IEICE TRANS. INF. & SYST., VOL.E103–D, NO.7 JULY 2020

PAPER Special Section on Information and Communication System Security

Identification of Kernel Memory Corruption Using Kernel Memory Secret Observation Mechanism

Hiroki KUZUNO[†,††a)] ***and*** **Toshihiro YAMAUCHI**[††b)]**,** ***Members***

SUMMARY Countermeasures against attacks targeting an operating system are highly effective in preventing security compromises caused by kernel vulnerability. An adversary uses such attacks to overwrite credential information, thereby overcoming security features through arbitrary program execution. CPU features such as Supervisor Mode Access Prevention, Supervisor Mode Execution Prevention and the No eXecute bit facilitate access permission control and data execution in virtual memory. Additionally, Linux reduces actual attacks through kernel vulnerability affects via several protection methods including Kernel Address Space Layout Randomization, Control Flow Integrity, and Kernel Page Table Isolation. Although the combination of these methods can mitigate attacks as kernel vulnerability relies on the interaction between the user and the kernel modes, kernel virtual memory corruption can still occur (e.g., the eBPF vulnerability allows malicious memory overwriting only in the kernel mode). We present the Kernel Memory Observer (KMO), which has a secret observation mechanism to monitor kernel virtual memory. KMO is an alternative design for virtual memory can detect illegal data manipulation/writing in the kernel virtual memory. KMO determines kernel virtual memory corruption, inspects system call arguments, and forcibly unmaps the direct mapping area. An evaluation of KMO reveals that it can detect kernel virtual memory corruption that contains the defeating security feature through actual kernel vulnerabilities. In addition, the results indicate that the system call overhead latency ranges from 0.002 μs to 8.246 μs, and the web application benchmark ranges from 39.70 μs to 390.52 μs for each HTTP access, whereas KMO reduces these overheads by using tag-based Translation Lookaside Buffers.

key words: *memory corruption, virtual memory management, system security, operating system*

1. Introduction

Kernel vulnerabilities in operating systems (OSs) have become a significant security risk [1], [2]. There are 2,240 kernel vulnerabilities in the Linux OS that were listed on the Common Vulnerabilities and Exposures (CVE) database until June 2019 (Fig. 1) [3].

Adversaries can exploit the kernel through such vulnerabilities. Preventive countermeasures must be developed to mitigate their attack scenarios. The adversary uses kernel vulnerabilities to modify the credentials in the kernel virtual memory. Privilege escalation forces the OS to grant root privilege to a non-privileged user account.

Manuscript received August 30, 2019.
Manuscript revised December 12, 2019.
Manuscript publicized March 4, 2020.

[†]The author is with Intelligent Systems Laboratory, SECOM Co., Ltd., Shibuya-ku, Tokyo, 150–0001 Japan.
[††]The authors are with Graduate School of Natural Science and Technology, Okayama University, Okayama-shi, 700–8530 Japan.
a) E-mail: kuzuno@s.okayama-u.ac.jp
b) E-mail: yamauchi@cs.okayama-u.ac.jp

DOI: 10.1587/transinf.2019ICP0011

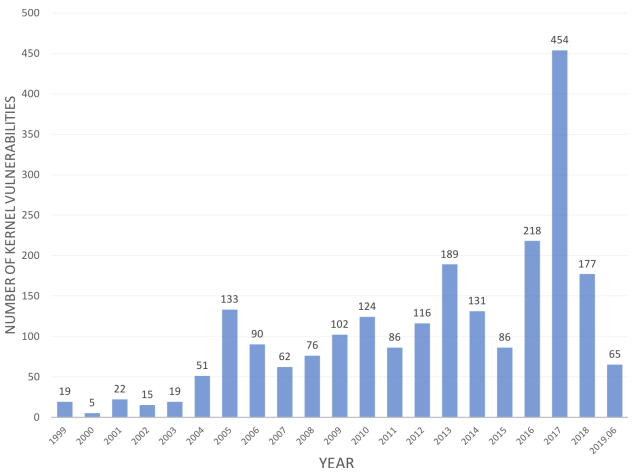

Fig. 1 Number of kernel vulnerabilities registered to CVE.

Full privilege is restricted by OS features, and the OS has two such features, namely capability [4] and Mandatory Access Control (MAC) mechanisms (e.g., SELinux [5]). The kernel has several countermeasures to prevent kernel vulnerability and data misuse. Kernel Address Space Layout Randomization (KASLR) disturbs the kernel functions and data position in the kernel virtual memory to conceal the identification of the virtual addresses of vulnerable functions [6]. Additionally, Control Flow Integrity (CFI) enforces kernel function flow validation between call and return relationships to prevent the injection of malicious code [7]. Return address monitoring of the stack is one method to detect kernel memory corruption and prevent malicious code initialization [8].

The CPU also has virtual memory access and an execution permission mechanism. The No eXecute bit (NX bit) is the execution permission flag for a virtual address in a page table entry [9]. Supervisor Mode Access Prevention (SMAP) and Supervisor Mode Execution Prevention (SMEP) are present in the CR4 register. SMAP prevents access to the user memory region, whereas SMEP prevents the code execution in the user memory region of the virtual memory at the supervisor [10]. Meltdown vulnerability uses the side channel approach to expose directory kernel functions and data virtual addresses. Therefore, Kernel Page Table Isolation (KPTI) has been proposed as a means of isolating the virtual address space between the user mode and the kernel modes in Linux [11].

Privilege restriction methods separate root privilege

features to minimize the damage to the OS environment in the event of a successful attack. Kernel and CPU countermeasures complicate the attack availability of kernel vulnerabilities based on the interaction between the user and kernel modes. However, these methods cannot prevent attacks that exploit kernel vulnerabilities in the kernel mode alone [12]–[15]. The adversary can avoid various security countermeasures by executing a kernel exploit code in the kernel mode to override the security feature functions in the kernel virtual memory (e.g., some kernel exploits disable SELinux via memory corruption [16], [17]).

In this study, we designed a novel security mechanism called "Kernel Memory Observer" (KMO) that can identify the illegal data manipulation of the kernel virtual memory, which could result in the security features being defeated. KMO provides a secret observation mechanism that equips an alternative kernel virtual memory as a secret virtual memory to monitor the original kernel virtual memory. Although the kernel has one virtual memory at the KPTI implementation, the design of KMO is such that the kernel virtual memory is completely separated to maintain its secrecy, and it is responsible for kernel monitoring code execution and valid data storing.

More specifically, KMO controls a virtual memory switching function that changes the kernel virtual memory space to the secret virtual memory space at various times during monitoring. KMO aims to prevent two scenarios: (i) malicious parameters at system call arguments that induce the injection of suspicious code targeting kernel vulnerabilities, (ii) a kernel vulnerability attack overwrites the kernel virtual memory, leading to a modification of KMO monitoring code and valid data. Therefore, KMO can successfully identify kernel virtual memory corruption.

In addition, KMO prevents the modification of the secret virtual memory and monitors valid data in the direct mapping region, which contains physical memory for effective allocation or collection. KMO forces the unmapping of the virtual memory region and relates KMO information from the direct mapping onto the kernel virtual memory.

We summarize the main contributions of this study below:

- We design a cutting edge security architecture, KMO, that provides a secret virtual memory to monitor the kernel virtual memory. KMO has a secret observation mechanism that provides three switching patterns between the secret virtual memory and the kernel virtual memory, whereas the unmapping method provides protection from direct mapping. Although kernel protection is being examined in multiple studies, no study has addressed the monitoring of kernel virtual memory at the kernel level. KMO provides the advantage of enhanced safety for the kernel, thereby combining the features of existing security mechanisms without virtualization. Moreover, it can be applied to the OS on a bare machine and to a guest OS on a cloud environment.
- We implement KMO on the latest Linux kernel with KPTI. For the evaluation, we examine KMO detection capability with regard to eBPF kernel vulnerability [15] and the prototype of illegal kernel modules that corrupt the kernel virtual memory to bypass the SELinux security feature. Additionally, KMO protects monitoring code and information from invalid access via the direct mapping region. The evaluation results for KMO revealed that its overhead is from 0.002 μs to 8.246 μs for each system call round time, whereas the application overhead is from 39.70 μs to 390.52 μs for each switching pattern for 100,000 HTTP accesses. KMO adopts the Process Context ID (PCID) of tag-based Translation Lookaside Buffers (TLBs) to mitigate these overheads and improve performance.

The preliminary version of this study appeared in [18]. We have added additional details on the background, discussion survey, and investigation along with evaluation results.

2. Background

2.1 Virtual Memory Management

Virtual memory is an illusion that provides domestic memory and memory isolation to running processes. It provides massive memory space on the kernel as compared to the physical memory space. Linux x86_64 allows each process to have its own virtual memory space. A multiple page table converts virtual addresses into physical addresses on the virtual memory space (Fig. 2). CR3 has the physical address of the page table of the current process, which refers to a traversal of the page table on the Memory Management Unit (MMU) and cache hit on the Translation Lookaside Buffer (TLB).

The virtual memory space layout and virtual address length differ for each CPU architecture. Linux x86_64 has a 48-bit virtual address length, which implies that the virtual address space has a size of 256 TB. The user space is 128 TB, and the kernel space is 128 TB, which contains the kernel function, data, module, direct mapping, and memory allocation address space (vmalloc, etc.).

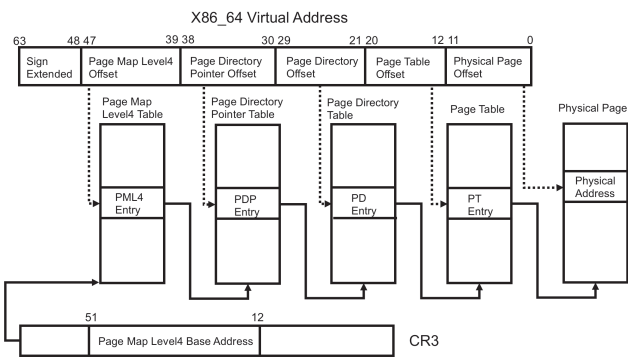

Fig. 2 Multiple page table converts virtual address into physical address.

Table 1 Number of categories in 128 memory corruption vulnerabilities (CVE registered) for Linux kernel.

DoS	Code Execution	Overflow	Bypass Feature	Gain Information	Gain Privileges
112	6	73	1	4	25

Table 2 PoC available Linux memory corruption vulnerability list since 2016. Types are referring to DoS: denial-of-service, Mem. Corr.: Memory Corruption, Priv: Gain Privileges.

CVE ID	Types	PoC	Publish Date	Description
CVE-2017-16995 [15]	DoS, Overflow, Mem. Corr.	✓	2017-12-27	A boundary check error in kernel/bpf/verifier.c
CVE-2017-1000112 [22]	Mem. Corr.	✓	2017-10-04	A race condition in net/ipv4/ip_output.c
CVE-2017-7533 [23]	DoS, Priv, Mem. Corr.	✓	2017-08-05	A race condition in the fsnotify implementation
CVE-2016-9793 [24]	DoS, Overflow, Mem. Corr.	✓	2016-12-28	A boundary check error in net/core/sock.c
CVE-2016-4997 [25]	DoS, Priv, Mem. Corr.	✓	2016-07-03	A boundary check error in setsockopt implementation

2.2 Separation of Virtual Memory

Kernel and user processes share virtual memory to enable high-speed management. Virtual memory access control relies on the protection of the privilege level in the kernel and the CPU, which restricts cross access between the user and kernel modes. A meltdown attack overcomes this protection, and a user process can then easily access the kernel virtual memory through a combination of out-of-order, exception, and cache latency on side channel attacks.

One meltdown attack countermeasure involves the separation of the virtual memory used for the kernel and user modes. Here, a process has two virtual memory spaces. The OS automatically changes the virtual memory during any privilege level transition from user mode to kernel mode. The user mode virtual memory only contains a small amount of kernel code that switches to the virtual memory to minimize the access range of any meltdown attack. KPTI [11] is the separation method for Linux, and the other OSs are equipped with similar mechanisms [19].

2.3 Kernel Vulnerability Attack

Kernel vulnerability categorizes several aspects of implementation. These implementations address 10 types of kernel vulnerabilities [1], and the relationships across 16 exploitation techniques to distinguish 16 types of Common Weakness Enumeration (CWE) [20] vulnerabilities [21]. In addition, the CVE database states that Linux has 8 types of vulnerabilities [3], one of which is privilege escalation. Malicious programs overwrite the credential variable in the kernel virtual memory to gain root privilege.

The OS utilizes privilege level management to protect the kernel code or data in the kernel virtual memory from the user mode, while KASLR/CFI reduces the success of kernel exploitation attacks, and SMAP/SMEP restricts the kernel mode execution of a malicious code in the user virtual memory. Nevertheless, 128 memory corruption vulnerabilities were reported for the Linux kernel until June. 2019 (Table 1) [3]. Memory corruption vulnerabilities (e.g., the eBPF vulnerability and others in Table 2) are still available whereby the directory allocates malicious code to the kernel virtual memory through a kernel vulnerability.

2.4 Threat Model

We postulate herein that a threat model (i.e., an adversary) exploits kernel vulnerability only in the kernel mode. An adversary aims to compromise the OS and become capable of running any program (e.g., shell command) without security restrictions. It first attempts to avoid the security features to gain full administrator capability. Moreover, the adversary changes the Linux Security Modules (LSM) hook function pointer variable to disable MAC in Linux. Consequently, the adversary achieves privilege escalation in the OS.

We ensure that memory corruption kernel vulnerability involving overwriting of the kernel virtual memory space occurs only in the kernel vulnerability target memory region that includes the security feature functions pointer, kernel module management data, and a direct mapping region. Additionally, we assume that the BIOS, MMU, TLB, and other hardware are safe.

3. KMO Design

We designed a "Kernel Memory Observer" (KMO) (Fig. 3) that creates a secret virtual memory in the kernel mode. This supports the execution of monitor code in the kernel virtual memory. It is established at a different location of the kernel virtual memory management from the latest kernel (e.g., Linux with KPTI). The KMO's kernel possesses two kernel virtual memories (i.e., original and secret).

3.1 Design Goal

3.2 Switching Patterns and Detection Capability

The objective of KMO is to protect the kernel security feature code, data, and kernel module on the kernel virtual memory; then, KMO monitors these memory regions to detect invalid overwriting.

The kernel virtual memory permits reading, writing, and execution in the kernel mode, but not in the user mode.

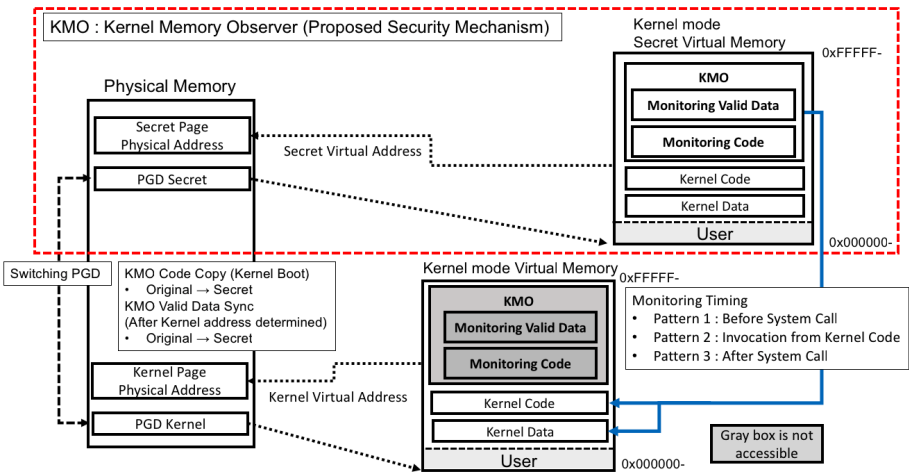

Fig. 3 Overview of monitoring on the secret virtual memory space.

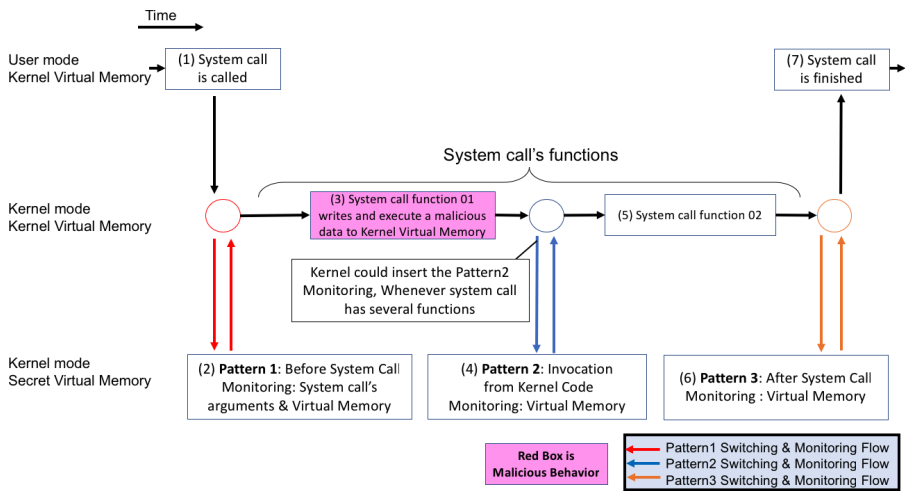

Fig. 4 Virtual memory switching patterns 1, 2, and 3.

latest kernel (e.g., Linux with KPTI), which enables the isolation of virtual memories, involves the user and the kernel virtual memory for a process. Whenever a process invokes a system call processing, the kernel automatically switches both virtual memories during privilege transitions between the user mode and kernel modes. All processes share the kernel virtual memory; therefore, the kernel still provides one virtual memory space that is available for the various features at the kernel layer. Kernel memory corruption vulnerability can potentially lead to invalid overwriting of specific kernel virtual memory region occurs only when the kernel code is running in kernel mode.

KMO creates a secret virtual memory space isolated from the original kernel virtual memory. This separation ensures that access violation is impossible between the secret and the original kernel virtual memory. KMO places the valid monitoring data and the monitoring code on the secret virtual memory, which is unaffected by the memory corruption on the kernel virtual memory. KMO generates the valid monitoring data from an original benign kernel code and data at the kernel boot. KMO then executes the monitoring code on the secret virtual memory. It verifies the kernel code and data for modification by comparing them with the valid monitoring data.

KMO is a monitoring mechanism that adopts three virtual memory switching patterns depending on the kernel manages process transition between the user and kernel modes (Fig. 4).

Pattern 1: **Inspection point is undertaken before the system call execution.** Pattern 1 involves inspecting whether the system call argument is suspicious data input before the adversary can execute malicious code by using the kernel vulnerability.

Pattern 2: **Inspection points during system call function or kernel code processing.** Pattern 2 inspects the kernel code and data in the kernel virtual memory. There may be inspection points having multiple functions during a system call consisting of multiple functions. Pattern 2 involves the direct detection of memory corruption in the

kernel virtual memory for any timing during the kernel function flow.

Pattern 3: **Inspection point is undertaken after the system call execution.** Pattern 3 inspects the kernel code and data in the kernel virtual memory. It reliably detects memory corruption after an attack completes a system call execution.

Therefore, KMO automatically switches from every pattern for system call invocation at the kernel layer. It combines multiple inspection point of patterns at one system call invocation. Although it is effective in detecting kernel memory corruption and attacks, the number of inspections results in a significant overhead. We examined the attacks on our mechanism to identify suitable inspection points in a running system.

Upon identifying the attack, the kernel handles the interruption of system calls by returning the error number to the user process. Additionally, the kernel is considered available to fix the modified memory region.

3.3 Design Approach

KMO overcomes three challenges facing the monitoring of kernel virtual memory in kernel mode.

Challenge 1: The monitoring code has access permission for monitored data and will be executed in the secret virtual memory. KMO has three virtual memory switching patterns with different inspection points on a running kernel with system calls. The inspection timing at which memory corruption is detected is also differs for each switching pattern. It efficiently monitors the already implemented kernel security feature and the module space in the kernel virtual memory to detect memory corruption attacks.
For virtual memory switching, KMO writes the physical address of the multiple-page table of the secret virtual memory into a specific register (i.e., CR3 register points to the page table for x86_64). The monitoring code executes in the secret virtual memory space. After monitoring, the KMO writes the physical address of the original kernel virtual memory into a specific register (i.e., the CR3 register for x86_64), and then continues the processing of the kernel code before the switching event occurs.

Challenge 2: The kernel code cannot access the secret virtual memory space.
KMO fully copies the secret memory space from the original one such that both memory spaces contain the same kernel code, kernel data, monitoring code, and monitoring data.
The monitoring code and valid monitoring data are not accessed through the page table flag management for the original kernel virtual memory. Therefore, in kernel mode, the original and secret virtual memory are completely isolated in KMO, ensuring that the kernel code acts on the original kernel virtual memory space by using its virtual addresses. Furthermore, it ensures that the monitored kernel code cannot access the kernel mode secret virtual memory space.

Challenge 3: The monitoring code and valid data are not affected through a direct mapping space. The kernel virtual memory management provides a direct mapping space containing the physical memory for effective page-based memory allocation and collection. KMO shares the physical memory between the kernel and the secret virtual memory, which can be abused by allowing direct access to the monitoring code and valid data KMO modifies the allocation mechanism of direct mapping to prevent memory corruption via the direct mapping space.
To exclude the monitoring code and valid data from the direct mapping of the kernel virtual memory, KMO forces the unmapping of these in the kernel virtual memory.
To exclude the monitoring code and valid data from the direct mapping of the kernel virtual memory, KMO forces the unmapping of these in the kernel virtual memory.

4. KMO Implementation

We implemented KMO with Linux as the target OS and x86_64 as the CPU architecture.

4.1 Secret Virtual Memory Space Management

KMO can monitor the kernel virtual memory (Fig. 5). The latest Linux kernel has the KPTI feature, which already provides each process has 2 virtual memory spaces.

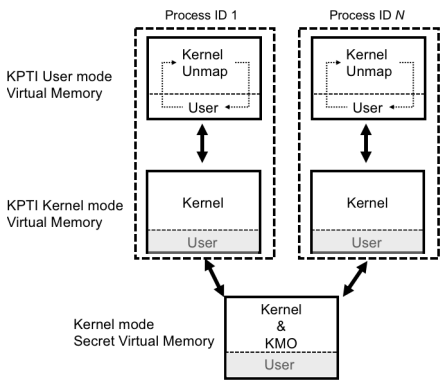

Fig. 5 Overview of secret virtual memory space for Linux kernel.

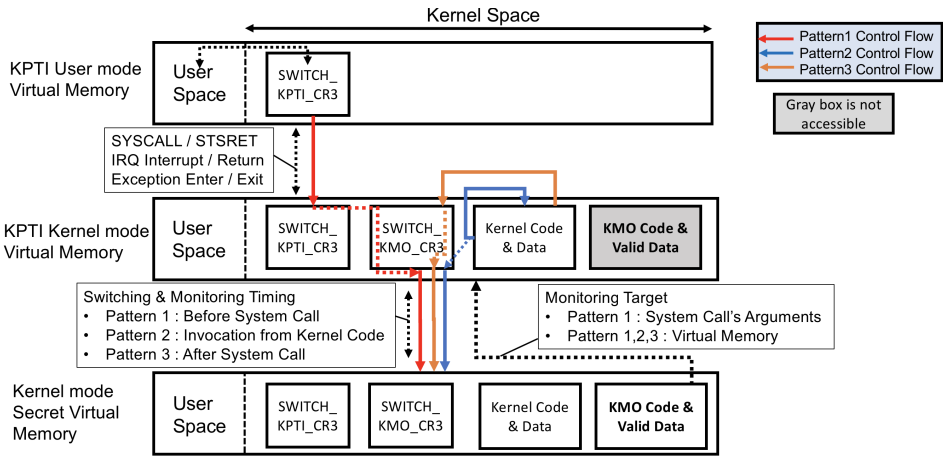

Fig. 6 Virtual memory space switching on Linux kernel.

For the kernel, the `pgd` variable of `init_mm` in `mm_struct` points to the physical memory address of the kernel virtual memory. KMO creates an additional virtual memory space on the kernel whose physical address is a 4 page (16 KB on x86_64) logical conjunction from the physical address of the `pgd` variable of `init_mm`. Moreover, the kernel code and data are duplicated from the `pgd` variable. KMO uses the physical address of the created virtual memory to switch from the kernel virtual memory to the monitoring of each process in the kernel mode.

4.2 Switching of the Virtual Memory Space

We implemented KMO to provide a virtual memory switching mechanism for the secret virtual memory space in kernel mode (Fig. 6).

In user mode, Interrupt (SYSCALL, IRQ) and Exception are triggered for the transition to kernel mode. It calls the SWITCH_KPTI_CR3 function on the virtual memory of the user and then changes to the kernel virtual memory space.

In kernel mode, KMO fulfills **challenge 1**, as the kernel calls the SWITCH_KMO_CR3 function, which calculates a 4-page offset to the physical address of the secret virtual memory space from the `pgd` variable of `init_mm`. The kernel writes this value to the CR3 register, followed by automatically switching the virtual memory space for monitoring. After the monitoring process, the SWITCH_KMO_CR3 function writes the physical address of the `pgd` variable in the `active_mm` of the `current` (`task_struct`) variable to the CR3 register, which can change the virtual memory space for the currently running process in kernel mode. The kernel then calls SWITCH_KPTI_CR3 to change the virtual memory space for the user, and the system changes to user mode via an interrupt (SYSRET Interrupt return) or exception (Exception exit).

KMO currently supports the PCID during CR3 register writing. It enables the cache of TLB entry (lower 12 bits of CR3 value) by using the conversion of caches between the virtual and physical addresses on the specific CPU. If the CPU or environment does not support PCID, KMO uses TLB flush after the CR3 register writing. However, the virtual address conversion is accompanied by overheads.

4.3 Monitoring of Virtual Memory Space

KMO has almost the same virtual memory space layout both original and secret virtual memory in Linux x86_64 (Fig. 7). KMO monitors the `security_hook_list` variable for LSM on the kernel text mapping and the module list variable `modules` in the kernel virtual memory. Additionally, KMO disables Copy on Write of the monitored data, whereas it supports targeted kernel space reading after virtual memory switching occurs. KMO fulfills **challenge 2** as both the monitoring code and the valid monitoring data have a designated flag setting that does not accept reading and writing at the supervisor level on the Page Table Entry.

KMO keeps the secret virtual memory space in the kernel boot sequence and then starts the monitoring feature according to the following sequence.

1. The `mm_init` function initializes the kernel virtual memory, whereas the `kaiser_init` of KPTI function initializes the virtual memory for the user on the kernel boot sequence.
2. KMO initializes the secret virtual memory in physical memory.
3. The `security_init` function initializes the LSM and MAC mechanism.
4. The `load_default_modules` function executes the module reading process on the kernel.
5. KMO duplicates the valid monitoring data between the secret and kernel virtual memory spaces.
6. KMO starts the monitoring feature in the secret virtual memory space.

4.4 Direct Mapping Management

Linux 4.4 (x86_64) has a direct mapping space of 64 TB.

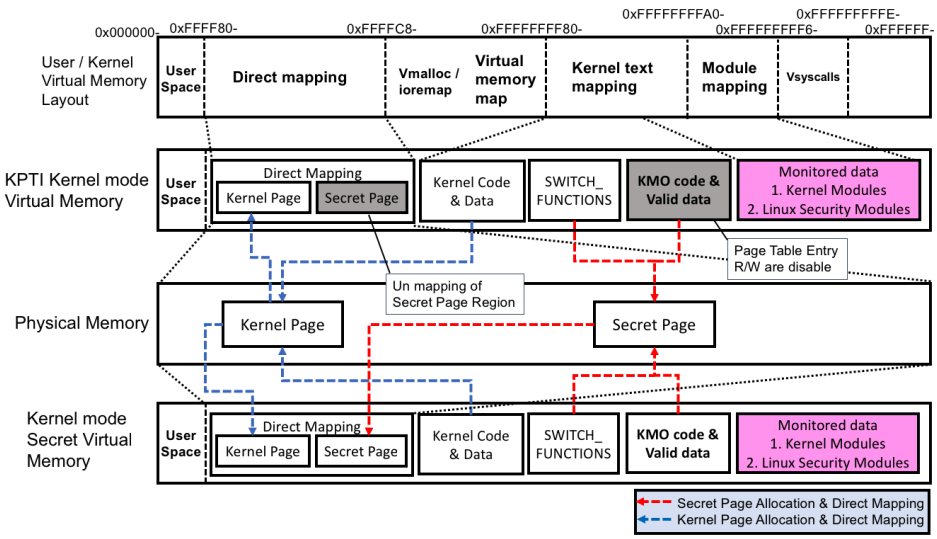

Fig. 7 Position and unmap region for the virtual memory space on Linux x86_64.

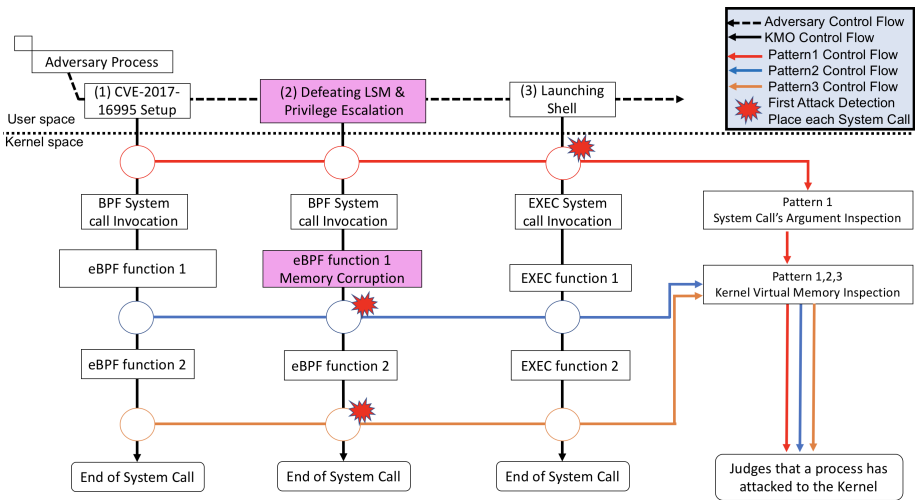

Fig. 8 Monitoring attacker process using the secret virtual memory space on Linux. The eBPF Vulnerability (CVE-2017-16995) attack flow with multiple system calls.

Therefore, the machine physical memory is mapped to a space of less than 64 TB, and the kernel manages physical page allocation by using direct mapping. Thus, it is possible to access the kernel code and the data virtual and direct mapping virtual addresses.

Linux uses the `init_mem_mapping` function to create the virtual memory direct mapping space. The `kernel_physical_mapping_init` function then maps the physical addresses to virtual memory. KMO covers **challenge 3**, as KMO uses the `remove_pagetable` function to unmap secret pages of the monitoring code and valid monitoring data from the direct mapping space in the kernel virtual memory after establishing a secret virtual memory setup (Fig. 7). Any access to the unmapped pages occurs through a page fault. Subsequently, KMO registers an original page fault handler of unmapped memory region for panic processing.

4.5 Kernel Vulnerability Attacking Case

In one of the memory corruption kernel vulnerability cases, the adversary uses the eBPF vulnerability [15] to modify the targeted data on the kernel virtual memory. The adversary finally takes the shell as the root capability without any LSM limitation after memory corruption overrides the SELinux function pointer, as well as credential information. KMO monitors these modifications and detects the following sequences (Fig. 8):

1. The adversary executes the PoC code of the eBPF vulnerability with the user privilege. The PoC code inserts malicious BPF code into the kernel virtual memory via the `sys_bpf` system call. Although KMO traps the system calls, this does not induce suspicious behavior at the time.

2. The adversary overwrites the LSM function pointer and performs privilege escalation through memory corruption via the `sys_bpf` system call at the kernel mode. KMO also traps the issued system calls. The KMO's Pattern 2 monitoring identifies the LSM function pointer modification on the kernel control flow. It compares the `security_hook_list` variable with the monitoring data containing valid data and determines whether the monitored data is invalid. If the KMO's Pattern 2 is not invoked at kernel, KMO's Pattern 3 also traps it and identifies the same malicious behavior after the eBPF system call execution.
3. The adversary launches the shell program from the PoC code. In KMO's Pattern 1, it traps the `sys_exec` system call and then determines whether it constitutes malicious behavior. System call arguments contain the shell program name, and memory corruption is already identified upon modification of the variable `security_hook_list`.

5. Evaluation

5.1 Evaluation Purpose and Environment

We evaluated KMO's effectiveness in term of detection capability and overhead. The evaluation items and objectives are described below:

E1: Monitoring system call argument experiment
We evaluate switching Pattern 1 of KMO to verify whether the target system call argument is valid before system call execution.

E2: Detection of overwriting of LSM function
We assess switching Patterns 2 and 3 of KMO to check whether they correctly identify an eBPF vulnerability PoC that modifies the LSM function's virtual address. We then determined the timing at which the attacks on the kernel virtual memory are detected.

E3: Evaluation of the prevention of access to direct mapping
We examine whether KMO prevents access to valid monitoring data in the direct mapping space after KMO unmaps secret pages.

E4: Measurement of overhead of system call interaction with KMO
We monitor the effect of kernel availability with KMO by switching the virtual memory space. We then measure the overhead by benchmark software to calculate the system call latency.

E5: Measurement of the overhead of application with KMO
We measure the performance overhead of a user process by using benchmark software on KMO, which adopts several virtual memory switching patterns.

We evaluated KMO on a physical machine with an Intel(R) Core(TM) i7-7700HQ (2.80 GHz, x86_64) and 16 GB DDR4 memory. The implementation targeted Linux Kernel 4.4.114 on Debian 9.0. We modified 17 source files regarding alternative virtual memory, virtual memory switching and monitoring functions, which required 1,653 lines in the Linux kernel. We customized eBPF kernel vulnerability [15] PoC for the modification of any virtual address on the kernel virtual memory. Of the evaluations E4 and E5, we compared the vanilla kernel with PCID (K0), KMO kernel without PCID (K1), and KMO kernel with PCID (K2). K0 supports TLB caches for user and kernel virtual memory. Although K1 does not enable a TLB cache of secret virtual memory, K2 pushes user, kernel, and secret virtual memory into TLB cache mechanism.

```
// Switching to secret virtual memory and monitoring, pattern 1
1.  [58.690804] target system call
2.  [58.690821] system call number: 0000000000000139
3.  [58.721702] module name: malicious_module
4.  [58.721731] Invalid malicious_module
// Switching to kernel virtual memory, pattern 1
5.  [58.727898] malicious_module: module license 'unspecified' taints kernel.
6.  [58.728542] Disabling lock debugging due to kernel taint
7.  [58.772438] Attack Module Init
```

Fig. 9 Monitoring results for Linux system call arguments.

5.2 Monitoring System Call Argument

We assumed a rootkit installation. KMO monitors the module installation mechanism that uses the `init_module` and `finit_module` system calls. It inspects the kernel module binary image from the system call argument and then outputs whether the module is invalid as the detection result. In the log message, switching Pattern 1 denotes the monitoring system call as "target system call" and the invalid module as "invalid module name".

KMO correctly identifies the invalid kernel from the system call argument (Fig. 9). The monitoring function detects invalid module names via the module binary for only 0.05 ms before the kernel executes the system call and then invokes the module initial function.

We confirmed that switching Pattern 1 yields the correct evaluation results for the monitoring and inspection of the system call argument. Although the module executes its initialization function, the module installation process is not yet completed at the time of detection in Patterns 1. This indicates that KMO interrupts the kernel code, specifically the system call invocation mechanism to determine if the validation is possible before system call processing.

5.3 Detection of Linux Security Module Overwrite

We create a kernel module that replaces the LSM hook function. Our custom eBPF vulnerability PoC forces the exchange of one LSM hook function in the `selinux_hooks` variable to the kernel module function on the kernel virtual memory. KMO stores the valid data at kernel boot. It then automatically identifies this memory corruption on switching patterns 2 and 3. These patterns compare the target LSM hook function's virtual address with the valid monitoring data, and then outputs the result is output as a log message.

Table 3 Overhead of switching virtual memory space and monitoring (μs).

System call	Vanilla kernel	KMO kernel		Overhead	
	PCID (K0)	NO PCID (K1)	PCID (K2)	K1-K0	K2-K0
fork+/bin/sh	914.900	946.758	925.269	31.858	10.369
fork+execve	260.357	274.589	265.324	14.232	4.967
fork+exit	238.784	255.276	244.128	16.492	5.344
fstat	0.359	0.384	0.377	0.025	0.018
open/close	7.245	7.598	7.293	0.353	0.048
read	0.356	0.358	0.358	0.002	0.002
write	0.309	0.312	0.310	0.003	0.001
stat	2.322	2.408	2.351	0.086	0.029

```
// Install LKM
1.  [78.654425] lkm_address_module: module license 'unspecified' taints kernel.
2.  [78.654853] Disabling lock debugging due to kernel taint
3.  [78.718498] dummy_hook_function Address Value ffffffffa0000000
4.  [78.718427] selinux_hooks[56].hook.file_permission Address ffffffff81e77c18
5.  [78.718444] selinux_hooks[56].hook.file_permission Address Value ffffffff812f3f20

// CVE-2017-16995 attack overrides LSM function address via sys_bpf()
6.  [*] attaching bpf backdoor to socket
7.  [*] UID from cred structure: 33, matches the current: 33
8.  [*] hammering cred structure at ffff88001c4399c0
9.  [*] replacing target function at ffffffff81e77c18
10. [*] credentials patched, launching shell...

// Switching to secret virtual memory and monitoring, pattern 2
11. [100.772834] Invalid lsm function is detected
12. [100.772854] Address ffffffff812f3f20    (Valid), ffffffffa0000000    (Invalid)
// Switching to kernel virtual memory, pattern 2

// Switching to secret virtual memory and monitoring, pattern 3
13. [204.010413] Invalid lsm function is detected
14. [204.010457] Address ffffffff812d5c70    (Valid), ffffffffa0000000    (Invalid)
// Switching to kernel virtual memory, pattern03

// LKM automatically outputs the target function virtual address
15. [108.769250] skpt_selinux_hooks[56].hook.file_permission Address ffffffff81e77c18
16. [108.769291] skpt_selinux_hooks[56].hook.file_permission Address Value ffffffffa0000000
```

Fig. 10 Monitoring result for LSM function.

An invalid case is denoted by "Invalid lsm function is detected" and "Virtual Address (Invalid)" in the detection.

KMO's detection result is successful on Patterns 2 and 3 (Fig. 10). Patterns 2 and 3 determine whether the illegal memory is overwritten after the LSM function is modified for detection.

We also confirm that switching patterns 2 and 3 determine the illegal memory corruption at suitable detection timings. Therefore, KMO has an effective detection capability for kernel vulnerability against attacks that modify the LSM function's virtual address to prevent its existence in the kernel virtual memory.

5.4 Evaluation of Direct Mapping Access Validation

We evaluate KMO when preventing the overriding of valid monitoring data with invalid data via the direct mapping region in the kernel virtual memory. The KMO unmaps the specific page of the valid data on direct mapping at kernel boot. Subsequently, the kernel module installation attempts to write the invalid data and output the result to a log message.

Figure 11 shows the log information after the unmapping process. The kernel module calculates the virtual address on direct mapping from the correct virtual address of the valid data and then accesses it. Next, the kernel issues the page request for the unmapped page having the virtual address on direct mapping. Thus, overwriting fails from the kernel module's write access via direct mapping.

```
[ 143.610533] calling change_kernel_physmap_data_attack()
[ 143.612688] valid data 1st virtual address: ffffffff820de600 (address), ffffffff812d5c80 (data)
[ 143.612922] valid data pfn: 00000000000020de
[ 143.612989] valid data pfn phys: 00000000020de000
[ 143.613009] valid data pfn virtual address: ffff8800020de000
[ 143.613218] valid data 2nd virtual address: ffff8800020de600 (address)
[ 143.613234] valid data 2nd virtual address: ffffffff812d5c80 (data)
// Unmapping valid data on direct mapping
[ 143.614117] BUG: unable to handle kernel paging request at ffff8800020de6005>]
string.isra.4+0x65/0xd0
```

Fig. 11 Preventing result for modification through direct mapping.

5.5 Measurement System Call Interaction Overhead

We compare the Linux kernel, including KMO's mechanism, with a vanilla Linux kernel to measure the performance overhead. We use the benchmark software, lmbench, and execute it 10 times to calculate the average score and determine whether each system call has an overhead effect.

The overhead results are the measurement switching virtual memory and monitoring features. The monitoring process compares only one LSM function's address. Consequently the switching of the virtual memory for each system call occurs (Table 3). lmbench shows different counts of system calls invoked for each benchmark. fork+/bin/sh has approximately 54 invocations; fork+execve has 4 invocations; fork+exit has 2 invocations; open/close has 2 invocations; and the others have 1 invocation.

Table 3 shows that the overhead of the KMO (NO PCID) and KMO (PCID) versions. In KMO (NO PCID), the system calls with the highest overheads is fork+exit (8.246 μs per 1 system call invocation). The system calls with lower overheads are read (0.002 μs) and write (0.003 μs). A kernel with KMO (NO PCID) exhibits an overhead of 0.002 μs to 8.246 μs for each system call invocation.

Otherwise, KMO (PCID) has the highest overheads is fork+exit (2.672 μs per 1 system call invocation). The system calls with lower overheads are write (0.001 μs) and read (0.002 μs). KMO (PCID) exhibits a range of overhead from 0.001 μs to 2.672 μs for each system call invocation.

The lmbench overhead results indicate that the performance of KMO is affected by the switching virtual memory and monitoring process cost. The PCID contributes to the context switching of processes and kernel thread memory access to reduce the overhead of KMO.

Table 4 ApacheBench overhead of virtual memory switching and monitoring on the Linux kernel (μs).

File size (KB)	Vanilla kernel	KMO kernel		Overhead	
	PCID (K0)	NO PCID (K1)	PCID (K2)	K1-K0	K2-K0
1	1, 041.26	1, 080.96	1, 050.60	39.70	9.34
10	1, 878.02	1, 962.70	1, 895.78	84.68	17.76
100	9, 621.70	10, 012.22	9, 718.02	390.52	96.32

Table 5 ApacheBench overhead of virtual memory switching patterns 1 and 3 with monitoring (μs).

File size (KB)	Vanilla kernel	KMO kernel (PCID)		Overhead	
	PCID (P0)	Pattern 1 (P1)	Pattern 3 (P3)	P1-P0	P3-P0
1	1, 041.26	1, 051.89	1, 050.01	10.63	8.75
10	1, 878.02	1, 892.16	1, 893.10	14.14	15.08
100	9, 621.70	9, 711.18	9, 678.76	89.48	57.06

5.6 Measurement Application Overhead

We compared the application overhead among the vanilla kernel, KMO (NO PCID), and KMO (PCID) kernels. Additionally, We evaluate the effect of PCID on both kernels with switching patterns 1 and 3 We run an Apache 2.4.25 process. The benchmark software is ApacheBench 2.4. The environment includes a 100-Mbps network, one connection, and benchmark file sizes of 1 KB, 10 KB, and 100 KB. The ApacheBench calculates one download request average of 100,000 accesses to each file. The client machine is an Intel(R) Core(TM) i5 4200U (1.6 GHz, two cores), with 8 GB of memory and running Windows 8 as the OS.

The virtual memory switching patterns call the same monitoring process because the evaluation measures the performance effect of the kernel on each switching pattern. The monitoring process compares the one LSM function's address. We compared the KMO (NO PCID) and KMO (PCID) kernels. KMO (NO PCID) switches the virtual memory every 200 system call invocations, whereas KMO (PCID) switches it every 100 system call invocations (Table 4). In KMO (PCID) for Patterns 1 and 3, the monitoring function is called to evaluate the differences of the two patterns' overheads for every 100 system call invocations (Table 5).

KMO (NO PCID) has an overhead of 39.70 μs to 390.52 μs, and KMO (PCID) has an overhead of 9.34 μs to 96.32 μs at each HTTP access. Additionally, the KMO (PCID) of Pattern 1 is 10.63 μs to 89.48 μs, and that of Pattern 3 is 8.75 μs to 57.06 μs at each HTTP access.

The overhead of ApacheBench depends on the total number of system call invocations in the process. The ApacheBench result shows that Patterns 1 and 3 slightly increase the overhead factor for a large file. When used on the benchmark, the overhead cost becomes relatively small at the application processing time. We consider that Pattern 1 to require an argument inspection of register transfer cost with a high impact. Pattern 3 incurs the same overhead cost, indicating that the switching of virtual memory and the memory monitoring have a constant of inspection cost.

6. Discussion

6.1 Performance Consideration

We consider the performance overhead, whereby KMO enables the reduction of the performance overhead by using tag-based TLBs that provide an Address Space Identifier. The PCID on x86 is the cache for the virtual address to physical address conversion. The cache on TLBs improves physical memory access without a page table walk to identify targeted page after a CR3 register update. Moreover, mitigation of overhead of KMO (PCID) relies on the number of TLB cache hits for page accesses while monitoring the secret virtual memory. Additionally, the Linux KPTI mechanism uses the PCID of TLB. Moreover, KMO (PCID) enables the storage of KPTI's caches without a TLB flush to perform a quick virtual memory switch. A virtual machine feature or cloud service may not provide the PCID of the virtual CPU, and KMO requires a performance penalty for calling the TLB flush for CR3 register updation in that environment.

Moreover, the application process has no overhead in user mode. Nearly the entire performance effect involves the switching of the virtual memory, followed by the monitoring feature in kernel mode. The overhead cost in the system call latency evaluation is identical for all types of system calls. We estimate that the actual application performance is proportional to the switching virtual memory and the monitoring process in the kernel mode after system call invocation in user mode.

The monitoring system call has a different cost for the type of system call argument. A string variable or address information has low overhead, but reading the data from the user mode has a high overhead. KMO requires nearly the same cost as an actual system call. In addition, the monitoring function's cost depends on the monitoring data size. We assume that the target kernel code or data affects the comparison process with the valid data.

6.2 KMO Detection Capability

Kernel vulnerabilities that enable privilege escalation have two effect types in the kernel layer. One type induces memory corruption on the kernel virtual memory (e.g., eBPF vulnerability [15]), whereas the other type does not create any kernel memory side effects (e.g., Dirty COW vulnerability [26]). If an adversary attempts to gain full administrator privilege of the OS, kernel memory corruption vulnerability is high priority to execute on attack scenarios for the defeating of security features. KMO provides a combination of switching virtual memory patterns having different inspection timings. Its feature detection capabilities compensate for the memory corruption of kernel vulnerability attacks for the kernel mode. During the evaluation, the eBPF vulnerability attack overwrites the SELinux functions' virtual address of the LSM hook variable that was automatically detected on KMO for protecting the security features on the kernel

Moreover, KMO identifies an attack code starting point from the user space and kernel space via using multiple system calls for the prevention of kernel vulnerability attacks leading to memory corruption. At an actual attack detection point, Pattern 1 determines the attack before system call execution on the kernel and prevents memory corruption. Although Pattern 2 identifies memory corruption, it interrupts the kernel execution flow of multiple functions having one system call. The user inserts a suitable detection point to reduce the effect of the kernel vulnerability attack. Preventing the execution of malicious code on one system call invocation for Pattern 3 is difficult because its checkpoint is just before switching back to the user mode. Therefore, Pattern 3 reliably detects memory corruption during kernel processing for multiple functions.

Additionally, we plan to support other security features that can run in the secret virtual memory space. This method could prevent attacks on kernel vulnerabilities that can evade a monitoring mechanism of a security feature on the kernel.

6.3 Limitation of KMO

KMO keeps the virtual memory switching functions in the kernel virtual memory space and then invokes them from the original kernel code. Although the adversary can potentially target the KMO's function, KASLR provides a random layout of the virtual memory space, thus obscuring the KMO functions' virtual addresses.

The adversary identifies the valid monitoring data's virtual address of direct mapping to manually calculate the position from the physical page's virtual address. In response, KMO unmaps the secret pages of the direct mapping space in the kernel virtual memory to reduce the attack surface.

7. Related Work

Operating System Security. The OS provides several security mechanisms that mainly focus on access management of the relation between a subject and an object, which is represented by whether a policy exhibits granularity of privilege range or type. Moreover, isolation architectures and models that control the separation combine in multiple layers from hardware to software [27], [28].

Kernel and CPU Enhancement. Linux also has security mechanism implementations such as SELinux [5], [29] and a capability [4] to restrict privileges. Prior studies have presented kernel security methods such as KASLR [6] for virtual memory randomization, CFI [7] for code flow integrity checking, and Code Pointer Integrity for the verification of a function's return address [30]. The CPU already possessed the NX-bit [9] for execution management and SMAP/SMEP [10] for access and the execution of control between a supervisor and a user of the virtual memory space. These reduce the kernel vulnerability attack effects.

Kernel Attack. Several attack concepts target the kernel virtual memory [11], [31], [32] to evade these security mechanisms by the side channel attack. KPTI or another method that separates the virtual memory space between the user and the kernel can mitigate such attacks [11], [33]. The kernel attack method uses both return oriented programming and anti-CFI [34], whereas the direct mapping space can execute the attack code only in the kernel mode [2], [35]. The device driver has a directory threat surface [36]. We believe that kernel virtual memory monitoring is essential to mitigating these attacks in kernel mode.

Virtualization Protection. Kernel monitoring mechanisms have a hypervisor, and a secure mode is proposed [28], [37]–[39]. Moreover, SecVisor [37] and TrustVisor [40] ensure that only the verified kernel code is running. GRIM also possesses a verified kernel code at the GPU layer [41], and Trusted Computing Base [42] verifies the integrity of the kernel code at the boot sequence. These mechanisms are run under the kernel layer and are unaffected by kernel vulnerability. KMO's monitoring feature resides in the kernel that requires relatively small overhead with the existing hypervisor methods.

Virtual Memory Protection. Virtual memory protection methods separate the memory space by the domain and granularity of memory access control [43]–[46]. The CPU feature, MPK, supports virtual memory protection [47], [48], page-based separation instructions [49], [50], and physical memory isolation for each process [51]. Moreover, monitoring of the same layer as the OS or hypervisor has a low overhead when hardware assistance is available [52]–[54]. The separation between the module and the kernel virtual memory space on the hypervisor increases system reliability [55].

Running Kernel Protection. The running kernel protection methods focus on invalid overwriting of kernel code and data, including the control flow or data flow tracing [7], [56], [57], and monitoring of the stack status [8], [58]. These methods target invalid overwriting of the kernel code or privilege variable in the kernel virtual memory. We regard these as an effective reference to reduce or trigger kernel

Table 6 Kernel monitoring feature comparison (✓ is supported; △ is partially supported).

Feature	SecVisor [37]	SIM [52]	ED-Monitor [53]	KMO
Memory Corruption Detection	✓		✓	✓
Memory Corruption Protection		△		△
System Call Argument Inspection		✓		✓
In Kernel Interception		✓	△	✓
Kernel Integrity	✓		✓	△
Cloud Environment Deployment		△	△	✓

monitoring.

Kernel Protection. The kernel protection methods have randomized page table positions in the physical memory [59], and RˆX restricts the permission of the kernel memory layout [60]. Additionally, the separation of the device driver code from the kernel provides granularity monitoring points [61], [62]. KMO has difference merits, that is, switching of the virtual memory space has no effect for attacks via kernel vulnerabilities and no interruption when running the kernel code. This leads to a collaboration of existing methods of kernel security mechanisms. Finally, we evaluate our mechanism's capability to protect existing kernel vulnerabilities and maintain stable operation on the system, and we compare it with other security mechanisms (e.g., SELinux, KASLR, and SMAP/SMEP).

7.1 Comparison with Related Work

We compared the security features of KMO and three research mechanisms (Table 6) [37], [52], [53]. KMO satisfies almost all the identified requirements for the running kernel and the cloud environment.

SecVisor [37], which completely monitors the kernel from the hypervisor layer intercepts device access to maintain the kernel integrity; however, the inspection granularity has a limitation: it is dependent on hardware assistance. Secure in VM (SIM) [52] directly inserts an alternative address space to the guest kernel from the hypervisor to monitor the kernel. The event driven (ED) Monitor [53] ensures hypervisor integrity as the same layer focuses upon the memory protection by having the kernel module insert a hooking placement.

Although the same privilege layer monitoring approach is similar to the KMO architectures, we provide finer inspection points for memory protection and detection through system calls or the insertion of a kernel function flow. Finally, although KMO may struggle to set a suitable inspection point on a kernel, users can manage effective kernel monitoring by considering a collaboration of existing methods. We believe that KMO contributes to device drivers or other potentially vulnerable regions in reducing the attack surface of the system.

8. Conclusion

The OS must mitigate various attacks that exploit kernel vulnerability. Most attacks that focus on kernel virtual memory corruption aim to perform privilege escalation or defeat security features. The OS kernel should reduce the effect of the attack on the adoption capability and MAC restrict privileges. Although KASLR, CFI, KPTI, and SMAP/SMEP mitigate kernel vulnerability attacks for memory corruption leading to privilege escalation or the avoidance of security features, only kernel layer attacks have the potential to succeed.

Our novel security mechanism, KMO, provides secret observation to monitor the original kernel virtual memory. KMO has multiple inspection points to determine invalid kernel virtual memory overwriting, identify malicious system call arguments, and prevent attacks through the direct mapping region. The evaluation of Linux with KMO could inspect system call arguments and identify the memory corruption of security features. The performance overhead from 0.002 μs to 8.246 μs in terms of each system call invocation on our kernel. The web client program overhead for KMO monitoring from 39.70 μs to 390.52 μs at the running process. KMO supports PCID on TLB, which reduces the overhead penalty of KMO, to enable the actual cost for monitoring and virtual memory switching.

Acknowledgments

This work was partially supported by JSPS KAKENHI Grant Number JP19H04109.

References

[1] H. Chen, Y. Mao, X. Wang, D. Zhou, N. Zeldovich, and M.F. Kaashoek, "Linux kernel vulnerabilities: state-of-the-art defenses and open problems," Proc. 2nd Asia-Pacific Workshop on Systems (APSys), 2011.
[2] P.V. Kemerlis, et al., ret2dir - Rethinking Kernel Isolation, the 23rd USENIX Conference on Security Symposium, pp.957–972, 2014.
[3] Linux Vulnerability Statistics, available from https://www.cvedetails.com/vendor/33/Linux.html. (accessed 2019-07-05).
[4] T.A. Linden, "Operating System Structures to Support Security and Reliable Software," ACM Computing Surveys (CSUR), vol.8, no.4, pp.409–445, 1976.
[5] Security-enhanced Linux, available from http://www.nsa.gov/research/selinux/, (accessed 2018-08-10).
[6] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh, "On the effectiveness of address-space randomization," Proc. 11th ACM Conference on Computer and Communications Security (CCS), pp.298–307, 2004.
[7] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti, "Control-Flow Integrity," Principles, Implementations, Proc. 12th ACM Conference on Computer and Communications Security (CCS), pp.340–353, 2005.

[8] P.V. Kemerlis, et al., kGuard - Lightweight Kernel Protection against Return-to-User Attacks, the 21st USENIX Conference on Security Symposium, 2012.
[9] Ingo Molnar, [announce] [patch] NX (No eXecute) support for x86, 2.6.7-rc2-bk2, http://lkml.iu.edu/hypermail/linux/kernel/0406.0/0497.html, 2004. (accessed 2018-08-10).
[10] D. Mulnix, Intel® Xeon® Processor D Product Family Technical Overview, https://software.intel.com/en-us/articles/intel-xeon-processor-d-product-family-technical-overview, 2015, (accessed 2018-08-10).
[11] D. Gruss, M. Lipp, M. Schwarz, R. Fellner, C. Maurice, and S. Mangard, "KASLR is Dead: Long Live KASLR," 2017 International Symposium on Engineering Secure Software and Systems (ESSoS), Lecture Notes in Computer Science, vol.10379, no.3, pp.161–176, Springer, Cham, 2017.
[12] CVE-2016-8655, available from https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-8655. (accessed 2019-05-12).
[13] CVE-2017-6074, available from https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-6074 (accessed 2019-05-12).
[14] CVE-2017-7308, available from https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-7308 (accessed 2019-05-12).
[15] CVE-2017-16995, available from https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-16995 (accessed 2019-05-12).
[16] Exploit Database, Nexus 5 Android 5.0 - Privilege Escalation, available from https://www.exploit-db.com/exploits/35711/ (accessed 2019-06-15).
[17] grsecurity: super fun 2.6.30+/RHEL5 2.6.18 local kernel exploit, available from https://grsecurity.net/~spender/exploits/exploit2.txt (accessed 2019-06-15).
[18] H. Kuzuno and T. Yamauchi, "KMO: Kernel Memory Observer to Identify Memory Corruption by Secret Inspection Mechanism," The 15th International Conference on Information Security Practice and Experience (ISPEC), Lecture Notes in Computer Science, vol.11879, pp.75–94, Springer, Cham, 2019.
[19] M. Lipp, et al., Meltdown - Reading Kernel Memory from User Space, the 27th USENIX Conference on Security Symposium, 2018.
[20] Common Weakness Enumeration, available from https://cwe.mitre.org/ (accessed 2019-06-15).
[21] Linux Kernel Defence Map, available from https://github.com/a13xp0p0v/linux-kernel-defence-map (accessed 2019-06-05).
[22] CVE-2017-1000112, available from https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-1000112 (accessed 2019-05-12).
[23] CVE-2017-7533, available from https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-7533 (accessed 2019-05-12).
[24] CVE-2016-9793, available from https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-9793 (accessed 2019-05-12).
[25] CVE-2016-4997, available from https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-4997 (accessed 2019-05-12).
[26] CVE-2016-5195, available from https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-5195 (accessed 2019-06-05).
[27] R. Shu, et al., A Study of Security Isolation Techniques, ACM Computing Surveys (CSUR), vol.49, no.3, pp.1–37, 2016.
[28] F. Zhang and H. Zhang, "SoK: A Study of Using Hardware-assisted Isolated Execution Environments for Security," Proc. Hardware and Architectural Support for Security and Privacy 2016, pp.1–8, 2016.
[29] R. Spencer, et al., The Flask Security Architecture: System Support for Diverse Security Policies, the 8th USENIX Conference on Security Symposium, 1999.
[30] K. Volodymyr., et al., Code-Pointer Integrity, 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI), 2014.
[31] R. Hund, C. Willems, and T. Holz, "Practical Timing Side Channel Attacks against Kernel Space ASLR," 2013 IEEE Symposium on Security and Privacy, pp.191–205, 2013.
[32] Y. Jang, S. Lee, and T. Kim, "Breaking Kernel Address Space Layout Randomization with Intel TSX," Proc. 2016 ACM Conference on Computer and Communications Security (CCS), pp.380–392, 2016.
[33] Z. Hua, et al., EPTI - Efficient Defence against Meltdown Attack for Unpatched VMs, 2018 USENIX Annual Technical Conference (ATC), 2018.
[34] N. Carlini, et al., Control-Flow Bending: On the Effectiveness of Control-Flow Integrity, the 24th USENIX Conference on Security Symposium, pp.161–176, 2015.
[35] H. Shacham, "The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)," Proc. 14th ACM Conference on Computer and Communications Security (CCS), pp.552–561, 2007.
[36] D. Song, F. Hetzelt, D. Das, C. Spensky, Y. Na, S. Volckaert, G. Vigna, C. Kruegel, J.-P. Seifert, and M. Franz, "PeriScope: An Effective Probing and Fuzzing Framework for the Hardware-OS Boundary," Proc. 26th Annual Network and Distributed System Security Conference (NDSS), 2019.
[37] A. Seshadri, et al.: "SecVisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity OSes," Proc. 21st ACM Symposium on Operating systems principles (SOSP), pp.335–350, 2007.
[38] A. Azab, K. Swidowski, R. Bhutkar, J. Ma, W. Shen, R. Wang, and P. Ning, "SKEE: A Lightweight Secure Kernel-level Execution Environment for ARM," Proc. 2011 Network and Distributed System Security Symposium (NDSS), 2016.
[39] Y. Cho, D. Kwon, H. Yi, and Y. Paek, "Dynamic Virtual Address Range Adjustment for Intra-Level Privilege Separation on ARM," Proc. 2017 Network and Distributed System Security Symposium (NDSS), 2017.
[40] M.J. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig, "TrustVisor: Efficient TCB Reduction and Attestation," 2010 IEEE Symposium on Security and Privacy, pp.143–158, 2010.
[41] L. Koromilas, G. Vasiliadis, E. Athanasopoulos, and S. Ioannidis, "GRIM: Leveraging GPUs for Kernel Integrity Monitoring," Proc. 19th International Symposium on Research in Attacks, Intrusions and Defenses, Lecture Notes in Computer Science, vol.9854, pp.3–23, Springer, Cham, 2016.
[42] Trusted computing group. tpm main specification. http://www.trustedcomputinggroup.org/resources/tpm_main_specification, 2003, (accessed 2018-08-10).
[43] E.W. Rhee, J. Rhee, and K. Asanović, "Mondrix: Memory Isolation for Linux using Mondriaan Memory Protection," Proc. 20th ACM Symposium on Operating systems principles (SOSP), pp.31–44, 2005.
[44] M. Castro, M. Costa, J.-P. Martin, M. Peinado, P. Akritidis, A. Donnelly, P. Barham, and R. Black, "Fast byte-granularity software fault isolation," Proc. 22nd ACM Symposium on Operating systems principles (SOSP), pp.45–58, 2009.
[45] T.C.-H. Hsu, K. Hoffman, P. Eugster, and M. Payer, "Enforcing Least Privilege Memory Views for Multithreaded Applications," Proc. 2016 ACM Conference on Computer and Communications Security (CCS), pp.393–405, 2016.
[46] J. Litton, et al., Light-Weight Contexts - An OS Abstraction for Safety and Performance, 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI), 2016.
[47] K. Koning, X. Chen, H. Bos, C. Giuffrida, and E. Athanasopoulos, "No Need to Hide: Protecting Safe Regions on Commodity Hardware," Proc. Twelfth European System Conference (EuroSys), pp.437–452, 2017
[48] A. Vahldiek-Oberwagner, et al., ERIM: Secure and Efficient In-process Isolation with Memory Protection Keys, CoRR abs/1801.06822, 2018.
[49] L. Mogosanu, A. Rane, and N. Dautenhahn, "MicroStache: A Lightweight Execution Context for In-Process Safe Region Isolation," The 21st International Symposium on Research in Attacks, Intrusions and Defenses (RAID), Lecture Notes in Computer Science, vol.11050, pp.359–379, Springer, Cham, 2018.
[50] T. Frassetto, et al., IMIX - In-Process Memory Isolation EXtension, the 28th USENIX Conference on Security Symposium, 2018.

[51] C.H. Kim, T. Kim, H. Choi, Z. Gu, B. Lee, X. Zhang, and D. Xu, "Securing Real-Time Microcontroller Systems through Customized Memory View Switching," Proc. 25th Network and Distributed System Security Symposium (NDSS), 2018.
[52] M.I. Sharif, W. Lee, W. Cui, and A. Lanzi, "Secure in-VM monitoring using hardware virtualization," Proc. 16th ACM Conference on Computer and Communications Security (CCS), 2009.
[53] L. Deng, P. Liu, J. Xu, P. Chen, and Q. Zeng, "Dancing with Wolves: Towards Practical Event-driven VMM Monitoring," Proc. 13th ACM SIGPLAN/SIGOPS International Conference, pp.83–96, 2017.
[54] Z. Zhang, Y. Cheng, S. Nepal, D. Liu, Q. Shen, and F. Rabhi, "KASR: A Reliable and Practical Approach to Attack Surface Reduction of Commodity OS Kernels," The 21st International Symposium on Research in Attacks, Intrusions and Defenses (RAID), Lecture Notes in Computer Science, vol.11050, pp.691–710, Springer, Cham, 2018.
[55] A. Srivastava, et al., Efficient Monitoring of Untrusted Kernel-Mode Execution, the 18th Annual Network and Distributed System Security Conference (NDSS), 2011.
[56] C. Song, B. Lee, K. Lu, W. Harris, T. Kim, and W. Lee, "Enforcing Kernel Security Invariants with Data Flow Integrity," Proc. 2016 Annual Network and Distributed System Security Symposium (NDSS), 2016.
[57] X. Ge, W. Cui, and T. Jaeger, "GRIFFIN: Guarding Control Flows Using Intel Processor Trace," Proc. 22nd ACM International Conference on Architectural Support for Programming Languages and Operating Systems (APLOS), pp.585–598, 2017.
[58] W. Huang, Z. Huang, D. Miyani, and D. Lie, "LMP: Light-Weighted Memory Protection with Hardware Assistance," Proc. 32nd Annual Conference on Computer Security Applications (ACSAC), pp.460–470, 2016.
[59] L. Davi, D. Gens, C. Liebchen, and A.-R. Sadeghi, "PT-Rand: Practical Mitigation of Data-only Attacks against Page Tables," Proc. 23th Network and Distributed System Security Symposium (NDSS), 2017.
[60] M. Pomonis, T. Petsios, A.D. Keromytis, M. Polychronakis, and V.P. Kemerlis, "kR^X: Comprehensive Kernel Protection against Just-In-Time Code Reuse," Proc. Twelfth European Conference on Computer Systems (EuroSys), pp.420–436, 2017.
[61] S. Boyd-Wickizer, et al., Tolerating Malicious Device Drivers in Linux, USENIX Annual Technical Conference (ATC), 2010.
[62] D.J. Tian, G. Hernandez, J.I. Choi, V. Frost, P.C. Johnson, and K.R.B. Butler, "LBM: A Security Framework for Peripherals within the Linux Kernel," 2019 IEEE Symposium on Security and Privacy, pp.967–984, 2019.

Hiroki Kuzuno received his M.E. degree in Information Science from the Nara Institute of Science and Technology, Nara, Japan, in 2007. Since joining SECOM in 2007, he has been engaged in research on cyber security specifically on networks and operating systems. He is a member of IEICE and IPSJ.

Toshihiro Yamauchi received his B.E., M.E. and Ph.D. degrees in computer science from Kyushu University, Japan in 1998, 2000 and 2002, respectively. In 2001 he was a Research Fellow of the Japan Society for the Promotion of Science. In 2002 he became a Research Associate in Faculty of Information Science and Electrical Engineering at Kyushu University. He has been serving as associate professor of Graduate School of Natural Science and Technology at Okayama University since 2005. His research interests include operating systems and computer security. He is a member of IPSJ, IEICE, ACM, USENIX and IEEE.