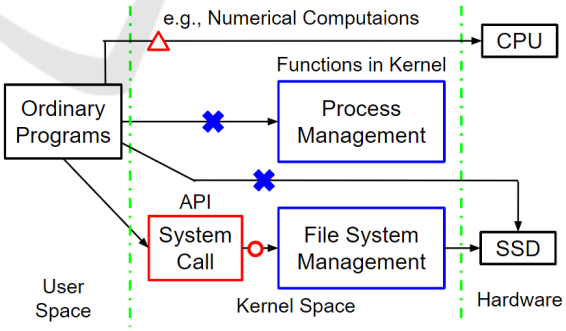

Figure 1: Overview of system call.

A system call represents the sole application programming interface (API) that allows ordinary user space programs to access the functionalities provided by the OS (see Figure 1). As of February 2024, 398 system calls have been implemented in kernel source code[1]. The kernel space refers to the memory region man-

[1]Listed in arch/x86/entry/syscalls/syscall_64.tbl in the kernel source code, ver. `6.1.0-22-amd64`.

Yoneda, N., Hatano, R. and Nishiyama, H.
Kernel-Level Malware Analysis and Behavioral Explanation Using LLMs.
DOI: 10.5220/0013149500003890

In *Proceedings of the 17th International Conference on Agents and Artificial Intelligence (ICAART 2025) - Volume 3*, pages 443-450
ISBN: 978-989-758-737-5; ISSN: 2184-433X

aged by the kernel. This region is granted a high privilege level and ordinary user programs are not permitted to access it directly. By contrast, the user space refers to the memory area outside the kernel space. The user space contains nearly all software used when a user is operating a machine, such as office applications, web servers, multimedia software, and command-line tools. These programs access the kernel functions through system calls. Therefore, most ordinary programs cannot directly access kernel functions or hardware devices without system calls[2]. Therefore, system-call sequences reveal the OS functionalities used by a program during its execution.

The evaluation of a program through its execution and the analysis of its behavior is known as dynamic analysis (DA). DA is primarily used for cybersecurity (Section 2) and programming language translation (Yoneda et al., 2024). The DA can be performed at various levels of granularity. At finer levels, machine instruction or memory access-level analyses are often employed (e.g., (Cohen and Nissim, 2018)). However, such fine-grained analyses present challenges, including the significant overhead incurred and the additional effort required to render the collected data interpretable to humans. By contrast, system call-level analyses are coarse-grained. Coarse-grained methods are more selective in terms of the information that they capture, such as system call invocations, resulting in lower overhead and easier interpretation than fine-grained methods. We refer to the data obtained from system call-level DA as "DA data" in the following sections.

1.3 Objective of this Study

The objective of this study is to determine whether a given malware sample truly exhibits malicious behavior in a test environment. However, despite numerous studies exploring malware detection using various levels of DA and machine learning (ML), conclusively demonstrating with clear evidence that malicious behavior occurred in a test environment remains a significant challenge. As DA involves executing a target program and converting its behavior into time-series data, the absence of malicious behavior during the analysis period is a critical limitation. This limitation must be verified, because there are cases in which even when a user attempts to execute malware, the intended functionality may not be achieved. For example, as explained in Section 1.1, malware may conceal its malicious behavior using anti-analysis techniques or fail to execute correctly because of the absence of the necessary libraries in the test environment. In addition, if this limitation is not verified, there is a risk of generating data on malware behavior that lacks information on malicious behavior. However, traditional ML-based malware-detection methods can classify samples as malicious. We emphasize that this issue is fundamentally distinct from false positives (i.e., misclassifying cleanware as malware). We are concerned with the possible scenario in which ML models classify data lacking malicious behavior as malware, thereby creating an illusion of successful detection. This is because ML does not directly detect malicious behavior; instead, it establishes a decision boundary between labels based on the provided features. Therefore, we do not consider "whether the classified label is malware or cleanware" and "whether malicious behavior was exhibited or not" to be equivalent. In other words, rather than comparing malware with cleanware, the determination should be based on whether the malware actually exhibited malicious behavior.

For these reasons, we consider the inability to provide conclusive evidence that the sample exhibits malicious behavior in a test environment to be a key issue. To address this, we tested the following four hypotheses:

(1) DA data can provide evidence of the malicious behavior exhibited by malware.

(2) The DA data can provide evidence of anti-analysis behavior exhibited by malware.

(3) By inputting DA data into a large language model (LLM), the DA data can be interpreted, and the malicious behavior of malware can be explained.

(4) The anti-analysis behavior of malware can be explained by inputting the DA data into the LLM.

Our Proposal is to shift the focus from an examination of the attack method, such as the determination of the malware family, to an evaluation of the maliciousness of the behaviors exhibited by the program. We employed LLMs because given that some LLMs, such as ChatGPT 4o[3], possess background knowledge of system calls and malware behavior, we believe that they can interpret DA data and explain the occurrence of malicious activity with concrete evidence. Additionally, we believe that LLMs are compatible with DA data since DA data consist of highly interpretable string-formatted data that provide insights into the behavior of computer programs. Moreover, one of the key advantages of LLMs is their *discussion* capabilities. That is, after interpreting the DA data that indicate malware behavior, LLMs

[2] As an exception, some operations such as numerical computations can directly invoke CPU instructions.

[3] https://chatgpt.com/?model=gpt-4o

should provide possible recovery strategies and methods to prevent future attacks through continuous dialogue and discussion.

The ultimate goal of this study is not only to detect malware but also to provide evidence that the malware exhibited malicious behavior in the test environment. Therefore, this study examines both the presentation of evidence from DA data and the explanation of malware behavior by leveraging DA data in conjunction with an LLM.

The remainder of this paper is organized as follows: Section 2 reviews the relevant literature with an emphasis on investigations of anti-analysis techniques, malware detection, the generation of malware explanations, and the DA techniques. Additionally, we draw a contrast between the approaches applied in these studies and ours. Section 3 presents the proposed method, including the developed DA system and the preprocessing method used to create a prompt for the LLM. Section 4 describes the experimental environment, collection, and execution of malware samples, and the use of LLMs. Section 5 presents the experimental results, followed by a discussion including an example of DA data that provides insight into malware behavior.

2 RELATED WORK

As explained in section 1.1, malware can employ anti-analysis techniques. According to (Chen et al., 2016) and (Bulazel and Yener, 2017), malware can detect debuggers or virtualizations by leveraging Windows APIs, specific fields within process environment blocks, and certain CPU instructions. In addition, to detect signs of analysis, malware may probe the environment, such as user profiles, running processes, and drivers, and monitor user interactions. Therefore, they examined anti-analysis techniques by focusing on how these features were implemented using various approaches. By contrast, our study investigates which system calls are involved in achieving anti-analysis behavior. While their work attempted to generate an interpretation of anti-analysis techniques depending on their experience, our study generated it depending on the DA data and the LLM perspective.

Several studies have focused on malware detection and have employed various DA and ML methods. For example, (Cohen and Nissim, 2018) and (Panker and Nissim, 2021) proposed frameworks for malware detection. They simulated a cloud-computing environment by running web and mail server services on VMs. They employed the memory level DA, extracted features from volatile memory dumps using the Volatility Framework[4], and detected malware using ML. (Nissim et al., 2018) obtained information about invoked system calls by analyzing volatile memory dumps. They combined sequential mining with ML algorithms to detect malware. Therefore, they employed both DA and ML because their primary goal was detection. However, as mentioned in Section 1.3, we want to emphasize that the determination of whether a sample is malicious should be based on concrete evidence of malicious behavior rather than a simple comparison with benign samples. Therefore, our study focuses on using LLMs to generate behavioral explanations and discusses maliciousness based on DA data.

(Sun et al., 2024) generated behavioral explanations of malware similar to a cyberthreat intelligence report by inputting DA data into an LLM (such as `ChatGPT 3.5`). To address the challenges in processing large-scale low-level DA data, they proposed two methods: Attack Scenario Graph (ASG) and Natural Language Description (NLD) transformations. ASG reduces the data volume by mapping DA data into a graph representation, whereas NLD converts low-level system calls into higher-level descriptions; for example, an `execve` system call becomes "execute a program." In contrast to their focus on the limitations of LLM, our study focused on the limitations of traditional malware analysis and detection techniques. We generated behavioral explanations based on the rationale that malware should be determined based on evidence of malicious behavior, while focusing on serving resources for cybersecurity professionals by automating the translation of DA data into human-readable reports.

DA techniques have been investigated to gain deeper insights and improve analysis efficiency. (Otsuki, 2016) developed Alkanet, a DA system for analyzing malware, designed to target Windows XP. It hooks system calls by setting hardware breakpoints at the jump targets of the `sysenter` instruction (which transitions to kernel mode) and `sysexit` instruction (which transitions to user mode). They focused on improving the execution efficiency by narrowing down information collection to system calls, which are typically indicative of the exhibition of malicious behavior by a program. (Hsu et al., 2023) modified the Linux kernel to prevent malware execution. They extracted code from the memory, created an ELF[5] file, and submitted it to the VirusTotal API for analysis.

[4] A memory analysis tool that can extract various features such as process information and network connections.

[5] An executable and linkable format (ELF) file is an executable binary file used in Linux. This format is analogous to the EXE file format used by Windows.

If malware was detected, the process was terminated using a kernel. We developed a DA system by modifying the Linux kernel. Detailed descriptions of our DA system and the motivations for its development are provided in Section 3.1.

3 METHODS

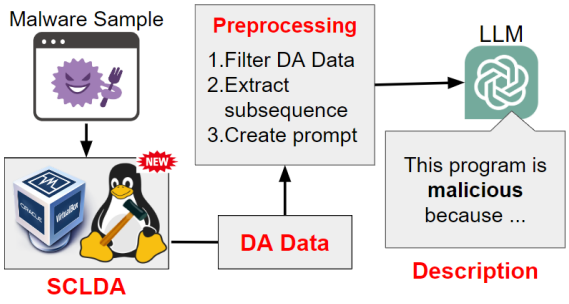

Figure 2: Overview of our approach.

Figure 2 presents an overview of the proposed approach. First, we perform system-call-level DA on malware samples to obtain DA data. We then preprocess the collected DA data and generate explanatory text by using it as the input for the LLM. Finally, we examine whether the DA data contains evidence of malicious and anti-analysis behavior with the generated text for support. The following sections describe each of these processes in detail.

3.1 System Call Level DA

As mentioned in Section 1.2, DA is a method by which the behavior of a program can be analyzed during its execution. While tools such as the *strace* command were designed to perform system-call-level DA in the user space of a Linux environment, we choose to modify the kernel source code directly to obtain more interpretable data on the program behavior from the kernel space. Moreover, performing DA through kernel source code modification may reduce the likelihood of interruptions caused by the anti-analysis technique employed by malware (explained in Section 5.3). To run the modified kernel, we construct a virtual environment using VirtualBox. By subjecting the target programs to DA within this environment, we are able to collect interpretable data that effectively captures the behavior of the program, regardless of the programming language or file type.

The information obtainable through our DA system, SCLDA[6], is summarized in Table 1. We modify the kernel source code related to the functionalities of system calls, networks, and process creation. First, we modify the content of the `SYSCALL_DEFINE` macro[7] to obtain the information in Table 1.

Table 1: Information collected by SCLDA.

Name	Meaning
Clock	Time elapsed since the kernel was started
TID	Thread ID that invoked the system call
Syscall ID	Unique ID of a system call
Retval	The return value of the system call
Arguments	The arguments passed to the system call

Second, we modify the source code of the network functionalities to transmit the collected information to our server program in the host environment through TCP/IP communication. To enhance the efficiency of this transmission, the information is accumulated in a buffer within the kernel. Once the buffer exceeds a certain threshold, a kernel thread (kthread), called the `sclda_thread`, is created to transmit this information. Consequently, processes executing user space programs are only required to handle the overhead associated with retrieving information, whereas the overhead of transmitting it imposes no additional burden.

Third, we modify the process creation mechanism to identify the origin of the invoked system calls. Because we modified the `SYSCALL_DEFINE` macro, information regarding all the system calls invoked by all the programs in the analysis environment is acquired. In the following section, we describe a method for capturing only the system calls invoked by a target program.

3.2 Thread-Level Aggregation

In modern Linux, processes and threads are uniformly managed as *tasks* and represented by a `task_struct` structure. Each `task_struct` is assigned a unique thread ID (TID). Therefore, we utilize the TID information to identify the origin of the system calls and determine whether the system-call information is relevant to the execution of the target program.

To illustrate the identification of TIDs associated with the target program, let us consider an example of running the `sample.elf`. First, bash invokes a system call (either `fork`, `vfork`, or `clone`) to create a task to run `sample.elf`. The created task then invokes a system call (either `execve` or `execveat`) to begin executing `sample.elf`. Subsequently, the task that initiated the execution of `sample.elf` may invoke system

[6]https://github.com/naru3-99/sclda

[7]The `SYSCALL_DEFINE` macro implements individual system calls. This macro is expanded into a kernel function before it is compiled.

calls such as `fork` to create new tasks that execute the components of the `sample.elf`. Therefore, the TIDs relevant to the target program include both the TID that initiated the execution of `sample.elf` and any descendant TIDs created by that TID.

To trace all the tasks generated in the test environment, we hooked the `kernel_clone` function[8]. This approach allowed us to gather the following data: the TID of the task that invoked the `kernel_clone` function (parent TID), the TIDs of the newly created tasks (child TIDs), and the executable file name associated with the created task, such as `sample.elf`. Therefore, we can trace the parent-child relationships of the TIDs.

DAData 1: Example of `execve` system call invocation

```
1 execve,retval=0,
2 filename=./sample.elf,
3 argv=[./sample.elf,],
4 envp=[SHELL=/bin/bash,]
```

To isolate only the system calls invoked by the target program, it is necessary to identify the TID that initiates the execution of the target program. To achieve this, we leverage information from `execve` or `execveat` system calls. These system calls receive parameters such as the filename to execute, commandline arguments, and environmental variables. For example, when command `./sample.elf` is executed from the bash shell, the SCLDA generates the DA data shown in DAData 1. To identify the relevant TID, we reference the `filename` (line 2 in DAData 1) and `argv` (line 3 in DAData 1) fields, searching for matches with the target program name. In the following examples, the `retval=` and `(arg1 Name)=` fields are omitted from the presentation of the DA data to conserve space. Consequently, we can differentiate the system call information relevant to the target program from irrelevant information.

3.3 Preprocessing

The following preprocessing steps are performed to construct prompts for input into an LLM:

(1) Extract subsequences of system calls that may indicate malicious behavior or anti-analysis techniques.

(2) Construct instructions for LLMs with DA data.

Regarding (1), this preprocessing is essential because of the limitations in the number of tokens that the LLM can handle. Given the scope of this study, extraction was performed manually rather than through automation. However, because automating this preprocessing appears to be feasible, it is suggested as a direction for future work. An example of a constructed prompt in (2) is as follows:

The Instructions. "We analyzed a suspicious program and obtained information regarding system-call invocation. We extracted an important subsequence of the system calls. Your task is to explain the overall behavior of the subsequence using fewer than 100 words. Specifically, mention whether the behavior can be considered malicious or whether it can be classified as anti-debugging or anti-VM behavior."

The Format of the DA Data. "(System Call Name), `retval` = (Return Value), `Arg1 Name` = (Argument1 Value). Note that the return and argument values are represented as internal numbers (e.g., int or long types) and are not shown as readable names such as `-EFAULT` or `-EINVAL`."

The DA Data. The subsequence of the system calls, such as DAData 1, is pasted.

In this study, we initially employ the aforementioned prompt to input a critical subsequence of system calls. Subsequently, we ask additional questions to the LLM directly, focusing on the points of interest identified by the authors.

4 EXPERIMENT

The experimental conditions are listed in Table 2. The programs, resources, and data obtained from our experiments are publicly available from the GitHub repository[9]. Additional experimental conditions were as follows:

- Kernel modifications were completed for 230 of the 398 system calls, specifically those from zero to 173, along with other significant system calls.
- The Malware Bazaar[10] API was used to collect malware samples. Using `file` command[11], we filtered and retained only the 64-bit ELF files.
- To mitigate the risks of network-based malware propagation and data leakage, a local PC was isolated from the Internet.

[8]The `kernel_clone` function is the primary routine responsible for task creation. The `fork`, `vfork`, and `clone` system calls are wrappers around this function.

[9]https://github.com/naru3-99/ICAART2025

[10]Malware Bazaar is a platform for sharing malware samples. https://bazaar.abuse.ch/

[11]The `file` command displays the format, target architecture, and whether the architecture is 32- or 64-bit when executed on an ELF file.

Table 2: Experimental Conditions.

Name	Configuration
Local PC	Core i9-13900k CPU, RTX4090 GPU, Windows 11 Pro
Python	Python 3.11.9
Virtualization software	Virtual Box 7.0.10 r158379
Virtual OS	Debian 12.5.0, modified Kernel 6.1.0-22-amd64
Virtual Environment	8 CPUs, 16GB RAM
LLM	ChatGPT 4o, as of September 2024

- To prevent interactions between malware samples, the system was restored to a clean state using the VirtualBox snapshot[12] feature prior to each execution.
- Because the exact completion time for each malware execution could not be determined, we standardized the execution duration for all samples to one minute.

5 RESULTS AND DISCUSSION

In this section, we discuss the validity of Hypotheses (1)-(4) presented in Section 1.3. In this study, we executed 17 malware samples and collected 34 patterns of DA data, with and without `sudo` privileges. Through manual analysis, we identified 72 subsequences of system calls that potentially indicated malicious or anti-analysis behaviors. Owing to the impracticality of presenting all the sequences obtained in this study, we selected the most significant sequences and provided detailed explanations.

5.1 Delete System Commands

DAData 2: Deletion of critical system commands.

```
1 unlink, -13, /sbin/reboot
2 unlink, -13, /usr/sbin/reboot
3 unlink, -2,  /bin/reboot
4 unlink, -2,  /usr/bin/reboot
```

One malware sample attempted to delete critical system commands (executable files) required for shutting down and rebooting the system, which was clearly identified as malicious behavior. In the case of DAData 2, malware was executed without `sudo` privileges. As a reminder, the first element represents the system call name, the second represents the return value, and the subsequent elements correspond to arguments. In this case, an `unlink` system call was used to remove the file, as specified by the first argument. Although omitted here for brevity, an `unlink` system call was invoked to delete the `shutdown`, `poweroff`, and `halt` commands (with the same paths as those shown in DAData 2). These subsequences were also included as inputs for the LLM. The key explanation provided by LLM is summarized as follows:

- The program attempted to delete important system files related to shutdown and reboot commands but failed.
- The return values indicate permission errors (`-13`) or file-not-found errors (`-2`).
- This behavior can be considered malicious.

Next, when the same malware was executed with `sudo` privileges, the return value of the `unlink` system call, which targeted commands in the `/sbin` folder, changed to zero. By contrast, for commands in other directories, the return value either changed from `-13` to `-2` or remained at `-2`. From this sequence, it can be inferred that the system commands were present in the `/sbin` folder and were successfully deleted, whereas no system commands were found in other directories. The key explanation provided by LLM is summarized as follows:

- The program successfully deleted the critical system binaries in `/sbin`, as indicated by the return values.
- The other attempts to delete similar files fail owing to their absence.
- This behavior is highly malicious, as it could destabilize or incapacitate the system.

Based on the above results, the DA data provided conclusive evidence of malicious behavior; the malware attempted to delete critical system commands and, in some cases, succeeded in deleting them, thus supporting Hypothesis (1). Furthermore, the LLM not only recognized that the malware used the `unlink` system call to attempt file deletion but also correctly identified that a return value of `-13` indicated insufficient permissions, whereas a return value of `-2` signified that the file did not exist[13]. This demonstrates

[12]The snapshot functionality in VirtualBox allows users to save the state of a VM at a specific point in time, enabling them to revert to this saved state later.

[13]According to the `include/uapi/asm-generic/`

that the LLM can provide appropriate explanations for the system-call subsequences observed during actual malware execution, as represented by the DA data, thereby supporting Hypothesis (3).

5.2 Self-Deletion

DAData 3: Self-Deletion.

```
1 readlink, 93, 4096,
2 /proc/self/exe,
3 /home/naru3/workdir/test/mal.elf
4 unlink, 0,
5 /home/naru3/workdir/test/mal.elf
```

This sample attempted to hide itself by deleting its executable files (DAData 3). The Linux virtual file system, specifically `/proc/self/exe`, contains a symbolic link to the executable that the current process runs. Therefore, using the `readlink`[14] system call on `/proc/self/exe`, malware could obtain the full path to its own executable. Then, it deleted itself using an `unlink` system call. The key explanation provided by the LLM is summarized as follows:

- The `readlink` system call was invoked to obtain the location of the file or verify its integrity.
- A deletion is a typical anti-analysis tactic employed by malware to hinder detection and forensic analysis.

Based on these results, the DA data successfully provided evidence of malicious behavior; that is, the malware obtained its full path and subsequently deleted itself, thus supporting Hypothesis (2). Furthermore, the LLM not only accurately interpreted the DA data and generated a correct explanation but also pointed out that the `readlink` system call was employed to verify file integrity, thereby supporting Hypothesis (4).

5.3 Killing Other Processes

DAData 4: Killing other processes.

```
1  open, 1, 32768, 0,
2  /proc/1025/status
3  read, 1421, 1, 4096,
4  Name:   sudo
5  Pid:    1025
6  PPid:   677
7  TracerPid:  0
8  ~~(omission)~~
9
10 open, 2, 32768, 0,
11 /proc/677/cmdline
12 read, 6, 2, 4096, -bash\x00
13 kill, 0, 1025, 9
```

This sample detected a process initiated by bash shell and sent a `SIGKILL` signal[15] (DAData 4). The `/proc/[PID]/status`[16] file is a virtual file that provides information about the process identified by the `[PID]` in the path. In this case, it is likely that the sample read `/proc/1025/status` to retrieve the parent process ID (PPID). Subsequently, the sample accessed the `/proc/677/cmdline` file, where `677` is the retrieved PPID. The `/proc/[PID]/cmdline` file contains the command-line arguments used when the process was executed. In DAData 4, the argument was `-bash`, indicating that the process was initiated by the bash shell. The sample then sent a `SIGKILL` signal to the process with PID = `1025` using the `kill` system call. The key explanation provided by LLM is summarized as follows:

- The program read the status information of process `1025`, accessed the command line of process `677`, and sent a `SIGKILL` signal to terminate process `1025`.
- This behavior suggests process monitoring and termination, often associated with malicious intent such as forcibly terminating security or monitoring processes to evade detection.

Using the same approach shown in DAData 4, the same sample attempted to terminate the kthread, named `sclda_thread`. As described in Section 3.1, the `sclda_thread` transmits the DA data to a server in the host environment. Although the return value of the `kill` system call was zero, it is likely that the `sclda_thread` was not terminated, because it is impossible to terminate the kthread from the user space. Moreover, as no anomalies were detected in the captured TCP packets, we concluded that the `sclda_thread` was not terminated. The key explanation provided by LLM is summarized as follows:

`errno-base.h` header file, which defines error codes in the kernel source, the explanation of the LLM is accurate.

[14]The `readlink` system call retrieves the target path of the symbolic link specified by the second argument and stores it in the buffer provided by the third argument.

[15]The `kill` system call sends a signal to the target. The first argument is the PID and the second specifies the signal to be sent. The `SIGKILL` signal (9) forcefully and immediately terminates the process without allowing it to catch, block, or clean.

[16]`TracerPid` (line 7 in DAData 4) displays the PID of a process that analyzes this process using tools such as `strace`. In the case of DAData 4, because no user-space process was analyzing this process, the displayed value was zero.

- This program sent a `SIGKILL` signal to terminate the `kthread` (i.e., `sclda_thread`).
- kthread operates at a privilege level superior to that of user-space processes, making it impossible to terminate a kthread from user space.

Based on these results, it can be concluded that the DA data accurately reflected the behavior exhibited by the program, supporting Hypotheses (1) and (2). In addition, it became evident that SCLDA demonstrated a certain level of resistance to anti-analysis techniques. The LLM correctly interpreted the DA data and provided an appropriate explanation, thereby supporting Hypotheses (3) and (4).

6 CONCLUSION

In this study, we generated behavioral explanations for malware by inputting DA data into LLMs. This approach is based on the understanding that dynamic analysis, which involves executing a target program and converting its behavior into data, is critically limited by the absence of malicious behavior during the analysis period. Additionally, we demonstrated that our DA system (i.e., kernel-level analysis) has the potential to partially disable malware anti-analysis techniques. This offers a clear advantage over conventional methods that rely on user-space tools, such as `strace`. We have made our DA system publicly available, including the modified kernel source code and other necessary programs.

We tested the four hypotheses presented in Section 1.3. Based on the experimental results and discussion, the following conclusions were drawn regarding the hypotheses:

(1) The DA data provided evidence of the malicious behavior exhibited by malware.

(2) The DA data indicated the use of anti-analysis techniques by the malware.

(3) The LLM was able to interpret DA data and explain the malicious behavior of malware.

(4) The LLM was able to explain the anti-analysis behavior of malware.

In particular, we found that the DA data could reveal both what the malware "attempted to do" and what it "actually did." Therefore, even if the execution of malicious behavior failed and no actual damage occurred, it was shown that there is a possibility of determining it as malware based on "it attempted to do."

REFERENCES

Bulazel, A. and Yener, B. (2017). A survey on automated dynamic malware analysis evasion and counter-evasion: PC, mobile, and web. In *Proceedings of the 1st Reversing and Offensive-Oriented Trends Symposium*, ROOTS, New York, NY, USA. Association for Computing Machinery.

Chen, P., Huygens, C., Desmet, L., and Joosen, W. (2016). Advanced or not? a comparative study of the use of anti-debugging and anti-VM techniques in generic and targeted malware. In *IFIP International Information Security Conference*.

Cohen, A. and Nissim, N. (2018). Trusted detection of ransomware in a private cloud using machine learning methods leveraging meta-features from volatile memory. *Expert Systems with Applications*, 102:158–178.

Hsu, F.-H., Hunag, J.-H., Hwang, Y.-L., Wang, H.-J., Chen, J.-X., Hsiao, T.-C., and Wu, M.-H. (2023). A kernel-based solution for detecting and preventing fileless malware on linux. *Preprints*.

Nissim, N., Lapidot, Y., Cohen, A., and Elovici, Y. (2018). Trusted system-calls analysis methodology aimed at detection of compromised virtual machines using sequential mining. *Knowledge-Based Systems*, 153:147–175.

Otsuki, Y. (2016). *Research on System Call Tracing for Malware Analysis based on Virtualization Technology*. PhD thesis, Ritsumeikan University.

Panker, T. and Nissim, N. (2021). Leveraging malicious behavior traces from volatile memory using machine learning methods for trusted unknown malware detection in linux cloud environments. *Knowledge-Based Systems*, 226:107095.

Sun, Y. S., Chen, Z.-K., Huang, Y.-T., and Chen, M. C. (2024). Unleashing malware analysis and understanding with generative ai. *IEEE Security & Privacy*, 22(3):12–23.

TrendMicro (2024). The linux threat landscape report. https://www.trendmicro.com/vinfo/us/security/news/cybercrime-and-digital-threats/the-linux-threat-landscape-report. Accessed: 2024-09-14.

W3Techs (2024). Usage statistics and market share of operating systems for websites. https://w3techs.com/technologies/cross/operating_system/web_server. Accessed: 2024-09-14.