Design and Evaluation of a Management Target Control Mechanism in a Function for Tracing Diffusion of Classified Information on KVM*

Hideaki Moriyama[1], Toshihiro Yamauchi[2], Masaya Sato[3], and Hideo Taniguchi[2]†

[1] Ariake College, Fukuoka, Japan
hideaki@ariake-nct.ac.jp
[2] Okayama University, Okayama, Japan
geoff@cs.miami.edu
[3] Okayama Prefectural University, Soja, Japan

Abstract

The leakage of classified information managed on computer systems can cause serious damage to organizations and individuals. In recent years, tasks that handle classified information have increasingly been executed in virtualized environments, and there is a growing need for a mechanism that can track and control the diffusion of classified information without modifying guest operating systems (OSs) or applications. Against this background, a study on a diffusion tracing function for classified information using the Kernel-based Virtual Machine (KVM) has been conducted. This function hooks system calls issued from the guest OS through the virtual machine monitor (VMM), identifies processes and files that may contain classified information (referred to as managed processes and managed files), and records the information as logs. This paper presents a management control mechanism that enhances the manageability of a diffusion tracing function implemented on the KVM. The mechanism allows users to dynamically add or remove managed processes and files at runtime and to monitor current targets through a proc file system-based interface. Evaluation results demonstrate that the proposed approach introduces negligible overhead and provides a lightweight, non-intrusive foundation for controlling managed targets in virtualized environments.

Keywords: Information leakage monitoring, Virtual machine monitor, Management control mechanism, Process file system

1 Introduction

When classified information managed within a computer system is leaked to the outside, it can cause serious losses to companies and individuals. In recent years, tasks involving classified information are often carried out in virtualized environments. In such environments, it is essential to have a technology that allows users and administrators to obtain and control the diffusion of classified information on OSs and applications running on virtual machines, without directly modifying the guest OSs.

To address this problem, previous studies [1, 2] proposed a function for tracing the diffusion of classified information that operates on a virtual machine monitor (VMM) and does not require modifications to the guest OS. We implemented this function on Kernel-based Virtual Machine

*Proceedings of the 9th International Conference on Mobile Internet Security (MobiSec'25), Article No. 77, December 16-18, 2025, Sapporo , Japan. © The copyright of this paper remains with the author(s).
†Corresponding author

(KVM). The function hooks and analyzes system calls issued by the guest OS through the VMM to identify processes and files that may possess classified information (hereafter referred to as managed processes and managed files), and outputs this information as logs.

However, with conventional log analysis methods, it is difficult to obtain the managed targets at arbitrary points in time or to modify them dynamically, which has been a major obstacle to efficient management operations. To overcome this limitation, this study enhances the existing KVM-based diffusion tracing function by introducing a management control mechanism that significantly improves operational manageability. Specifically, we design and implement two key features: (i) a procfs-based interface that enables users to instantly obtain the current list of managed processes and files without relying on log parsing, and (ii) a dynamic control mechanism that allows adding or removing managed targets at runtime through simple write operations to virtual files. These enhancements eliminate the inefficiency of conventional log-based approaches and provide flexible, lightweight management without modifying guest OSs. Furthermore, we evaluate the proposed mechanism in terms of processing overhead and its impact on guest OS performance, demonstrating that the additional cost is negligible while maintaining the functionality of the tracing system.

The main contributions of this paper are as follows:

1. We design and implement a procfs-based management mechanism that allows dynamic addition and removal of managed processes and files in the function for tracing diffusion of classified information on KVM.

2. The proposed mechanism eliminates the need to parse large volumes of log data by providing direct, on-demand access to the current list of managed targets.

3. The design achieves low overhead without modifying the guest OS, ensuring practical deployability in virtualized environments.

4. Evaluation results using LMbench and kernel build benchmarks demonstrate that the proposed mechanism maintains stable performance under representative workloads.

The function for tracing diffusion of classified information is designed for use in virtualized environments based on KVM. Owing to its architectural design, the proposed mechanism can also be applied to cloud and edge computing infrastructures that support the mobile and IoT environments. In this regard, this study contributes to enhancing mobile internet security.

2 Function for Tracing the Diffusion of Classified Information

2.1 Overview

The function for tracing the diffusion of classified information [1, 2] is a mechanism that tracks the diffusion of classified information based on system call events. It treats processes and files that may possess classified information as managed processes and managed files, respectively, and monitors how classified information spreads by intercepting and analyzing their system calls.

Figure 1 shows an overview of the VMM-based tracing function, which uses a kernel-based virtual machine (KVM) and a 64-bit Linux OS with a 3.6.10 kernel as the VMM and the guest OS. The operation flow of this function is as follows.

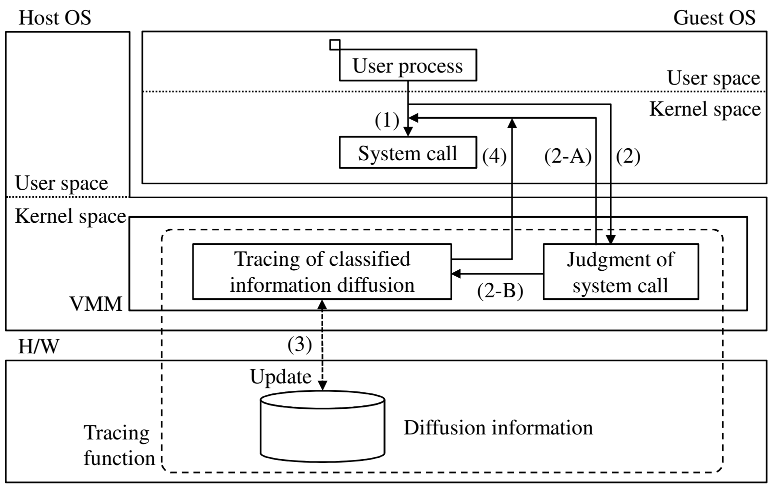


Figure 1: Overview of the tracing function

1. A user process in the guest OS issues a system call.
2. The virtual machine monitor (VMM) detects this system call.
3. The function determines whether the system call is related to the diffusion of classified information.
   (a) If it is not related, the control is immediately returned to the guest OS.
   (b) If it is related, the function collects information such as process IDs and file inode numbers and updates its internal management information.
4. When new managed processes or managed files are detected, their information is logged to `/var/log/messages`.
5. Control is returned to the guest OS to continue normal execution.

Through this architecture, the diffusion of classified information can be detected at the VMM layer without modifying the source code of the guest OS. Users can obtain all detected diffusion information by inspecting the system log at `/var/log/messages`.

The function for tracing the diffusion of classified information is implemented as a kernel module loaded on the host OS in a KVM environment. When a system call issued by the guest OS is executed, it is trapped using the mechanism provided by KVM and hooked on the VMM side to determine whether it is related to the diffusion of classified information. If it is determined to be related, the corresponding process or file is registered as a managed process or managed file, respectively.

The management information stores managed processes in an array indexed by their process IDs (PIDs) and managed files in an array of structures containing their inode numbers and path names. This design allows managed targets to be easily added or removed, enabling efficient tracking and management.

2.2 Limitation of the Conventional Approach

In the original implementation, the function only recorded diffusion events in the system log, and it was not possible to directly obtain the current list of managed targets. To identify the current set of managed processes and managed files at an arbitrary point in time, the user had to:

1. Execute a log parsing program.
2. Collect all log entries from `/var/log/messages`.
3. Reconstruct the current list by analyzing the history of additions and deletions.

This approach had several drawbacks. First, as the number of log entries increased, the time required for parsing grew significantly. Second, searching and updating the target list became slower as the number of managed targets increased. Third, since the tracing function only produced log outputs and provided no interface for on-demand management, users could not control or retrieve the current list of managed targets in real time. As a result, it was difficult to efficiently manage targets or obtain real-time information during tracing.

To address these limitations, we redesigned the management mechanism so that users can obtain the current list of managed processes and files on demand via a procfs-based interface, instead of outputting it every time new diffusion events are detected. This redesign shifts from a log-based architecture to an on-demand monitoring model, eliminating unnecessary log output and improving the efficiency of log management.

3 Design of a Method for Listing Managed Targets

3.1 Problem of the Conventional Log-Based Approach

In the previous implementation of the tracing function, information about managed processes and managed files was only recorded as log entries in `/var/log/messages`. To obtain the list of currently managed targets at an arbitrary point in time, the user had to perform the following steps:

1. Execute a log parsing program as a user process.
2. Search all log entries in `/var/log/messages` in chronological order.
3. Reconstruct the current state by adding newly detected targets and removing those that were terminated or deleted.

This approach has several problems. First, as the number of log entries increases, the time required to parse them also increases significantly. Second, as the number of managed targets grows, searching and updating the list becomes increasingly time-consuming. Finally, the user must re-scan all log entries each time they want to obtain the current list, which introduces a large overhead.

These problems make it difficult to efficiently and quickly obtain the list of managed targets during tracing. To address this issue, we designed a new method that allows direct and immediate listing of managed targets without log parsing.

3.2 Overview of the Proposed Method

We implemented the proposed method by using the procfs, which is a virtual filesystem that provides dynamic kernel information as files under the `/proc` directory. In this method, the tracing function provides the current list of managed targets directly as virtual files created in `/proc`. Specifically, the following two files are provided:

- `/proc/processlist`: provides the list of managed processes
- `/proc/filelist`: provides the list of managed files

These virtual files are created when the tracing function starts and are removed when it stops. A user can obtain the current list of managed targets simply by reading these files. The operation flow is as follows:

1. A user reads `/proc/processlist` or `/proc/filelist`.
2. The tracing function scans its internal management information.
3. It outputs the information about currently registered managed processes or managed files as the content of the virtual file.

This design enables users to immediately obtain the current list without analyzing any logs, which greatly reduces the time required for management operations.

The virtual files `/proc/processlist` and `/proc/filelist` are created in the kernel module by calling `proc_create()`. They are associated with struct `file_operations`, and the read callback function outputs the content of the internal data structures.

- Managed processes are stored in a fixed-size array (32,768 entries) indexed by their process IDs (PIDs). When a process is detected as related to classified information, the corresponding array element is set to 1.
- Managed files are stored as an array of structures, each containing the inode number and the absolute pathname of the file. When a new managed file is detected, a new entry is appended to the array.

When the read callback is invoked, the module scans these data structures and outputs only the entries currently marked as active. This ensures that the output always reflects the current state of the managed targets, even when accessed at arbitrary timings.

We created a simple test case related to the diffusion of classified information and confirmed that the managed processes and managed files were correctly reflected in the lists at each point in time as expected.

Figure 2 shows an overview of the proposed management control mechanism using procfs. In addition to the tracing function described in Figure 1, this mechanism provides two proc files, `/proc/processlist` and `/proc/filelist`, and a management control processing component. Users can obtain the current list of managed processes and files by reading these proc files (6). Furthermore, managed targets can be dynamically added (7) or removed (8) through simple write operations. The information is immediately reflected in the management data structures and utilized in the tracing of classified information diffusion (9).

However, the method only supports retrieving the list of managed targets and does not support modifying them. To solve this limitation, the next Section introduces a mechanism for dynamically adding and removing managed targets.

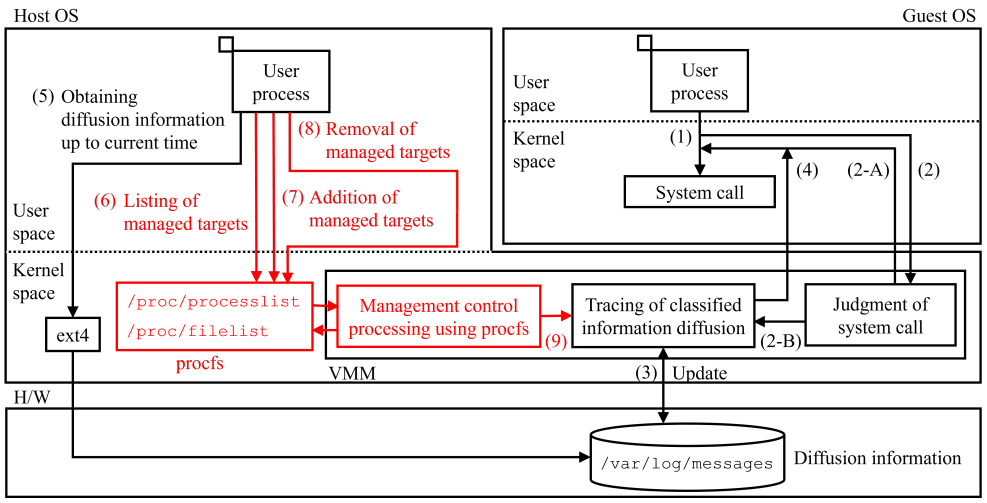


Figure 2: Overview of the management control mechanism using procfs

4 Mechanism for Dynamically Adding and Removing Managed Targets

4.1 Objective

Using `/proc/processlist` and `/proc/filelist`, users can instantly list the current managed processes and managed files. However, this method only supports listing information and does not allow adding or removing managed targets arbitrarily during execution.

Currently, the tracing function automatically updates managed targets based on detected diffusion events originating from initially registered managed files. Therefore, if a user wants to add a new important file as a managed file or remove an unnecessary target from monitoring, they cannot do so without restarting the function.

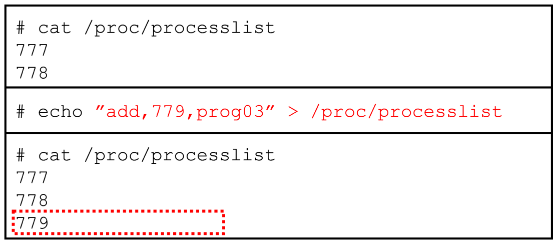
To address this limitation, we designed and implemented a mechanism that allows dynamically adding and removing managed targets using the procfs.

4.2 Design Policy

This mechanism extends the existing virtual files (`/proc/processlist` and `/proc/filelist`) by adding a write interface to them. Users can now add or remove managed targets by writing commands to these files.

This design has the following advantages:

- Users can dynamically change the managed targets at any time.
- Because the operation is performed via virtual files under `/proc`, it incurs lower overhead than physical file I/O.
- The kernel module parses and updates the information upon writing, so the changes are reflected immediately without restarting the guest OS or the VMM.

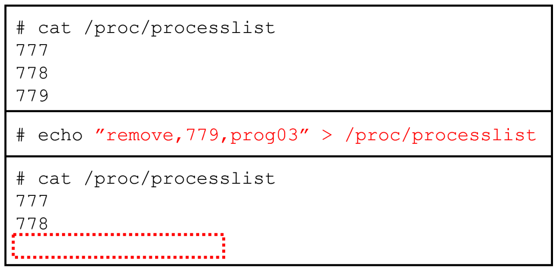


```
# cat /proc/processlist
777
778
```

```
# echo "add,779,prog03" > /proc/processlist
```

```
# cat /proc/processlist
777
778
779
```

(a) Addition of a managed process



```
# cat /proc/processlist
777
778
779
```

```
# echo "remove,779,prog03" > /proc/processlist
```

```
# cat /proc/processlist
777
778
```

(b) Removal of a managed process

Figure 3: An example of adding and removing managed processes

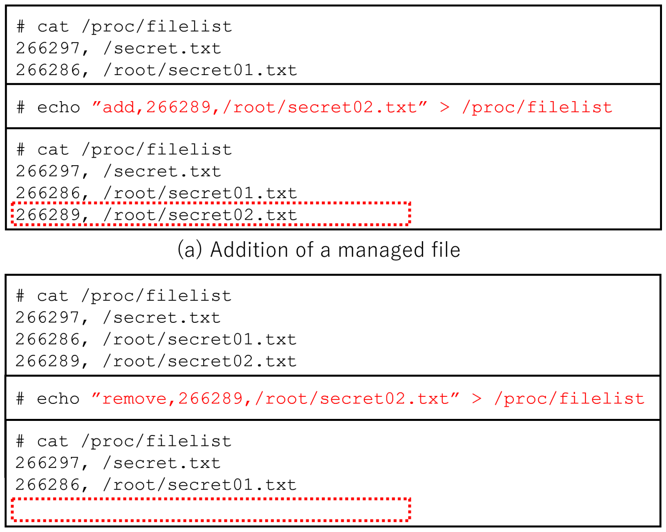
4.3 Implementation

We added a write operation function to the struct `file_operations` of `/proc/processlist` and `/proc/filelist`. When a user writes a command to these files, the kernel module parses the input string and updates the internal management information.

- Adding or removing managed processes
  - Command format: <operation>,<PID>,<process name>
  - <operation>is either add or remove
  - <PID>is process ID
  - <process name>is the name of the process
- Adding or removing managed files
  - Command format: <operation>,<inode number>,<absolute file path>
  - <operation>is either add or remove
  - <inode number>is the inode number of the file
  - <absolute file path>is the absolute pathname of the file

The `/proc` interface is intended to be used only by specific users (such as root). It is implemented as a standalone component dedicated to the tracing function and is independent from other kernel functions. The code size is small (about 180 lines), and because add/remove operations are infrequent, the impact on performance and safety is minimal.

An example of adding and removing managed processes is shown in Figure 3. In Figure 3(a), two processes (PID: 777 and 778) are initially registered. By writing "add,779,prog03" to `/proc/processlist`, the process with PID 779 can be added as a managed process. In



```
# cat /proc/filelist
266297, /secret.txt
266286, /root/secret01.txt
```
```
# echo "add,266289,/root/secret02.txt" > /proc/filelist
```
```
# cat /proc/filelist
266297, /secret.txt
266286, /root/secret01.txt
266289, /root/secret02.txt
```

(a) Addition of a managed file

```
# cat /proc/filelist
266297, /secret.txt
266286, /root/secret01.txt
266289, /root/secret02.txt
```
```
# echo "remove,266289,/root/secret02.txt" > /proc/filelist
```
```
# cat /proc/filelist
266297, /secret.txt
266286, /root/secret01.txt
```

(b) Removal of a managed file

Figure 4: An example of adding and removing managed files

Figure 3(b), writing "remove,779,prog03" to `/proc/processlist` removes the process with PID 779 from the list of managed processes.

Similarly, an example of adding and removing managed files is shown in Figure 4. In Figure 4(a), two files (inode numbers: 266297 and 266286) are initially registered. By writing "add,266289,/root/secret02.txt" to `/proc/filelist`, the file `/root/secret02.txt` (inode number: 266289) can be added as a managed file. In Figure 4(b), writing "remove,266289,/root/secret02.txt" to `/proc/filelist` removes the file `/root/secret02.txt` (inode number: 266289) from the list of managed files. Because this mechanism works as normal file write operations, users can easily use commands such as echo.

4.4 Evaluation of the Management Mechanism

We evaluated the proposed management mechanism using the procfs for listing managed targets and modifying them by dynamically adding and removing entries, based on the following three aspects. The evaluation environment is shown in Table 1.

(Evaluation 1) Processing time for outputting the list of managed processes or managed files to virtual files.

(Evaluation 2) Processing time for adding and removing managed processes.

(Evaluation 3) Processing time for adding and removing managed files.

The processing time was calculated by obtaining timestamps before and after the target operation using the `rdtscll` instruction, measuring the difference in clock cycles, and converting it to elapsed time based on the CPU clock frequency. Each operation was executed ten times under the same conditions, and the average value was used as the result.

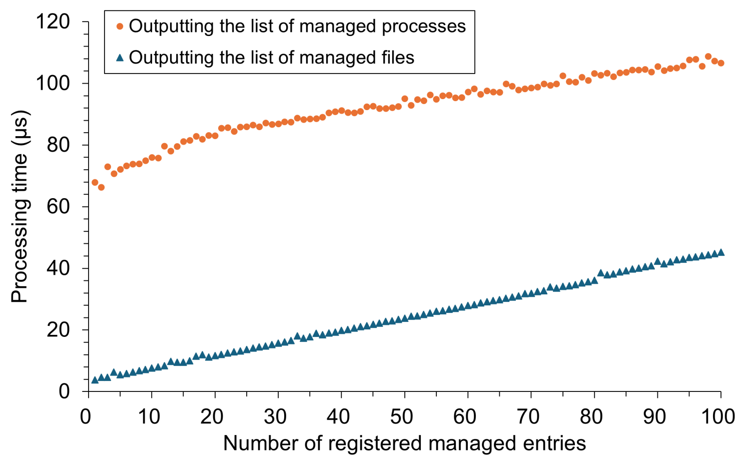


Figure 5: Processing time for outputting the list of managed processes or managed files

4.4.1 Outputting the List of Managed Targets

As Evaluation 1, we measured the processing time for outputting the list of managed processes or managed files to virtual files. Because the processing time depends on the number of managed targets, the number of entries was gradually increased from one to one hundred, and the processing time was measured. The results are shown in Figure 5.

The processing time for managed processes was 67.92 µs with one entry and increased moderately to 106 µs with one hundred entries. For managed files, the processing time increased from 3.84 µs with one entry to 45.27 µs with one hundred entries, showing an almost linear increase. This difference is considered to be due to the fact that obtaining the list of managed processes involves scanning an array indexed by PIDs. Since the list output is performed only occasionally by users, its impact on the tracing function is considered small.

4.4.2 Adding and Removing Managed Processes

As Evaluation 2, we measured the processing time for adding and removing managed processes. Similar to Evaluation 1, the number of managed processes was varied from one to one hundred during the measurements. For the removal operation, one hundred managed processes were pre-registered, and the number of removal targets was gradually increased from one to one hundred. The results are shown in Figure 6. The processing time for addition increased linearly from 4.11 µs to 351.5 µs, and the processing time for removal increased linearly from 3.52 µs to 349.97 µs.

4.4.3 Adding and Removing Managed Files

As Evaluation 3, we measured the processing time for adding and removing managed files. The number of managed files was varied from one to one hundred during the measurements. The results are shown in Figure 7. The processing time for addition increased linearly from 4.34 µs to 414.5 µs, and the processing time for removal increased linearly from 4.1 µs to 421.1 µs.

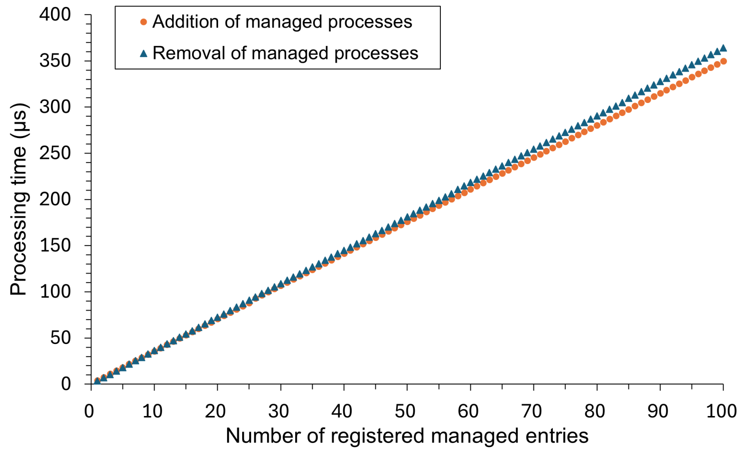


Figure 6: Processing time for adding and removing managed processes

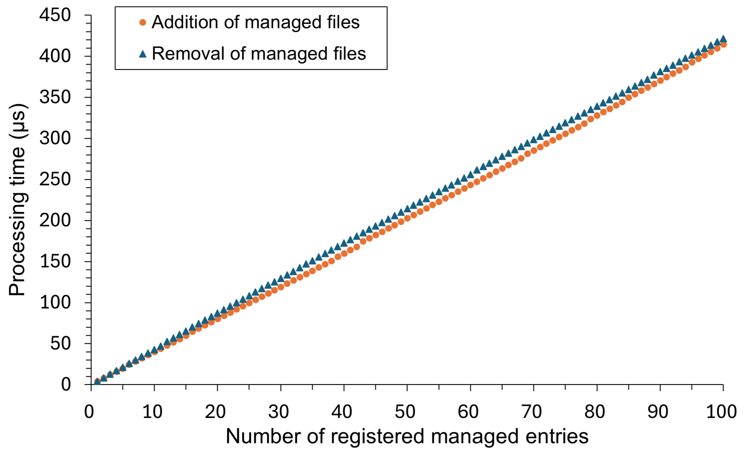


Figure 7: Processing time for adding and removing managed files

4.4.4 Summary of Evaluation Results

From Evaluations 2 and 3, the processing time for adding or removing a single managed process or managed file was between 3.5 and 4.4 µs, which indicates that the overhead introduced by the proposed mechanism is small and has little impact on the tracing function.

5 Impact of Introducing the Tracing Function and the Management Mechanism on Guest OS Performance

5.1 Objective

This Section evaluates how the proposed function for tracing the diffusion of classified information and the associated management mechanism using procfs affect the overall processing

Table 1: Evaluation environment

Processor	Intel Xeon E5-2609 (8 cores)
Memory	64 GB
Host OS	Fedora 18
Kernel	Linux 3.6.10 (64-bit)
VMM	KVM-kmod-3.6

Table 2: System call processing time measured using LMbench (µs)

System Call	**Baseline**	**Tracing**	**Management Control**
`getpid`	0.134	7.131	7.120
`read`	0.214	10.664	10.508
`write`	0.266	10.870	10.712
`stat`	0.774	7.849	7.821
`fstat`	0.246	7.301	7.289
`open+close`	1.954	16.456	16.213

performance of the guest OS. While Section 4 showed that the management mechanism itself is lightweight, it operates as a kernel module on the host OS and runs in parallel with the guest OS. Therefore, its execution may potentially interfere with the performance of processes running inside the guest OS. We measured the processing performance of the guest OS with and without the tracing function and the management mechanism to quantitatively clarify their impact.

The evaluation was conducted in the same environment shown in Table 1. We prepared three environments for comparison: a baseline environment without the tracing function, an environment with only the tracing function, and an environment with both the tracing function and the management mechanism. The guest OS was configured to run on a single physical CPU core to minimize the influence of scheduling fluctuations.

Two types of benchmarks were used: the `lat_syscall` microbenchmark from LMbench to measure system call latency, and a kernel build workload to evaluate overall system performance. For the build test, the Linux 3.6.11 source code was used, and the user, sys, and real times were measured using the `time` command.

5.2 Evaluation of System Calls Processing Performance

To clarify the potential impact of introducing the management control mechanism using procfs, we measured the system call processing time on the guest OS using the `lat_syscall` microbenchmark from LMbench 3.0. The results are shown in Table 2.

In the *tracing environment* and the *management control environment*, an additional overhead occurs when system calls issued by the guest OS are hooked from the host OS to determine whether they are involved in the diffusion of classified information. For this reason, the system call processing time in these environments is longer than that in the *baseline environment.* However, the processing times in the tracing environment and the management control environment are almost the same, indicating that introducing the management control mechanism does not cause additional delays.

Table 3: Kernel build processing time measured with the `time` command (seconds)

Environment	User (s)	Sys (s)	Real (s)
Baseline	126.6953	29.3433	162.5103
Tracing	133.3983	50.7623	191.4727
Management	132.9000	51.0137	191.2370

5.3 Evaluation of Kernel Build Processing Performance

To clarify the impact of running the management control mechanism on the host OS while executing a kernel build on the guest OS, we measured the processing time of the build. The kernel used for the build was Linux 3.6.11, and the number of parallel jobs was set to one. A configuration file was generated using `make allnoconfig`, and the build was executed with this configuration. The `time` command was used to measure the user, sys, and real times. To eliminate the effect of changes in the number of managed processes or managed files on the processing time, the registered information was set to unrelated processes and files that were not involved in the kernel build.

The kernel build processing time was measured on the guest OS in the *baseline*, *tracing*, and *management* environments. The results are shown in Table 3. The processing time in the tracing and management environments was longer than that in the baseline environment, and the sys time in particular increased by approximately 70%. Most of this increase is considered to be caused by the overhead of hooking system calls and determining whether they are involved in the diffusion of classified information in the tracing function. The processing times in the tracing and management control environments were almost the same.

5.4 Evaluation of the Impact of Executing Management Control During Kernel Build

Next, we examined the effect of executing the management control operations themselves during the kernel build. While executing the kernel build on the guest OS, we simultaneously executed the management control operations on the host OS and measured the processing time. In addition, to investigate the impact when the processing load of the management control mechanism increases, we measured the build processing time while increasing the number of executions of each management control operation (listing, addition, and removal) to 1, 10, and 100 times.

The overhead of executing management control operations using `/proc/processlist` is shown in Figure 8, and the overhead when using `/proc/filelist` is shown in Figure 9. The overhead values shown in the figures represent the increase in processing time compared to the kernel build time in the baseline environment (Table 3). The vertical bars represent the overhead when the number of executions of each management control operation was increased to 1, 10, and 100 times, and the solid line represents the baseline overhead when no management control operations were executed, which corresponds to the time increase from the baseline to the management control environment (user: 6.20 s, sys: 21.67 s, real: 28.73 s).

As shown in Figures 8 and 9, in the list output operation, the kernel build processing time increased as the number of executions increased. Executing `/proc/processlist` 100 times increased the real time by 44.0 s, and executing `/proc/filelist` 100 times increased the real time by 75.1 s. This is because, under the environment where the number of physical cores was limited to one, the list output operations on the host OS occupied the CPU, causing interruptions and waiting in the kernel build processing on the guest OS. In contrast, the

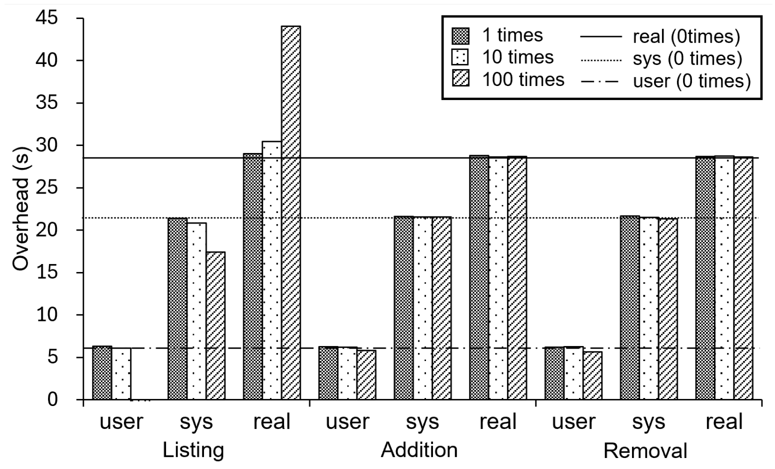


Figure 8: Overhead of executing management control operations using `/proc/processlist`

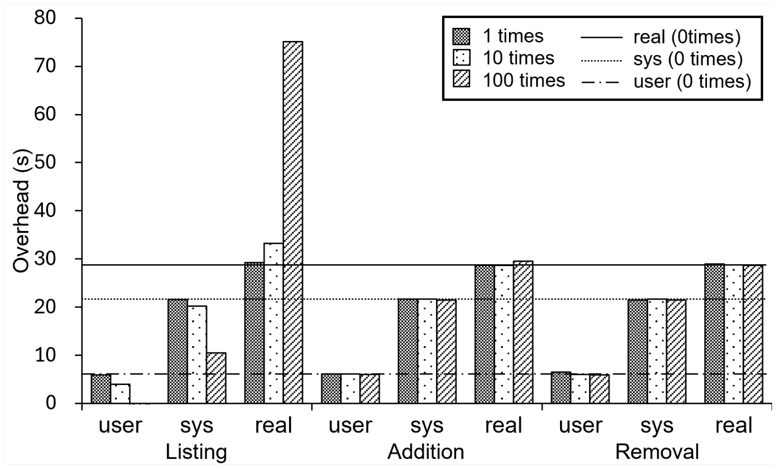


Figure 9: Overhead of executing management control operations using `/proc/filelist`

addition and removal operations on each virtual file occupy the CPU for only a short time and thus had little impact on the kernel build processing, resulting in almost the same processing time as that in the tracing environment.

We also evaluated the impact of repeatedly performing management operations during the kernel build workload. While adding or removing managed processes and managed files did not significantly affect the build time, repeatedly reading `/proc/processlist` or `/proc/filelist` during the build slightly increased the elapsed time when executed very frequently. For example, reading `/proc/processlist` one hundred times during the build increased the real time by approximately 44 seconds, and reading `/proc/filelist` one hundred times increased it by approximately 75 seconds. This is because frequent read operations from the host side consume CPU resources and cause the guest OS processes to wait. However, in typical operation, users are unlikely to perform such frequent read operations, so the effect is expected to be negligible.

These results demonstrate that the tracing function itself introduces measurable overhead due to the interception and analysis of system calls, as reflected in the increase in system call latency and sys time. However, adding the management mechanism on top of the tracing function does not further increase the overhead. Although excessively frequent reads of `/proc` files can delay guest OS processing by consuming host CPU resources, such usage is unrealistic in practice. Therefore, we conclude that the proposed management mechanism has little impact on the processing performance of the guest OS and can be introduced without degrading the performance of user workloads.

6 Related Work

Virtualization technology has been widely utilized for system monitoring and security enhancement [3, 4]. Studies in this field are generally categorized into two approaches: in-VM and out-of-VM. In-VM approaches introduce monitoring mechanisms inside guest OSs, enabling direct observation but requiring kernel modifications or agent installation. Sharif et al. [3] proposed a secure in-VM monitoring method using hardware virtualization, and SecPod [5] improved performance by leveraging nested page tables; however, both approaches required modifications to the guest OS.

In contrast, out-of-VM approaches monitor virtual machines from the hypervisor side without modifying the guest OS. Nitro [4] proposed an efficient method for tracing system call behaviors by selectively retrieving register values to reduce overhead. However, Nitro monitors all system calls occurring within the guest OS, which results in high monitoring cost and lacks the capability to restrict the scope of monitoring to specific processes or files. In our previous study [2], we proposed the function for tracing the diffusion of classified information on KVM, which similarly enables information-flow monitoring without modifying the guest OS. While Nitro focuses on general system-call-level behavior tracing, our previous work targets a specific information-flow domain—the diffusion of classified information—and features restricted monitoring scope and management of tracing results. However, that implementation managed monitored targets statically, requiring log reconstruction whenever the management target list was updated.

As a subset of out-of-VM approaches, many studies have proposed monitoring mechanisms based on Virtual Machine Introspection (VMI). These methods allow direct observation of guest OS behavior from the hypervisor side but are known to suffer from performance degradation due to monitoring overhead. Hizver et al. [6] and Shi et al. [7] reduced monitoring load through periodic sampling; however, their methods risk missing transient events occurring between sampling intervals. Taubmann et al. [8] improved efficiency by filtering unnecessary events. Although these studies share the same goal of reducing monitoring overhead in VMI-based systems and align with the lightweight design direction of this work, they do not provide mechanisms for dynamically controlling the monitoring scope. In contrast, our study enables flexible runtime addition and removal of monitored targets through a procfs-based management mechanism, providing a more practical and operationally efficient configuration compared to existing VMI-based methods.

Dangl et al. proposed VMIFresh [9], which redesigns the cache architecture of LibVMI to balance access efficiency and data freshness. Furthermore, Schwarz and Rossow introduced 00SEVen [10], which realizes VMI under AMD SEV-SNP environments by employing privileged in-VM agents in Confidential Virtual Machine (CVM) environments. While these methods improve the flexibility of VMI, they involve detailed memory access analysis and complex event translation, leading to significant runtime overhead. In contrast, our study focuses on a function for tracing the diffusion of classified information on KVM and introduces a procfs-based management mechanism enabling dynamic control of monitoring targets. By limiting the monitoring scope, our approach achieves lower runtime overhead and higher operational efficiency compared to general-purpose LibVMI-based frameworks.

In addition, recent studies have aimed to extend the applicability of VMI to cloud and IoT environments. He and Li proposed MDCRV [11], which applies VMI-based monitoring to containerized workloads for malware detection, demonstrating that VMI remains practically applicable in modern infrastructures. Building on these directions, our study provides a lightweight and dynamically controllable tracing mechanism for KVM-based environments.

7 Conclusion

This study designed and implemented a management control mechanism for the function for tracing the diffusion of classified information on a VMM. The mechanism enables both listing and dynamic addition or removal of managed targets, and its performance was evaluated in detail.

In conventional log aggregation approaches, obtaining the list of managed targets at an arbitrary point in time required scanning a large number of log entries, which resulted in long processing times. To address this problem, we implemented `/proc/processlist` and `/proc/filelist` using the procfs, enabling the system to directly refer to the management information stored in the kernel module and instantly list the managed targets at any given time. In addition, to solve the problem that users could not dynamically control the managed targets during execution, we introduced a write interface through procfs, allowing users to dynamically add and remove managed processes and managed files at runtime. By integrating this mechanism with the existing listing method, we significantly improved the operational manageability of the tracing function.

Because the proposed management mechanism operates on kernel information from user space, it could potentially affect system performance through operation overhead. Therefore, we conducted evaluations to clarify this impact. We measured the processing time of the proposed mechanisms using the rdtscll instruction, and the results showed that add and remove operations took approximately 4 µs per item, and listing 100 target processes took approximately 106 µs. Although the processing time for listing is relatively longer compared to add or remove operations, the listing of managed targets is performed only occasionally at the user's discretion. Therefore, its impact on the diffusion tracing function is considered negligible. Furthermore, we evaluated their impact on the processing performance of the guest OS using LMbench and a kernel build workload, and found that introducing the management control mechanism caused almost no additional overhead compared to using only the tracing function. This confirmed that the impact on the performance of the guest OS is small.

These findings demonstrate that this study successfully established a non-intrusive and lightweight tracing platform that enables flexible control of managed targets.

Future work will include extending the diffusion tracing function to newer Linux kernels, evaluating scalability under larger workloads and multi-core or multi-VM environments, enhancing system robustness through access control for the procfs interface, performing comparative evaluations with VMI-based monitoring frameworks, and integrating real-time visualization and alert mechanisms to improve usability.

Acknowledgments

This work was partially supported by JSPS KAKENHI Grant Numbers JP23K24848 and JP25K03119.

References

[1] Shota Fujii, Masaya Sato, Toshihiro Yamauchi, and Hideo Taniguchi. Evaluation and design of function for tracing diffusion of classified information for file operations with kvm. *The Journal of Supercomputing*, 72(5):1841–1861, February 2016.

[2] Hideaki Moriyama, Toshihiro Yamauchi, Masaya Sato, and Hideo Taniguchi. Improvement and evaluation of a function for tracing the diffusion of classified information on KVM. *Journal of Internet Services and Information Security (JISIS)*, 12(1):26–43, February 2022.

[3] Monirul I. Sharif, Wenke Lee, Weidong Cui, and Andrea Lanzi. Secure in-VM monitoring using hardware virtualization. In *Proc. of the 16th ACM conference on computer and communications security*, pages 477–487. ACM, November 2009.

[4] Jonas Pfoh, Christian Schneider, and Claudia Eckert. Nitro: Hardware-based system call tracing for virtual machines. In *Advances in Information and Computer Security. Lecture Notes in Computer Science*, volume 7038, pages 96–112. Springer, November 2011.

[5] Xiaoguang Wang, Yue Chen, Zhi Wang, Yong Qi, and Yajin Zhou. SecPod: a framework for virtualization-based security systems. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 347–360, Santa Clara, CA, July 2015. USENIX Association.

[6] Jennia Hizver and Tzi-cker Chiueh. Real-time deep virtual machine introspection and its applications. In *Proc. of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, volume 49, pages 3–14. ACM, July 2014.

[7] Jiangyong Shi, Yuexiang Yang, and Chuan Tang. Hardware assisted hypervisor introspection. *SpringerPlus*, 5(1):647, May 2016.

[8] Benjamin Taubmann and Hans P. Reiser. Towards hypervisor support for enhancing the performance of virtual machine introspection. In *IFIP International Conference on Distributed Applications and Interoperable Systems. Lecture Notes in Computer Science*, volume 12135, pages 41–54. Springer, June 2020.

[9] Thomas Dangl, Stewart Sentanoe, and Hans P. Reiser. VMIFresh: Efficient and fresh caches for virtual machine introspection. *Computers & Security*, 135:103527, 2023.

[10] Fabian Schwarz and Christian Rossow. 00SEVen – re-enabling virtual machine forensics: Introspecting confidential VMs using privileged in-VM agents. In *Proceedings of the 33rd USENIX Security Symposium*. USENIX Association, 2024.

[11] Xinfeng He and Riyang Li. Malware detection for container runtime based on virtual machine introspection. *The Journal of Supercomputing*, 80(6):7245–7268, November 2023.