International Journal of Networking and Computing – www.ijnc.org, ISSN 2185-2847
Volume 13, Number 2, pages 273–286, July 2023

Memory Analysis Based Estimation of Hook Point by Virtual Machine Monitor

Masaya Sato[†1], Taku Omori[†2]
†1 Faculty of Computer Science and Systems Engineering,
†2 Graduate School of Computer Science and Systems Engineering,
Okayama Prefectural University
Soja, Okayama, 719-1197, Japan

Toshihiro Yamauchi[†3], Hideo Taniguchi[†4]
†3 Faculty of Environmental, Life, Natural Science and Technology,
†4 Graduate School of Environmental, Life, Natural Science and Technology,
Okayama University
Okayama, Okayama, 700-8530, Japan

Received: February 15, 2023
Revised: May 5, 2023
Accepted: June 1, 2023
Communicated by Toru Nakanishi

Abstract

The behavior of virtual machine (VM) programs are monitored by virtual machine monitors (VMMs) for security purposes. System calls are frequently used as a monitoring point. To monitor the system calls, the VMM inserts a breakpoint, called a hook point, into the memory of the monitored VM. The hook points are determined based on experimental knowledge. However, reading the source codes of operating systems (OSes) requires specialized knowledge. In addition, the appropriate hook point differs among OSes and OS versions. Analyzing the source code in each OS update is impractical. Searching for the appropriate hook point for various OSes is also difficult. To address these problems, we propose a method for estimating the hook point using a memory analysis technique. The proposed method acquires the memory of the monitored VM and then searches for an appropriate instruction appropriate to hook. The search instructions depend on the processor architecture. In addition, we also proposed a method for searching the appropriate instruction using a single step execution. This version reduces the cost for searching the instructions and improve robustness for various Linux versions. The experimental results showed that the proposed method precisely estimates the hook point for various OS versions and OSes. In addition, the overhead of the proposed method is small, considering the boot time of the monitored VM.

Keywords: System Call, Virtual Machine Monitor, Operating System

1 Introduction

A virtual machine monitor (VMM) is used for the security monitoring of virtual machines (VMs). A VMM has the privilege of accessing the memory used for the VM, as well as intermediate hardware access. Therefore, VMMs are suitable for the security monitoring of VMs. Analysis of malicious

software (i.e., malware) is a representative application of VMs. To analyze malware, separating infected and analysis environments is important because sophisticated malware can attack the analysis environment [7]. Because VMs are separated from each other, malware infection of a VM does not affect on other VMs or VMMs. Therefore, VMs are suitable for malware analysis. Security monitoring is also used to detect and prevent malicious activities on VMs [5, 2]. VMs are used as infrastructure for cloud computing; therefore, detecting and preventing malicious behavior from outside of the VMs is required. Security monitoring using a VMM enables us to passively or actively monitor the behavior of programs from outside of the VM.

System call monitoring is a representative method of actively monitoring VMs [3, 12, 6, 9, 14]. We can collect valuable information from system calls. For example, sequences of system calls are used to detect malicious processes acting on VMs [6, 9, 14]. The use of a breakpoint is a method for monitoring system calls using a VMM. There are two types of breakpoints: software- and hardware-based. In this study, we focused on hardware breakpoints. To set a hardware breakpoint, the VMM must store the address of the VM in a debug register. By setting the breakpoint, the VMM can monitor the system call invocation by capturing the debug exception that occurs at the address. This point is called the hook point. The hook point must be set to an appropriate address such that valuable information may be collected from the VM. Therefore, determining the address for the hook point requires specialized knowledge.

To monitor a system call, a hook point is set in the Linux kernel. One study used a system call entry point for monitoring [9]. However, this is insufficient for acquiring process information because the stack is not switched to the kernel stack at the point. A previous study [10] analyzed the source code of an operating system (OS) to search for appropriate instructions for the breakpoint. There are two methods for determining the hook points. The first method involves setting a hook point with a static address. The other method involves setting a hook point with an offset from the entry point of the system call. However, these methods have two problems: frequent updates of the Linux kernel and address space layout randomization. The Linux kernel is updated frequently (almost every day); therefore, analyzing Linux kernels with every update is unrealistic. Moreover, recent Linux kernels have a security feature called kernel address space layout randomization (KASLR) [1]. KASLR randomizes the starting address of the Linux kernel for each boot. This means that the first method for setting the hook point is useless. Therefore, automation to determine a hook point is a challenging problem.

In this study, we propose a method to estimate a hook point and automatically set a breakpoint to the estimated address for system call detection. First, the VMM identifies the system call entry point by observing the value of the `IA32_LSTAR` model specific register (MSR) on the target VM. The VMM then copies the memory portion from the system call entry point and searches for machine codes suited for a hook point. Finally, the VMM stores the found address in a debug register to set a hardware breakpoint. We surveyed past Linux kernel versions and found that only three types of machine code are appropriate for the hook point. Therefore, the proposed method can precisely set the hook points in various Linux kernel versions without source code analysis. The evaluation results showed that the proposed method precisely sets the hook point in three versions of the Linux kernel, even with KASLR. In addition, the performance overhead incurred by the proposed method is confirmed to be negligible in practical environments.

In addition to the above study, we extended the proposed method to reduce the cost to analyze the source codes. The searching part of the proposed method consists of two phases. The first phase is for searching an instruction for stack switching (the first target instruction). The second phase is for searching the instruction following the first target instruction (the second target instruction). The VMM estimates the second target as a hook point in the proposed method. However, the searching phase is based on the heuristics what an instruction following the stack switching. The first target is determined based on the structure of operating system but the policy for searching the second target is based on the results of our survey on various versions of Linux's source coeds. This means the proposed method lacks robustness for the future Linux versions and other operating systems. To address this problem, we employed a single step execution for searching the target instruction following stack switching. In this paper, we detail the design and experimental results of the extended version of the proposed method.

The contributions made in this paper are as follows:

1. We proposed a method estimate a hook point for a system call monitoring by a VMM. The hook point is determined by developers' knowledge in related work. Our proposal helps developers determining the hook point because the cost for analyzing source codes and memory content is reduced. We proposed two methods for estimating the hook point: memory-analysis based and single-step based. The memory-analysis based method can set the hook point based on the knowledge of instructions used on the VM. This helps developers setting breakpoints in accordance with their purposes. The single-step based method reduces the cost for analyze the source codes and memory content of an OS on the VM. The single step mode is more versatile on various OS versions and OSes.

2. We evaluated the availability of the proposed method and performance overhead. The experimental results show that the proposed method is available on almost all versions of Linux on x86-64 CPUs. Measurement results show the performance overhead of the proposed method caused once at a boot time and small enough.

2 Related Work

2.1 Security Monitoring of Virtual Machines

VMMs are used for security monitoring of VMs [16, 5, 2, 3, 12, 6, 9, 14, 7]. VM is used for malware analysis [3, 7], VM introspection [5, 12, 9, 11, 2], and VM protection [16, 17]. There is various targets for monitoring including memory, file, and behavior of malicious processes. However, almost all events are triggered by system call on the VM, thus, system call has an important role on all of the above purposes. Even though the system call monitoring causes large performance overhead, transparent monitoring is required. Inserting agents into VMs are efficient, and easy to deploy. However, the agent-based approach is prone to attack and easily detected by malware or attackers. For this reason, the system-call based approach is commonly used for transparency and security of the monitoring mechanism.

2.2 System Call Detection using Breakpoints

A hardware breakpoint using debug registers is a method used to detect the invocation of system calls on a VM from a VMM. Although there are various host-based tools for collecting system call traces, including `strace` and `auditd`, we focus on system call detection from outside of the VM. Figure 1 shows how the VMM hooks the invocation of a system call on a VM using a hardware breakpoint. This method inserts a hardware breakpoint into a system call routine before the system call service routines (e.g., `open`, `read`, and `write`) are called. In addition, the VMM modifies the VM exit control field of data region of the virtual machine control structure to cause VM exit by a debug exception. A system call invocation on the VM causes a debug exception, and VM exit occurs in this situation. The VMM detects the system call invocation by capturing the VM exit.

The hook point varies for different purposes. Figure 2 shows an overview of system call execution in Linux. The system call routine can be divided into three parts: an entry point, a call for system call service routines, and an exit point. A hook at the entry point is suitable for collecting system call numbers and arguments. To collect granular information (e.g., process ID and files opened by the process), a hook at the call of the system call service routines or the exit is suitable [11]. Analyzing the process information requires the address of the kernel stack; therefore, hooking at the address after the instruction to switch the stack is required. In the `SYSENTER` instruction for 32-bit OSes, the stack is automatically switched to a kernel stack. The VMM can access process information without a workaround. However, in the x86-64 environment, `SYSCALL` does not switch the stack; therefore, the breakpoint must be set after the stack is switched. If the researcher needs to monitor changes in the process' data while executing the system call, monitoring the exit point is required. Anomalies can be detected by comparing the data between the entry and the exit points.

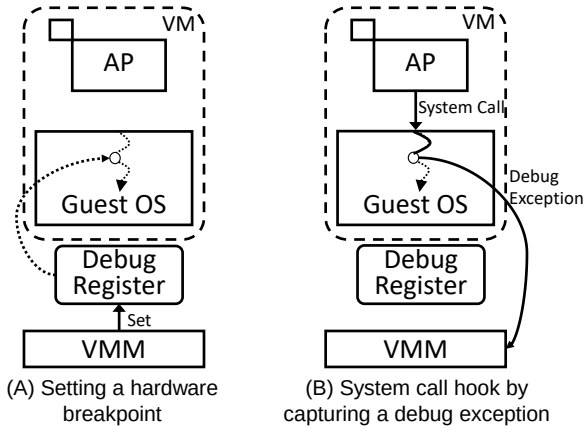

Figure 1: Hooking system call by the hardware breakpoint.

In each phase, the address for setting a breakpoint must be determined. Two methods have been used to determine the hook point. The first method statically determines the address of the hook point by analyzing the source code of the Linux kernel. However, the address of the hook point changes if the Linux kernel is updated and modified by users. In addition, randomization of the starting address of the Linux kernel using KASLR accordingly changes the address of the system call routine. Another method uses an offset from the entry point. Changes in Linux kernels cause changes of the address of the system call routines. However, the offset is not frequently changed in each Linux version. KASLR does not affect the offset, because randomization affects the starting address of the Linux kernel. In addition, the entry point is available through the register for the `SYSCALL` instruction. Therefore, using the offset to determine the address of the hook point is preferable.

2.3 Memory Forensics

Memory forensics, a technique for investigating artifacts of malware or malicious activities from memory images or live memory, is crucial in security monitoring and malware analysis [15, 13, 8]. Volatility [15] is a tool for extracting variables from a memory image. This tool is frequently used for malware analysis but symbol information for each OS version is required to extract variables from memory images. Autoprofile [8] addresses this problem by combining the memory image and source codes. Rekall's approach [13] is similar to Autoprofile. Memory forensics is basically conducted on variables managed by the kernel on the VM. Our aim is to analyze the instructions from the VM's memory thus the target for analysis is different from these researches. In addition, we aim to analyze the VM's memory not with the source codes but with the features of the processor's architecture.

Feng et al. proposed cross-version memory analysis for automatically constructing a profile for memory forensics [4]. Cross-versions memory analysis compares the memory of an old and a new version of a software. They focus on instruction accessing data structure required for constructing profiles for memory forensics. Those instructions' operands are statically determined at a compile time. Then, the operands of those instructions are abstracted in their profile. If the same instruction patterns are detected but the operands are different in the old and the new versions, the operands of the new versions are used for the new profile. The approach is similar to our proposal that analyzes and searches characteristic instructions from the VM's memory. However, their target is to find such characteristic instructions. In contrast, we aim to find instructions appropriate for hook point.

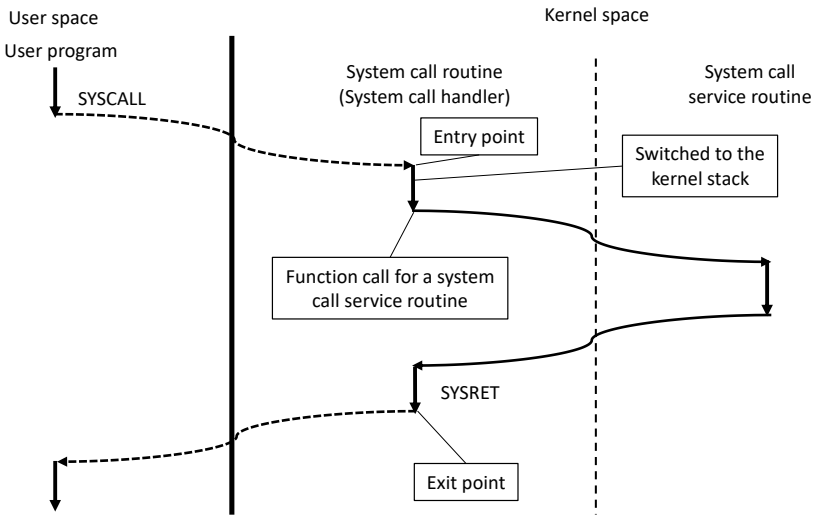

Figure 2: Overview of the system call execution flow in Linux.

3 Estimation of Hook Point

3.1 Challenges

There are two challenges encountered when determining the hook point:

1. Application to various OSes and OS versions
   The implementation of system call is different in various OSes. In addition, system call routines are changed in past Linux kernels, which means that the hook point may differ in future Linux versions. Adopting the system call detection for various OSes requires reading of each OS source code. Frequent updates for a Linux kernel make it difficult for developers to statically determine a hook point. Changes in Linux kernels caused by Linux distributions may change the address of the system call routine; therefore, the same problem can be observed.

2. Dynamic setting of a hook point to handle randomization by KASLR
   Current Linux kernels randomize address spaces. In this situation, statically determined hook points are useless because the starting address of the Linux kernel changes in each deployment to the memory at the boot time. KASLR randomizes the starting address only once at the boot time; therefore, the dynamic setting of a hook point solves the problem resulting from the application of KASLR.

The method for determining the hook point using the offset from the entry point addresses Challenge 2; however, Challenge 1 remains to be overcome. As the code for the system call routine changes, the offset from the entry point changes. In addition, the method using the offset requires the analysis of the Linux kernel. This analysis requires specialized knowledge about the structure of the OS and processor architecture; therefore, we also aim to reduce the cost of determining the hook point.

3.2 Assumed Environment

We assume that the system call is invoked through the `SYSCALL` instruction in x86-64 processors. The target operating system in our study was a 64-bit Linux system running on a VM. The VMs were constructed using the VT-x hardware virtualization extensions. System call information, including system call numbers, arguments, process information, and kernel information, must be collected by the VMM. This means that the VMM must hook the system call after the stack is switched to the kernel stack. We do not rely on a memory forensic framework like volatility framework because it

requires symbol information of the analysis target OS [15]. Relying on tools reduces the cost for analysis but dependency on other tools increases. Reducing such dependencies improves versatility for various OSes and processor architectures.

3.3 Overview

To estimate the hook point, our proposal conducted on the following procedures.

1. Identification of the entry point to the system call
2. Search of the first target instruction (stack switching)
3. Search of the second target instruction (estimation target)
   (a) Memory analysis based method
   (b) Single-step based method

To set the hook point into the system call, the VMM identifies the entry point at first. Then, the VMM copies the memory portion of the VM and searches for an instruction used for stack switching. To collect detail information of each process, process control block of the VM's process is required. A pointer to the process control block is stored in the kernel stack. Thus, we need to set the hook point after the stack is switched to the kernel stack. For this reason, the second target instruction is the instruction following the first target instruction. The second target is the estimation target on this research. We proposed two methods for the third procedure: the memory-analysis method and the single-step method.

3.4 Identification of the Entry Point to the System Call

The proposed method uses the value stored in the `IA32_LSTAR` MSR as the entry point for the system call. There are two methods for obtaining the entry point: calculating the static address by analyzing the symbol table or reading the value from the `IA32_LSTAR` MSR. We decided to employ the second method because the address in the symbol table is useless when KASLR is applied. By contrast, the `IA32_LSTAR` MSR contains the address after KASLR randomizes the starting point of the kernel. Consequently, the proposed method does not suffer from difficulties related to KASLR.

3.5 Memory Analysis

Figure 3 shows the flow for estimating the hook point using memory analysis. The details of each step are as follows:

1. A program on the target VM executes an instruction that triggers a VM exit. A VM exit occurs, and the VMM determines whether the VM exit is the first occurrence after a value is stored in `IA32_LSTAR` MSR.
2. The VMM obtains the address of the system call entry point from the `IA32_LSTAR` MSR.
3. The VMM copies the memory from the VM. The starting address for copying is the entry point. The copying unit size is determined in a later experiment.
4. The VMM searches for the target machine codes from the copied memory. If any target machine code is found in the copied memory, the VMM stops copying and moves to the next part. If no target machine code is found, the VMM repeats the processing from Step 3).
5. The VMM sets the found address as the hook point, and then returns the processing to the target VM.

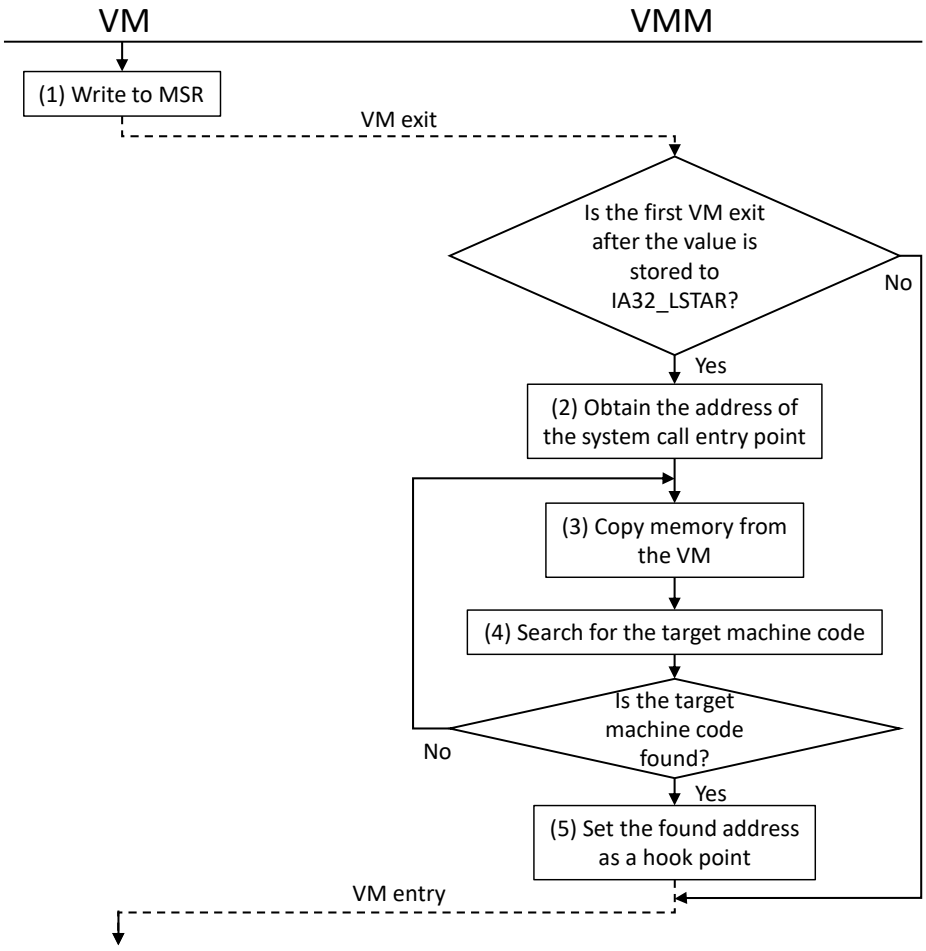

Figure 3: Flow for determining the target machine code from the VM's memory.

The target machine code in Step (4) is determined using our survey. We surveyed Linux versions from 2.6.12 to 5.15.1 to determine what machine code is suitable for setting the hook point. Linux 2.6.12 is the oldest, and 5.15.1 is the latest version available on the GitHub repository for our survey. The search for the target machine code consists of two phases. The first phase is to search for the machine code used for switching the stack. This phase involves preparation for searching for the hook points. The second phase searches for the next machine code that is commonly used in various Linux versions.

The survey policy is as follows:

- The target machine code appears after the machine code switches from the user stack to the kernel stack.
- The target machine code commonly appears in various kernel versions.
- The target machine code appears only once in the area from the entry point to the starting address of the system call service routine.

In our survey, we found that system call flow changed twice in the surveyed Linux version. Therefore, there were three system-call patterns. Considering the results of our survey, we decided to employ the target machine code for the first phase as `movq` which stores an address to the stack pointer for all versions and for the second phase as `movq` for Linux 2.6.12–4.0 and `pushq $__USER_DS` for 4.1–5.15.1. In summary, the proposed method estimates the hook point by searching `movq` and

`pushq $__USER_DS` from the memory copied by the target VM. Therefore, the proposed method is applicable to various Linux versions.

3.6 Searching the instruction following stack switching

Memory analysis method in Section 3.5 is applicable to various Linux versions, however, the found instruction will change in future Linux versions. In addition, these instructions are differed in other OSes. Thus, the method in Section 3.5 requires continuous analysis of source codes and memory in future Linux versions.

The main reason of the above problems is that the instruction length of x86 processors is not always the same in each instruction. Thus, the offset from the first to the second target instruction is not always the same. The first target instruction is versatile to various OSes and OS versions because the first target instruction is determined based on the characteristics of the processor's architecture. However, the second target instruction is not always the same in future OS versions. If the second target instruction changed in future versions, we need to analyze the source code of those versions. In addition, the instruction is differed in each OS. This degrades the robustness of the proposed method to future Linux versions and other OSes. For example, in NetBSD and OpenBSD, other instructions are used. Thus, the proposed method is not applicable to those OSes.

To overcome this problem, we extended the proposed method using a single step execution (single-step method). The single step execution is a function of processors. If the single step execution is turned on, the execution of each instruction causes an exception. The VMM can catch the exception with the memory address causing the exception. For this reason, the VMM can acquire the memory address of an instruction following the first target instruction.

Additionally, we employed a monitor trap flag to improve the transparency of the proposed method. Basically, the single step execution is available through a trap flag. However, the trap flag is visible to the guest OSes through register access. Malicious programs change their behavior if they found themselves monitored by the VMM. In addition, the monitoring through the trap flag will be disabled by modifying the register value by malware on the VM. Thus, using a trap flag degrades the transparency and security of the monitoring. In contrast, the monitor trap flag and its exception are not visible to the guest OS but visible to the VMM. Therefore, we used the monitor trap flag to implement the single-step method. Note that the monitor trap flag is a function implemented in x86-64 processor; therefore, the single-step method is depending on x86 processors.

Figure 4 shows the flow of the single-step method using a monitor trap flag. The detection of the first target instruction is the same as the normal method. After detected the first target instruction, the VMM sets the breakpoint into the found address and detects the execution of that instruction. Then, the VMM sets the monitor trap flag and returns processing to the VM. The execution of the next instruction on the VM causes a VM exit and the VMM gets the address of the following instruction. Finally, the VMM sets the current address as the hook point and returns the processing to the VM. This flow determines the hook point without knowledge of the instructions used in the VM. Thus, the single-step method can set the breakpoint without knowing what an instruction is following the stack switching. Memory analysis is required only once to find the first target instruction (stack switching).

Additionally, the flow in Figure 4 is executed only once when the first system call is invoked on the VM. Moreover, the additional flow does not require memory copy. Thus, the overhead caused by the single-step method is estimated as twice the time for a VM exit (almost one microsecond or less).

3.7 Discussion

3.7.1 Comparison of methods for detecting the second target

We proposed two methods for detecting the second target address in Sections 3.5 and 3.6. Table 1 shows strengths and weaknesses of the methods. As shown in the table, the memory-analysis method is versatile from the viewpoint of processors architectures because it does not rely on a feature of single-stepping. However, the memory-analysis method is prone to detect a wrong instruction as a

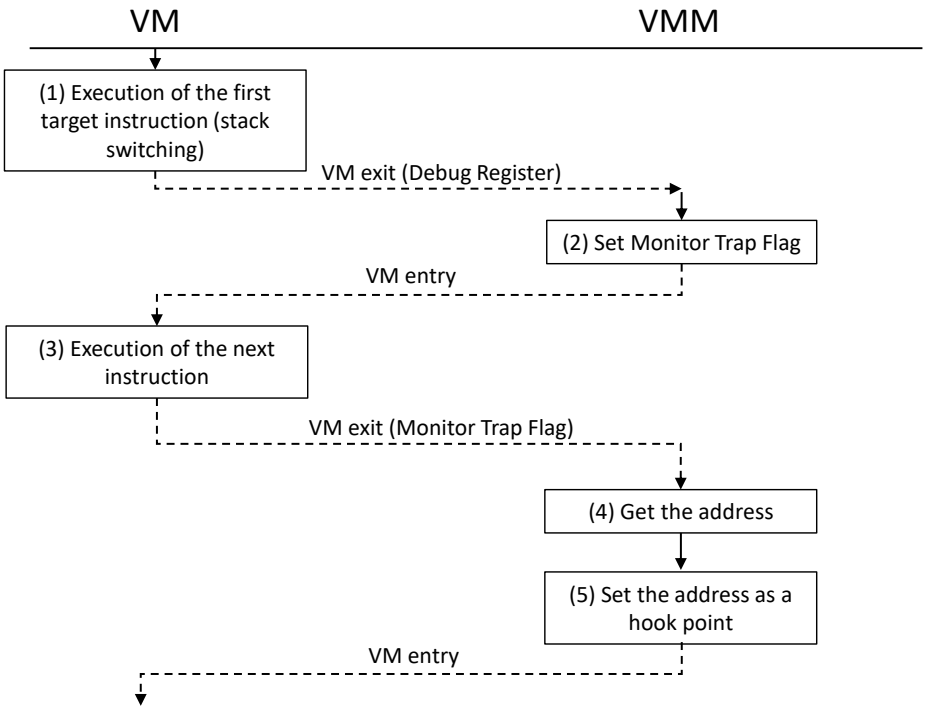

Figure 4: Flow of finding the address of the second target instruction using monitor trap flag.

Table 1: Comparison of two methods for detecting the second target

	Memory-analysis method	Single-step method
Strengths	(1) Applicable without a single-stepping feature of processors (2) Smaller performance overhead	(1) Effective when stack switching is detected
Weaknesses	(1) Requires prior knowledge for the instruction following the stack switching (2) Possibilities of false positives and false negatives are greater than the single-step method	(1) Requires a single-stepping feature of processors (2) Single-stepping causes an additional performance overhead

hook point (detailed in Section 3.7.2). In contrast, the single-step method is more resistant to this problem. The performance overhead of the single-step method is greater than the memory-analysis method.

To compare the performance overhead, we conducted an experiment to measure the time for estimating a hook point. The experimental result show that the memory-analysis method is faster than the single-step method and the performance overhead caused by the single-stepping is about 6 microseconds. For better performance, the memory-analysis method exceeds the single-step method; however, the overhead is negligible because the hook point estimation works only once at a boot time. In contrast, false positives are more problematic and should be reduced.

For this reason, the single-step method is superior to the memory analysis method if the single-stepping feature is available on the processor. If the single-stepping feature is not available, the memory-analysis method still remains as a method to estimate a hook point.

Table 2: Instructions used for stack switching.

OS	Instructions for stack switching
FreeBSD 13.2-RELEASE	`movq PCPU(RSP0), %rsp`
NetBSD 9.3-RELEASE	`leaq SP(0), %rsp`
OpenBSD 7.3	`xchgq %rax, %rsp`

3.7.2 False Positives and False Negatives

In our study, a false positive is associated with wrongly detecting an instruction unrelated to stack switching as a hook point. We found that the first target machine code is `movq` which stores an address to the stack pointer and the second target machine codes were `movq` and `pushq $__USER_DS`.

For the first target, `movq` is common but accessing the stack pointer is limited. Thus, a false positive on the first target is very limited. Additionally, the basic concept behind the proposed method is effective. Although Linux employs other instructions to switch the stack, we can specify the target code by gradually narrowing the range for detection.

For the second target, `movq` and `pushq` are common and are used in other operations. In our survey of various Linux versions, detecting the above codes resulted in no false positive on current Linux versions. However, there is no guarantee that the proposed method will be effective for the future Linux versions. Therefore, we need to keep monitoring of newer Linux versions to utilize the memory-analysis based method.

A false negative in our study indicates that the proposed method cannot detect any hook point. This case is rare because `movq` and `pushq` are common instructions. We believe that this situation will not change until the processor employs other instruction sets. Changes in the instructions used for switching stacks affect the effectiveness of the proposed method. However, the proposed method can be used to determine the instructions used for stack switching.

3.7.3 Applicability to Other OSes

The proposed method utilizes the characteristics of the processor' architecture. An OS kernel must switch the stack from a user stack to a kernel stack to isolate its execution from the user space. Because this design is common to various OSes, the proposed method is applicable to other OSes. From the above viewpoint, the proposed method is applicable when an OS uses a specific instruction for stack switching.

We surveyed FreeBSD, NetBSD, and OpenBSD to confirm applicability of the proposed method. Table 2 shows OSes and instructions used for stack switching in system call handlers. As shown in the table, different OSes employ different instructions for stack switching. Only the instruction of FreeBSD is the same as Linux, and NetBSD and OpenBSD use different instructions for stack switching. Consequently, our proposal can be applied to OSes other than Linux and FreeBSD by extending the target machine codes to support other instructions accessing the stack pointer.

3.7.4 Different Architectures

It is possible to extend the proposed method to other processor architectures (e.g., ARM and RISC-V). The proposed method can be applied to other architectures (1) if the entry point of system call can be detected by a VMM and (2) if the architecture requires an OS kernel to switch the stack when the running mode of a program changes from a user mode to a kernel mode. If the processors automatically switches the stack, there is no need for using the proposed method.

To explore this, we surveyed other processor architectures. Table 3 summarizes instruction for system calls, entry point of system calls, and necessity for stack switching. Interrupt Vector Tabel (IVT) and MSR can be detected by a VMM through registers; thus, the condition (1) is fulfilled in all architectures. However, architectures other than RISC-V and x86-64 automatically switches the stack; thus, the condition (2) is fulfilled only on RISC-V and x86-64. Consequently, we found our proposal is applicable to RISC-V processors other than x86-64.

Table 3: Processor architectures and applicability of the proposed method. Our proposal can be applied when both (1) and (2) are Yes.

Architecture	Instruction	(1) A VMM can detect the entry point of syscall?	(2) Is a kernel responsible for stack switching?	Is our proposal applicable?
Arm	`swi`	Yes (IVT[1])	No	No
Arm64	`svc`	Yes (IVT)	No	No
MIPS	`syscall`	Yes (IVT)	No	No
RISC-V	`ecall`	Yes (IVT)	Yes	Yes
x86-32	`sysenter`	Yes (MSR[2])	No	No
x86-64	`syscall`	Yes (MSR)	Yes	Yes

Table 4: Environment for evaluation.

CPU	Intel Core i7-6700 (3.40 GHz, 4 cores)
VCPU	Domain-0: 1
	Measurement VM: 1
Memory	Domain-0: 8 GB
	Measurement VM: 4 GB

Table 5: Experimental results for detecting invocation of a system call on various Linux versions.

Linux version	Detected? (Offset)	Detected with KASLR?	Detected by the single-step method?
3.2	✓(85)	✓	✓
4.19.18	✓(37)	✓	✓
5.15.1	✓(41)	✓	✓

On the other hand, other processor architectures are inappropriate to utilize our proposals. For example, the x86-32 architecture automatically switches the stack from a user stack to a kernel stack. In this case, the proposed method does not play a significant role.

4 Evaluation

4.1 Purpose and Evaluation Environment

We evaluated the applicability and performance overhead of the proposed method. To verify the applicability of the proposed method to various versions of Linux, we estimated a hook point for three versions of Linux. In addition, we evaluated the applicability of the single-step method to the Linux versions. Furthermore, we evaluated the performance impact of the proposed method on the boot time.

Table 4 shows the evaluation environment. We used Xen 4.13.0 as a VMM and Linux 3.2, 4.19.18, and 5.15.1 as guest OSes. Through our analysis, we found that there were only three types of system call routines. Therefore, we applied the proposed method to these three Linux kernel versions with different system call routines. Performance evaluation was conducted with Linux 4.19.18 as a guest OS on the measurement VM.

4.2 Applicability to Various Linux Versions

We conducted experiments to verify the effectiveness of the proposed method using various Linux versions. We also tested the applicability of the proposed method using KASLR.

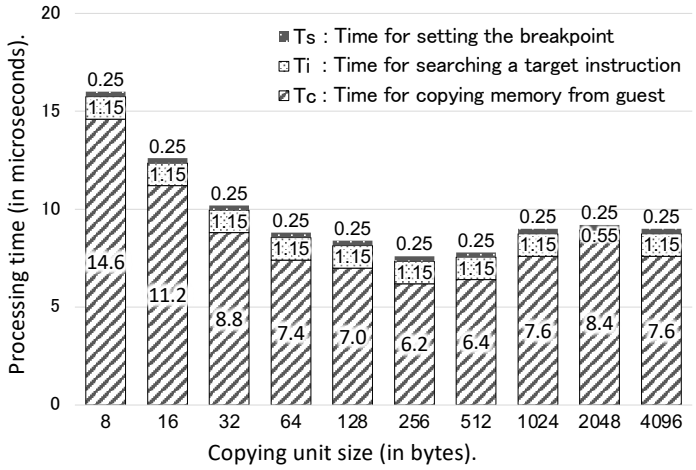

Figure 5: Overhead of the proposed method.

Table 5 presents the experimental results. We conducted these experiments with Linux versions 3.2, 4.19.18, and 5.15.1. We surveyed Linux kernels from 2.6.12 to 5.15.1 and found that three types of system call routines are used. The three Linux kernels indicated above have different system call routines; therefore, we tested the proposed method using these kernels. As shown in the results, the proposed method correctly estimated the hook points and successfully detected invocations of a system call in each Linux version, even when KASLR was applied.

Table 5 lists the offset of the address from the entry point of the system call for each Linux version. The offset differs for each Linux version; however, the proposed method determines the target instruction precisely. In Linux 3.2, the machine code that was found is `movq`. In Linux 4.19.18 and 5.15.1, the machine code was `pushq $__USER_DS`. We confirmed that the above two patterns do not appear in the system call routine. Therefore, there were no false positives in these Linux versions. However, the instruction for searching may differ in future Linux versions. Therefore, we plan to continue this analysis for future Linux versions to make the proposed method applicable in future versions.

We also conducted the experiment to confirm the efficiency of the single-step method to the above Linux versions. The experimental results show the single-step mode is applicable to the Linux versions without knowing what an instruction is following the `movq`.

4.3 Performance Evaluation

We measured the time required to write a value to the MSR and evaluated the performance overhead incurred by the proposed method. To measure the time required to estimate the hook point, we measured the time required for writing a value to the MSR. The proposed method is triggered when a VM exit occurs for the first time during OS's boot time. In our study, the instruction that causes VM exit on the Linux kernels is `MSR_WRITE`. Therefore, we evaluated the time required to write a value to the MSR. We measured the time required for copying memory from the guest, searching for a target instruction, and setting the hook point. The other part of the proposed method does not require additional processing time for the unmodified Xen. To measure the performance overhead, the processing time originating from the unmodified Xen was excluded.

Figure 5 shows the measurement results. T_c, T_i, and T_s are the times for copying memory from the guest, searching for a target instruction, and setting the hook point, respectively. Because the required memory size to be copied cannot be determined beforehand, the proposed method repeats memory copies with some sizes. Therefore, we measured the overhead for various copying unit sizes.

We confirmed that the processing time for copying memory from the guest accounted for 80% to 91% of the overhead using the proposed method. The offset of the target instruction from the system call entry point is 37, and the number of copy changes ranges from five, three, two, and one

for copying unit sizes of 8, 16, 32, and 64, respectively. The results are similar when the copying unit size is greater than 64 bytes, because the copy occurs once. In this experiment, the overhead was the smallest when the copying unit size was 256 bytes.

In addition to the above experiment, we compared the overhead with the boot time. The proposed method was triggered only when a VM exit occurred for the first time. This implies that the performance overhead affects the boot time. In our experiment, the booting time of a VM with Linux 4.19.18 is approximately 3.75 s. Therefore, the performance overhead of the proposed method is negligible.

5 Conclusions

We proposed a method to estimate the hook point for system call detection using a virtual machine monitor. The proposed method helps developers and researchers determine the memory address suitable for the hook point. By copying the memory from the target VM and searching for the characteristic machine code, the VMM can automatically set a breakpoint. We determined the target codes based on the architectural characteristics of the investigated system; therefore, our approach can be applied to various versions of Linux.

In addition, we proposed a method using single step execution for determining the target code. The memory-analysis method searches for two codes for setting the hook point. However, the single-step method only requires one code (which is basically `movq` for stack switching) and the following codes is automatically determined by single step execution. This version still requires the knowledge of the codes but reduces the cost for analyzing the source codes and machine codes in future OS versions.

The evaluation results show that the proposed method is applicable to various Linux versions, even when KASLR is applied. The single-step method is also applicable to the Linux versions. Performance evaluations showed that the overhead of the proposed method is less than 16 microseconds. The evaluation result also shows the overhead is negligible compared to the VM's boot time.

The experimental results show that the basic concept of the proposed method will be applied for detecting other valuable instructions for developers and analysts. In this paper, we employed the proposed method to detect the hook point for system call detection. In our future work, we will extend the proposed method to other hook points, OSes, and architectures.

Acknowledgment

This work was partially supported by KAKENHI Grant Numbers JP19H04109 and JP22H03592.

References

[1] K. Cook. Kernel address space layout randomization. In *Linux Security Summit*, 2013.

[2] T. Dangl, B. Taubmann, and H.P. Reiser. Rapidvmi: Fast and multi-core aware active virtual machine introspection. In *The 16th International Conference on Availability, Reliability and Security 2021*, pages 1–10, 2021.

[3] A. Dinaburg, P. Royal, M. Sharif, and W. Lee. Ether: Malware analysis via hardware virtualization extensions. In *15th ACM Conference on Computer and Communications Security*, pages 51–62, 2008.

[4] Q. Feng, A. Prakash, M. Wang, C. Carmony, and H. Yin. Origen: Automatic extraction of offset-revealing instructions for cross-version memory analysis. In *Proceedings of the 11th ACM on Asia conference on computer and communications security*, pages 11–22, 2016.

[5] T. Garfinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. *Ndss 2003*, 3(2003):191–206, 2003.

[6] X. Jiang, X. Wang, and D. Xu. Stealthy malware detection and monitoring through vmm-based "out-of-the-box" semantic view reconstruction. *ACM Trans. Inf. Syst. Secur.*, 13(12, Issue 2):1–28, 2010.

[7] O. Or-Meir, N. Nissim, Y. Elovici, and L. Rokach. Dynamic malware analysis in the modern era—a state of the art survey. *ACM Computing Surveys (CSUR)*, 52(5):1–48, 2019.

[8] F. Pagani and D. Balzarotti. Autoprofile: Towards automated profile generation for memory analysis. *ACM Transactions on Privacy and Security*, 25(1):1–26, 2021.

[9] J. Pfoh, C. Schneider, and C. Eckert. Nitro: Hardware-based system call tracing for virtual machines. In *Proc. International Workshop on Security*, pages 96–112, 2011.

[10] M. Sato, H. Taniguchi, and R. Nakamura. Virtual machine monitor-based hiding method for access to debug registers. In *2020 Eighth International Symposium on Computing and Networking (CANDAR)*, pages 209–214, 2020.

[11] S. Sentano, B. Taubmann, and H.P. Reiser. Virtual machine introspection based ssh honeypot. In *Proc. 4th Workshop on Security in Highly Connected IT Systems (SHCIS '17)*, pages 13–18, 2017.

[12] M. Sharif, W. Lee, W. Cui, and A. Lanzi. Secure in-vm monitoring using hardware virtualization. In *Proc. 16th ACM Conference on Computer and Communications Security*, pages 477–487, 2009.

[13] A. Socała and M. Cohen. Automatic profile generation for live linux memory analysis. *Digital Investigation*, 16:S11–S24, 2016.

[14] J. Soni, N. Prabakar, and H. Upadhyay. Behavioral analysis of system call sequences using lstm seq-seq, cosine similarity and jaccard similarity for real-time anomaly detection. In *2019 International Conference on Computational Science and Computational Intelligence (CSCI)*, pages 214–219, 2019.

[15] volatilityfoundation. volatility. https://github.com/volatilityfoundation/volatility.

[16] J. Wang, M. Yu, B. Li, Z. Qi, and H. Guan. Hypervisor-based protection of sensitive files in a compromised system. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, SAC '12, page 1765–1770, New York, NY, USA, 2012. Association for Computing Machinery.

[17] D. Zhan, L. Ye, B. Fang, X. Du, and Z. Xu. Protecting critical files using target-based virtual machine introspection approach. *IEICE TRANSACTIONS on Information and Systems*, 100(10):2307–2318, 2017.