IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received July 2, 2021, accepted July 26, 2021, date of publication July 30, 2021, date of current version August 16, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3101452

Mitigation of Kernel Memory Corruption Using Multiple Kernel Memory Mechanism

HIROKI KUZUNO[1] AND TOSHIHIRO YAMAUCHI[2], (Member, IEEE)
[1]Intelligent Systems Laboratory, SECOM Company Ltd., Mitaka, Tokyo 181–8528, Japan
[2]Graduate School of Natural Science and Technology, Okayama University, Okayama 700–8530, Japan

Corresponding author: Hiroki Kuzuno (kuzuno@s.okayama-u.ac.jp)

This work was supported in part by the Japan Society for the Promotion of Science (JSPS) KAKENHI under Grant JP19H04109.

ABSTRACT Operating systems adopt kernel protection methods (e.g., mandatory access control, kernel address space layout randomization, control flow integrity, and kernel page table isolation) as essential countermeasures to reduce the likelihood of kernel vulnerability attacks. However, kernel memory corruption can still occur via the execution of malicious kernel code at the kernel layer. This is because the vulnerable kernel code and the attack target kernel code or kernel data are located in the same kernel address space. To gain complete control of a host, adversaries focus on kernel code invocations, such as function pointers that rely on the starting points of the kernel protection methods. To mitigate such subversion attacks, this paper presents multiple kernel memory (MKM), which employs an alternative design for kernel address space separation. The MKM mechanism focuses on the isolation granularity of the kernel address space during each execution of the kernel code. MKM provides two kernel address spaces, namely, i) the trampoline kernel address space, which acts as the gateway feature between user and kernel modes and ii) the security kernel address space, which utilizes the localization of the kernel protection methods (i.e., kernel observation). Additionally, MKM achieves the encapsulation of the vulnerable kernel code to prevent access to the kernel code invocations of the separated kernel address space. The evaluation results demonstrated that MKM can protect the kernel code and kernel data from a proof-of-concept kernel vulnerability that could lead to kernel memory corruption. In addition, the performance results of MKM indicate that the system call overhead latency ranges from 0.020 μs to 0.5445 μs, while the web application benchmark ranges from 196.27 μs to 6,685.73 μs for each download access of 100,000 Hypertext Transfer Protocol sessions. MKM attained a 97.65% system benchmark score and a 99.76% kernel compilation time.

INDEX TERMS Memory corruption, kernel vulnerability, system security, operating system.

I. INTRODUCTION

A kernel vulnerability attack in an operating system (OS) is a serious issue that compromises security in [1]. Kernel memory corruption is a common type of vulnerability that is used as an attack vector in [2]. Approximately 5,850 kernel memory corruption vulnerabilities were reported until 2020 as shown by the common vulnerabilities and exposures (CVE) list [3].

The adversary carries out a privilege escalation, which employs an illegal kernel memory corruption to overwrite or force kernel code invocations that result in the modification of privileged information variables to obtain full control of an administrator account as mentioned earlier [1], [2].

The associate editor coordinating the review of this manuscript and approving it for publication was Junaid Arshad.

To prevent such subversions, several kernel protection methods have been implemented. In [4], the authors have proposed the stack monitoring of the kernel code to prevent buffer overflows. The study conducted in [5] proposed the kernel address space layout randomization (KASLR), which randomizes the position of the virtual address range of the kernel codes and data in the kernel address space. Another study conducted in [6] proposed the control flow integrity (CFI), which verifies a relation of kernel code invocations to prevent the execution of malicious programs with modifications of the return address. Moreover, the kernel memory observer (KMO) segregates the kernel monitoring code in the dedicated kernel address space in [7]. To prevent meltdown side channel attacks, kernel page table isolation (KPTI) isolates the kernel memory from the kernel and user modes in [8].

In [9], the trusted computing base (TCB) ensures the trustworthiness of a small set of system components. TCB of the firmware and kernel are combined with a secure boot that validates the TCBs using signatures stored in the tamper-proof device during the bootstrap process in [10]. Additionally, another mechanism was developed in [11], wherein the supervisor mode access prevention (SMAP) forcefully denies access, and the supervisor mode execution prevention (SMEP) prevents execution on the user region.

These countermeasures mitigate the effects of kernel memory corruption, thereby reducing the success rate of kernel attacks (i.e., identification of vulnerable kernel code, insertion of malicious code, and finally, the execution of a malicious code). Nevertheless, invocation placements of kernel codes for kernel protection methods (e.g., function pointer) must be assigned to the same kernel address space with a vulnerable kernel code. The invocation placements of kernel protection methods at the kernel layer remains an unaddressed threat. Adversaries can try to replace these invocation placements with a function pointer of meaningless kernel codes to defeat the kernel security mechanisms (such as mandatory access control (MAC)) of SELinux in [12]) and gain elevated privileges via vulnerable kernel codes in [13], [14].

This paper describes the characteristics of a novel security mechanism known as the multiple kernel memory (MKM) to enhance the resistance of the kernel to kernel memory corruption. As part of MKM, two kernel address spaces (trampoline and security) are introduced. To isolate the invocation placements and kernel codes of kernel protection methods from a vulnerable kernel codes, MKM assigns the feature of kernel address space switching to the trampoline kernel address space. It also assigns kernel protection methods to the security kernel address space. Thereby, MKM encapsulates the vulnerable kernel code in the original kernel address space. The multiple kernel address space of MKM mitigates kernel vulnerability attacks that subvert the kernel protection methods and the invocations placement of the kernel code. Therefore, MKM forcefully restricts the visible region of the kernel address space for vulnerable kernel codes to reduce the likelihood of damage caused by kernel memory corruption. An overview of the MKM security mechanism is as follows:

- As part of the MKM design, two kernel address spaces are provided one for the trampoline and the another for security to set the kernel code execution boundary. The kernel address space for the trampoline acts as the gateway between the user and kernel modes, and it supports the switching feature of the kernel address space. Subsequently, the kernel address space for security covers the kernel protection methods. The original kernel address space supports remaining the kernel codes that contain the kernel vulnerabilities.
- Reducing the kernel's attack surface requires a separation of the kernel address space to isolate the accessible kernel code range at the kernel layer. The MKM mechanism eliminates the risk of kernel memory corruption of the invocation placements of kernel code (e.g., the kernel switching gateway of the kernel address space and the kernel protection methods). These kernel codes and function pointer values are stored on the trampoline and security kernel address space. The MKM forcefully encapsulates the vulnerable kernel codes that can only move to the original kernel address spaces.

In short, the primary contributions of this study are:

1) The proposed MKM mechanism, which is a novel kernel memory separation mechanism designed to specifically protect kernel protection methods at the kernel layer. The MKM approach improves resilience against kernel vulnerability attacks. This paper also discusses the threat model, capability, limitations, portability, and hardware considerations of MKM.
2) The efficacy of the implemented MKM was evaluated with an actual kernel vulnerability (CVE-2017-16995 in [15]) Proof-of-Concept (PoC) code that subverts SELinux and the MKM switching of the kernel address space. The kernel observation mechanism accurately detected both cases as kernel memory corruption. MKM's performance evaluation results indicate that the round time overhead for system calls were between 0.020 μs to 0.5445 μs. The overhead for each Hypertext Transfer Protocol (HTTP) download access via a web application was between 196.27 μs to 6,685.73 μs. Additionally, MKM achieved an acceptable performance score that indicates system benchmark suites of 97.65% and a compiling time of 99.76% for Linux kernel.

Finally, this paper highlights the contribution differences from the previous paper in [16]. Although this paper is based on conference proceedings in [7], [16], the present paper provides additional design, evaluation results, and implementation portability. Additionally, this paper introduced the general concept of MKM that can mitigate the memory corruption for running kernel and actual kernel vulnerability attacks that cannot break the security capability of MKM in [16]. As an evaluation result, the performance overhead was evaluated based on the system call invocation and HTTP server, which has a micro coverage in [16]. In a previous paper [16], the authors discussed the memory observation mechanism that identifies the kernel memory corruption at the kernel layer, which is a different study of the primary concept of this paper. The present paper also includes a certain part of the results, as described in [16] that is otherwise cited in a submitted paper. However, definite differences between the two papers have been carefully highlighted. First, this paper distinctly aims to establish the security boundary for attacking target kernel code and vulnerable kernel code. Second, it addresses the assignment of kernel codes and function pointers to different kernel address spaces (e.g., trampoline, security, and original) and generalization of the kernel address space switching sequences for the design of kernel components. Third, it details the evaluation of the feasibility of implementing MKM along with UnixBench score

and Linux kernel compiling time (e.g., compiler and linker), to attempt calculating actual kernel layer overhead for user's environment usage. Furthermore, to obtain a more technical standpoint, this study investigated the proposed portability of mechanism. MKM's design and implementation can be applied to other OSes (e.g., FreeBSD and XNU kernel) with different architectures (e.g., ARM). This result indicates that MKM is one of the various mitigation approaches for kernel memory corruption.

II. MEMORY CORRUPTION AND COUNTERMEASURES

A. PRIVILEGE ESCALATION ATTACK

Adversaries need to execute arbitrary program code that exploits kernel vulnerabilities to gain complete privilege escalation at the kernel layer as shown by Chen *et al*. [2]. To circumvent kernel and central processing unit (CPU) protection methods, they can corrupt the kernel memory to allow the reading and writing of kernel code or kernel data as shown by Kemerlis *et al*. [1].

On Linux, adversaries can achieve privilege escalation with kernel memory corruption as presented in Table 1. PoC codes insert a malicious program that forcefully invokes privilege update kernel functions. The functions, `commit_creds` and `prepare_kernel_cred` switch the user privileges to root privileges and are usually used by Linux's account management system in [17]–[19]. Additionally, the kernel memory corruption requires the overwriting of user account privilege information (`UID`) variables in the `cred` structure on the kernel memory in [15]. Linux uses MAC to restrict root capability in [12]. Adversaries must subvert the kernel protection methods of the Linux security module (LSM) at the kernel layer. The kernel memory corruption can also result in the replacement of the function pointer value of the LSM for `security_hook_list` with a function pointer value that points to a non-checking access control method. Moreover, adversaries alter the security context variable to circumvent the MAC system (e.g., SELinux) in [13], [14].

TABLE 1. Typical Linux memory corruption vulnerability list. 'Types' refers to Mem. Corr.: Memory corruption, Priv. Inv.: Privilege update kernel code invocation.

CVE Number	Types	Description
CVE-2016-4997 [17]	Mem. Corr., Priv. Inv.	PoC invokes credential functions
CVE-2016-9793 [18]	Mem. Corr., Priv. Inv.	PoC invokes credential functions
CVE-2017-16995 [15]	Mem. Corr.	PoC overwrites privilege information
CVE-2017-1000112 [19]	Mem. Corr., Priv. Inv.	PoC invokes credential functions

B. PROTECTION OF KERNEL MEMORY

Traditional protection mechanisms used by the CPU and kernel control the referencing of the kernel address space from user processes as mentioned earlier [20], [21]. To conduct kernel vulnerability attacks, an adversary requires virtual address information to execute an arbitrary program code on the kernel address space. Figure 1 shows that the page table maintains page entry assignments, which allocates relationships between the physical and virtual addresses for each page of the page table. The Linux x86_64 architecture uses 48 bit virtual addresses and 4 KB page sizes. The physical address of the page table is held in the control register 3 (CR3) in [22].

The user processes of an adversary require the virtual addresses to execute a malicious program via a kernel vulnerability. Although KASLR / CFI protects the kernel against exploitation kernel attacks from the user processes of an adversary as mentioned [5], [6], a meltdown side channel attack that allows a user process to refer to any location in the kernel address space without the use of existing kernel protection methods (e.g., KASLR) as demonstrated in [8]. The kernel memory isolation is the countermeasure for meltdown side channel attacks (e.g., Linux KPTI in [8]).

III. THREAT MODEL

The threat model is for an adversary who corrupts the kernel memory to execute kernel vulnerability attacks that are only successful if they are executed in the kernel mode in [2]. The adversary's aim is to obtain complete administrator privileges on the kernel. Herein, the adversary is a normal user that executes user processes to alter the function pointer variable of the access control mechanism (e.g., LSM hook) using vulnerable kernel codes, consequently disabling the MAC, updating credential variables, and the switching function of kernel address space on the original kernel address space of the MKM.

This enables the user process of the adversary to insert a malicious code into the original kernel address space, overwriting any kernel code and data. Consequently, the adversary gains complete administrator privileges on the kernel.

However, to achieve kernel memory corruption, the kernel vulnerability and the targeted kernel code and data need to be on the same kernel address spaces. Therefore, this attack cannot overwrite other kernel address spaces.

IV. THE DESIGN OF MKM

A. DESIGN CONCEPT

This study designed the multiple kernel memory (MKM) mechanism that retains the capabilities of kernel protection at the kernel layer. It prevents the subversion of the invocation placement i.e., the kernel code and function pointer.

- *Security Requirement of the Kernel Protection Method Invocation:* A traditional kernel address space is shared by the entire kernel code. Adversaries can easily subvert kernel protection methods to bypass accurate privilege checking for privilege escalation via the kernel memory modification. Further protection against kernel memory corruption requires the complete isolation of the kernel address space of kernel protection methods and kernel code invocation placements from the vulnerable kernel code in the running kernel.

B. KERNEL RESILIENCE CHALLENGE

The design concept presented in section IV-A is the challenge faced by the kernel with MKM in terms of its

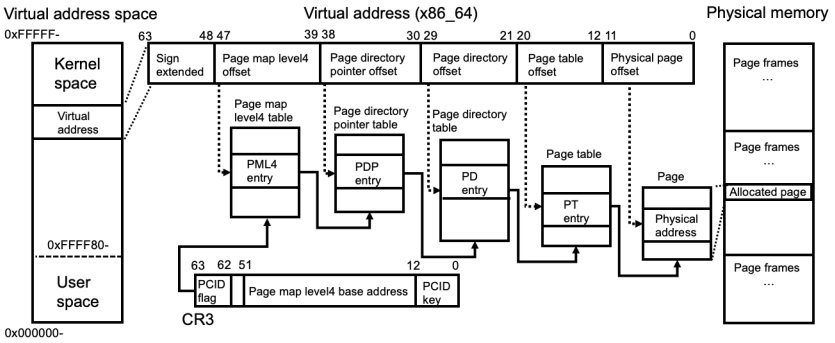

FIGURE 1. Overview of page table structure (Linux x86_64 architecture) in [22].

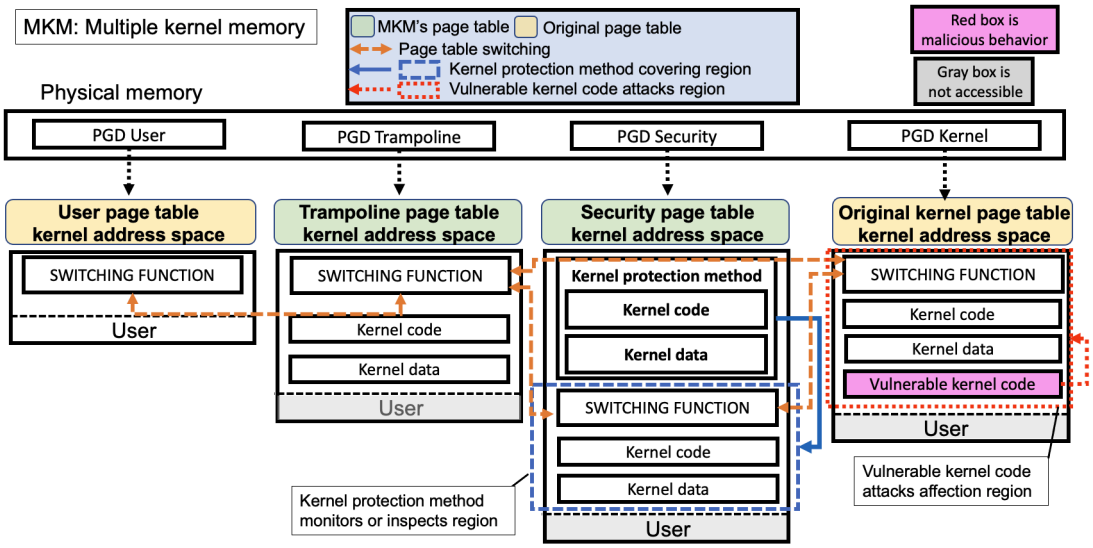

FIGURE 2. Overview of the multiple kernel memory design.

resilience. To address this concern, the following provisions are required:

- *Isolated Kernel Address Space of Kernel Protection Method Invocation*:
  To isolate the kernel address space and to execute the kernel protection method, the MKM ensures kernel resilience by manually assigning a set of kernel codes to segregated kernel address spaces. It ensures that if the kernel code is forcefully accessed, executed, and corrupted, the corruption is confined to the isolated kernel address space.

MKM satisfies the requirement of separated kernel address spaces using two dedicated kernel address spaces. Figure 2 shows an overview of how MKM provides kernel memory isolation by allocating a small set of kernel code and kernel data to each kernel address space. The role of each proposed kernel address space is as follows:

Trampoline: The trampoline kernel address space facilitates the switching functions. The main part of the trampoline kernel address space acts as a gateway for the transition between the user mode and kernel mode.

Security: The security kernel address space employs features of kernel protection methods that contain specific sets of kernel codes and data. Only the security kernel address space can execute these kernel codes. The "in and out" transition is forcefully permitted with the trampoline kernel address space between the user and kernel modes.

Even though the vulnerable kernel codes exist at the kernel layer, the reach of these kernel codes is limited to their kernel address space. The design of MKM implies that the kernel protection method codes can only execute when on the appropriate kernel address space when in kernel mode. The trampoline and security kernel address space achieve the segregation from the vulnerable kernel codes by performing kernel address space switching of the MKM.

C. KERNEL ADDRESS SPACE SWITCHING SEQUENCE

MKM defines three switching sequences for switching between multiple kernel address spaces. It ensures that the jump to the trampoline kernel address space is located in the middle of sequence 1 and 3 as shown in Figure 3.

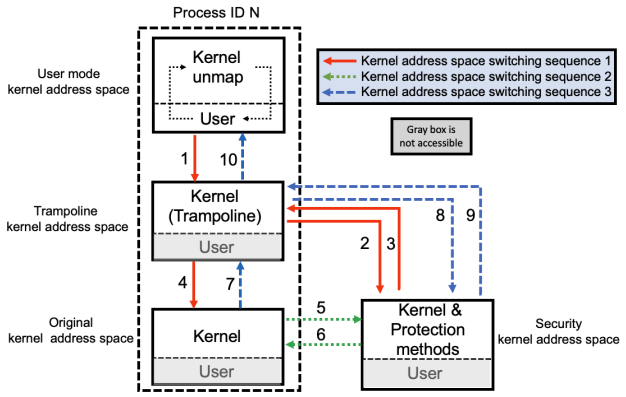

FIGURE 3. Overview of the switching sequences of the kernel address space.

Additionally, MKM forces the use of these sequences when transitioning between the user mode and the kernel mode as a result of a user process issuing a request to the kernel, as described below:

Sequence 1: User $\xrightarrow{1}$ Trampoline $\xrightarrow{2}$ Security $\xrightarrow{3}$ Trampoline $\xrightarrow{4}$ Kernel

Sequence 1 represents a system call invocation or exception request that triggers a transition from the user mode to the kernel mode. MKM is involved when switching from the kernel address space of the user mode to the original kernel address space, via the trampoline and security kernel address space. It is necessary for the execution of the kernel protection methods before the kernel feature deals with the request from the user process.

Sequence 2: Kernel $\xrightarrow{5}$ Security $\xrightarrow{6}$ Kernel

Sequence 2 represents the timing of the kernel protection method invocation during kernel feature processing. MKM switches from the original kernel address space to the security kernel address space.

Sequence 3: Kernel $\xrightarrow{7}$ Trampoline $\xrightarrow{8}$ Security $\xrightarrow{9}$ Trampoline $\xrightarrow{10}$ User

Sequence 3 represents the return to the user mode from the kernel mode. At this point, the kernel features have completed the request of the user process. MKM executes kernel protection methods while in the kernel mode. The switching flow is from the original kernel address space to the kernel address space of the user mode via the trampoline and security kernel address space.

D. ATTACK SURFACE OF MKM

MKM suppresses the attack surfaces from vulnerable kernel code. The point of reaming kernel attack surface is that a kernel vulnerability attack can lead to the corruption of the kernel memory of other kernel code or data stored in the original kernel address space in the kernel mode.

Adversaries can inject the attack code that only disrupts the switching function of original kernel address space of MKM. This disruption intercepts the execution of the kernel protection methods for the switching of the kernel address space during sequence 2 and sequence 3. MKM cannot prevent kernel memory corruption that occurs during kernel processing and after the system call invocation. The trampoline and security kernel address space retain their ability to execute the switching feature of the kernel address space (e.g., kernel code and function pointer) for sequence 1 if subjected to this attack. Therefore, the reduction of the attack surface protects the kernel protection methods and data that are executable at the kernel level. Hence, MKM can execute the kernel protection methods prior to the system call.

V. MKM IMPLEMENTATION

In this study, the MKM is implemented on Linux for the x86_64 CPU architecture and leveraged KPTI.

A. PAGE TABLE MANAGEMENT

MKM leverages KPTI by repurposing the kernel address spaces pre-assigned to them to act as the user and original kernel page tables. Additionally, MKM introduces the trampoline and security page tables for dedicated kernel address spaces as shown in Figure 4. The trampoline page table supports the switching gateway of the kernel address space. All the user processes share the security page table that supports the kernel protection methods. MKM assigns and executes the kernel memory monitoring on the security page table for the case study.

During implementation, MKM sets the initial value of the trampoline page table to the `pgd` variable of the `init_mm` structure. The security page table uses a four-page size (16 Kbytes on the x86_64 architecture) offset from the physical address of pgd (i.e., CR3 value + 0 × 4000). The user page table uses a one-page size (4 Kbytes on the x86_64 architecture) offset from the physical address of pgd (i.e., CR3 value + 0 × 1000). The original kernel page table is the `kernel_pgd` variable of `mm_struct` of the `task_struct` structure.

B. PAGE TABLE SWITCHING

MKM automatically maintains page tables switching for the execution of kernel code on the specific kernel address spaces. Figure 4 indicates that MKM employs the switching functions of the page table to transition to each page table. These functions forcibly shift the kernel address space and then continue the kernel processing to invoke the kernel codes. The three switching sequences were implemented as follows:

Sequence 1: User $\xrightarrow{1}$ Trampoline $\xrightarrow{2}$ Security $\xrightarrow{3}$ Trampoline $\xrightarrow{4}$ Kernel

Sequence 1 represents the transition from the user mode to the kernel mode. MKM employs the `SWITCH_KPTI_CR3` function to

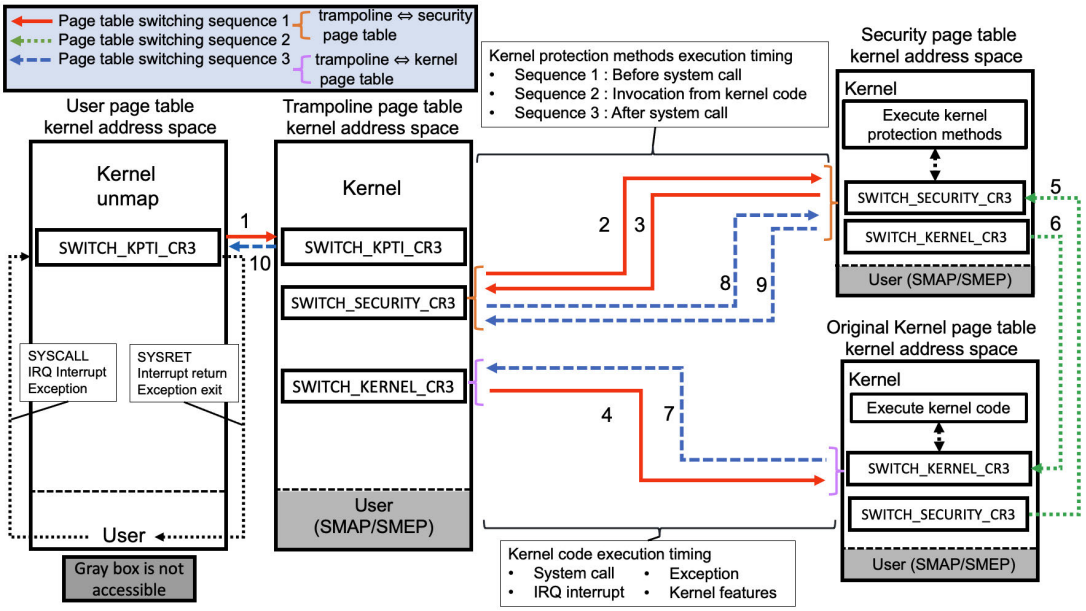

FIGURE 4. Timing of switching kernel address space.

write the physical address of the trampoline page table to the CR3. MKM also employs `SWITCH_SECURITY_CR3` to write the physical address of the security page table to the CR3. It reverts to the trampoline page table after executing the kernel protection methods. The `SWITCH_KERNEL_CR3` function writes the physical address of the original kernel page table to the CR3.

Sequence 2: Kernel $\xrightarrow{5}$ Security $\xrightarrow{6}$ Kernel

MKM utilizes `SWITCH_SECURITY_CR3` and `SWITCH_KERNEL_CR3` to switch between the security and original kernel page tables during kernel processing. Kernel protection methods are executed on the security page table.

Sequence 3: Kernel $\xrightarrow{7}$ Trampoline $\xrightarrow{8}$ Security $\xrightarrow{9}$ Trampoline $\xrightarrow{10}$ User

Sequence 3 represents the transition from the kernel mode to the user mode. The MKM uses `SWITCH_KERNEL_CR3` to write the physical address of the trampoline page table to the CR3. Moreover, MKM uses `SWITCH_SECURITY_CR3` and `SWITCH_KPTI_CR3` to write the user page table to the CR3 through the security and trampoline page tables. MKM executes kernel protection methods and reverts to the user page table via the trampoline page table.

VI. CASE STUDY

A. SECURITY FEATURES OF MKM

MKM provides the same kernel code and kernel data mapping for each kernel address space. To present an actual running case of a protection method, MKM employs the kernel observation mechanism that is only visible and executable on the security page table. The kernel observation mechanism monitors the kernel code, data, and the kernel module as shown in Figure 5. More specifically, it monitors the LSM variables, and the switching functions of the page table for memory corruption during evaluation.

During kernel boot time, MKM ensures the accuracy of the monitoring process by invoking the kernel observation mechanism after the virtual addresses of the target kernel code are specified on the kernel page table. Subsequently, the kernel observation mechanism writes valid data containing the monitoring data to the security page table.

MKM automatically handles the timing of the executions of the protection methods during switching sequences of the page table. The kernel observation mechanism starts monitoring features before and after the system call invocation and during kernel processing. During monitoring, the kernel observation mechanism compares the target data and the valid data on the security page table to catch any occurrences of kernel memory corruption.

B. PAGE TABLE SWITCHING ATTACK ON MKM

Attacks on the MKM kernel that subvert the page table switching function, leading to a complete disruption of the kernel method invocation protection, must be considered. In this study, the PoC code of the extended Berkeley packet filter (eBPF) CVE-2017-16995 was employed to evaluate the resilience of MKM to such an attack. The customized PoC code can modify any kernel address space of the original kernel page table. The PoC invokes the `map_update_elem` function of `kernel/bpf/syscall.c` to execute the exploit code that is directly inserted into the kernel. Figure 6 depicts the mechanisms of this attack and its

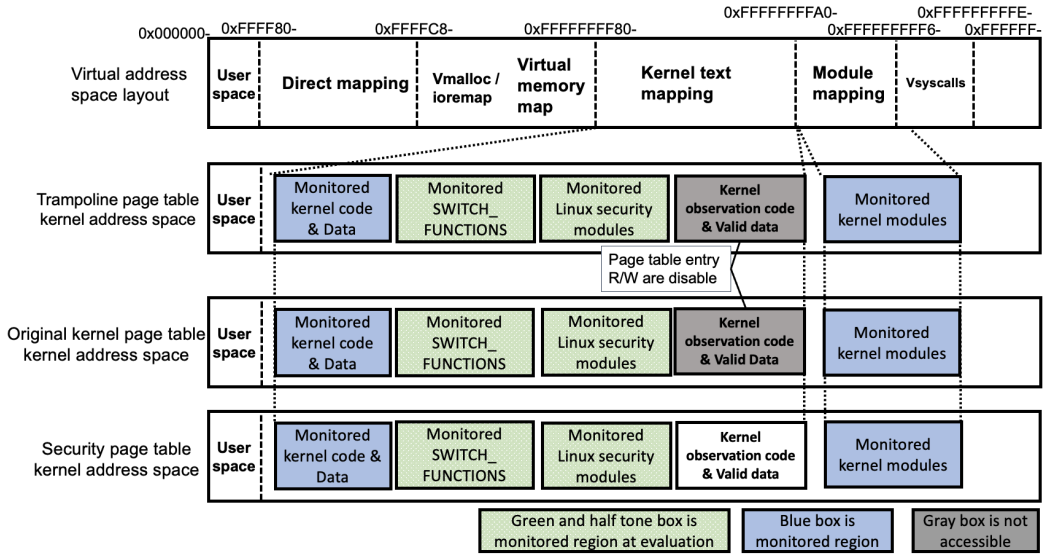

FIGURE 5. Monitoring region and evaluated region of kernel code on MKM.

detection flow. The user process of the adversary injects the exploit code into the kernel address space of the original kernel page table. Although the exploit code is executed while in the kernel mode, it corrupts the original kernel page table up to the switching functions of the page table. (e.g., `SWITCH_SECURITY_CR3` and `SWITCH_KERNEL_CR3`). The details of the kernel memory corruption and kernel processing flow are as follows:

1) The adversary's user process uses the `map_update_elem` function through an eBPF system call.
2) The adversary's user process overwrites the `SWITCH_KERNEL_CR3` function on the original kernel page table via kernel vulnerability CVE-2017-16995.
3) The MKM kernel directory switches from the kernel page table to the user page table.
4) The user process of the adversary invokes an exec system call. MKM switches the trampoline page table to the security page table to automatically start the kernel observation mechanism.
5) The kernel observation mechanism executes the monitoring code.
6) The kernel observation mechanism identifies illegal memory corruption by comparing the virtual address of `SWITCH_KERNEL_CR3` with valid data.

The user process of the adversary does not target the kernel code and kernel data on other page tables (e.g., trampoline and security) on the MKM kernel. Moreover, it is difficult for the adversary to evade the inspection timing before the system call invocation of MKM as kernel observation mechanism is invoked after the user process of the adversary has already compromised the host.

VII. EVALUATION

A. SECURITY CAPABILITY

The security capability of the MKM kernel is validated through the identification and measurement times of kernel memory corruptions to evaluate the security capability of the MKM kernel.

1) Detecting kernel memory corruption of LSM using MKM:
   MKM uses the kernel observation mechanism to determine whether the function pointer virtual addresses of the LSM are valid.
2) Detecting kernel memory corruption of the switching feature of the page table:
   MKM retains the switching capability of the page table after kernel memory corruption. Additionally, the kernel observation mechanism inspects whether the virtual address of the switching function is valid.

B. PERFORMANCE MEASUREMENT

During practical implementation, the performance was measured to compare the effects of kernel feasibility for a vanilla kernel and the MKM kernel.

1) Experimental results for the system call invocations overhead:
   Benchmarking software was used to calculate the system call latency overhead.
2) Experimental results for application overhead:
   The application executed several switches of the page tables. Benchmarking software was used to measure web access performance.
3) Experimental results of kernel processing overhead:
   Measured the processing performance overhead of the MKM kernel using benchmarking software and kernel compilation time.

C. ENVIRONMENT OF THE EVALUATION

1) IMPLEMENTATION

MKM was implemented for the Linux x86_64 kernel with KPTI and the kernel observation mechanism. To evaluate the capabilities of MKM with the kernel observation mechanism,

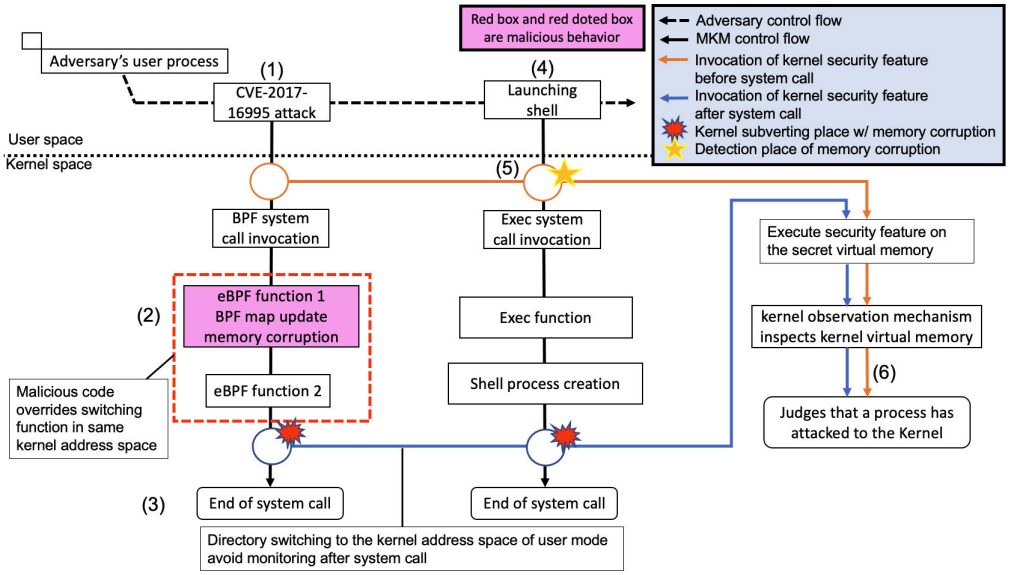

FIGURE 6. Attack and detection flow of adversary process on MKM.

actual kernel vulnerability is necessary to lead to memory corruption (e.g., CVE-2017-16995). This kernel vulnerability required a Linux kernel 4.4.114. MKM's overhead evaluation requires stable behavior for the whole of the latest Linux kernel 5.0.0. The OS environment is the Debian 9.0 distribution and the CVE-2017-16995 PoC code was customized.

MKM supports dedicated page table management, switching, and monitoring features. Those features required the addition of 27 source files and 1,254 lines of code to Linux kernel 4.4.114, and the addition of 46 source files and 1,245 lines of code to Linux kernel 5.0.0.

2) EQUIPMENT

Evaluations were conducted on a stand-alone server equipped with an Intel (R) Core (TM) i7-7700HQ (2.80 GHz, x86_64) processor and 16 GB of DDR4 memory. The client machine had an Intel (R) Core (TM) i5-4200U (1.6 GHz) with 8 GB of memory and Windows 10 OS. The network environment for the application benchmark included a 1 Gbps hub supporting different ports for the server and client machines.

D. DETECTION EXPERIMENTS FOR MEMORY CORRUPTION

1) DETECTING MEMORY CORRUPTION OF LSM USING MKM

The memory corruption resulting from the use of the eBPF kernel attack, CVE-2017-16995 disables the LSM feature of Linux. When the `sys_bpf` system call is invoked, the customized PoC code replaces the LSM hook function pointer of `selinux_hooks` with the virtual address of the original linux kernel module (LKM) function. During kernel boot, MKM stored the virtual address of the function pointer for later use as ground truth. The kernel observation mechanism inspects the target virtual addresses before and after the system call invocations. During inspection, the target virtual address is compared to the stored valid, ground truth virtual address on the security page table. Log messages record the result of the comparison.

```
1.  // Install LKM
2.  [  78.654425] lkm_address_module: module license 'unspecified' taints kernel.
3.  [  78.654853] Disabling lock debugging due to kernel taint
4.  [  78.718459] dummy_hook_function Address Value ffffffffa0000000
5.  [  78.718427]  selinux_hooks[56].hook.file_permission pointer Address ffffffff81e77c18
6.  [  78.718444] selinux_hooks[56].hook.file_permission pointer Address Value ffffffff812f3f20

7.  // Switching to Secret virtual memory and monitoring
8.  [  100.767961] Address ffffffff812f3f20 (Valid), ffffffff812f3f20 (Invalid)
9.  // Switching to KPTI Kernel virtual memory (Trampoline)

10. // Switching to Kernel virtual memory
11. // CVE-2017-16995 attack overrides LSM function address via sys_bpf()
12. // Switching to KPTI Kernel virtual memory (Trampoline)

13. // Switching to Secret virtual memory and monitoring after sys_bpf()
14. [  100.772834] Invalid lsm function is detected
15. [  100.772854] Address ffffffff812f3f20 (Valid), ffffffffa0000000 (Invalid)
16. // Switching to KPTI Kernel virtual memory (Trampoline)

17. // LKM automatically outputs the target function virtual address
18. [  108.769291] selinux_hooks[56].hook.file_permission pointer Address Value ffffffffa0000000
```

Red text is the detection point of kernel memory corruption and valid / invalid virtual address values

FIGURE 7. Detection of the kernel memory corruption of the LSM function pointer.

After memory corruption has been identified, the messages "Invalid LSM function is detected" and "Virtual Address (Invalid)" are logged. As shown in Figure 7, the MKM executed the kernel observation mechanism accurately, identifying the invalid LSM function pointers.

First, LKM logs the virtual address of the kernel module function and function pointer value of the target LSM (lines 2–6). MKM invokes the kernel observation mechanism and does not detect illegal behavior (line 8). The eBPF kernel attack using PoC commences at the kernel address space of the original kernel page table (line 11). Subsequently, MKM switches to the security page table via the trampoline page

```
1.  // Install LKM
2.  [  105.738923] lkm_address_module: module license 'unspecified' taints kernel.
3.  [  105.739633] Disabling lock debugging due to kernel taint
4.  [  105.802694] dummy_hook_function Address Value ffffffffa0000000
5.  [  105.802637] vmem_switching_function pointer Address ffffffff821257e0
6.  [  105.802657] vmem_switching_function pointer Address Value ffffffff810593e0

7.  // Switching to trampoline, security page tables, and monitoring
8.  [  159.480809] Address ffffffff810593e0 (Valid), ffffffff810593e0 (Valid)
9.  // Switching to trampoline page table

10. // Switching to kernel page table
11. // CVE-2017-16995 attack overrides the virtual memory switching function address
        via sys_bpf
12. // Switching to trampoline page table

13. // Switching to security page table and monitoring at next system call
14. [  159.484758] Invalid vmem switching function is detected
15. [  159.484776] Address ffffffff810593e0 (Valid), ffffffffa0000000 (Invalid)
16. // Switching to trampoline page table

17. // LKM automatically outputs the target virtual address
18. [  165.857354] vmem_switching_function pointer Address Value ffffffffa0000000
```

Red text is the detection point of kernel memory corruption and valid / invalid virtual address values

FIGURE 8. Detection of the memory corruption of page table switching function.

table. This enables the kernel observation mechanism that detects and identifies the actual attack (lines 14 and 15). Finally, LKM logs that the LSM virtual address points to the function of the kernel module function (line 18).

The kernel observation mechanism used by MKM identifies the memory corruption attack within 0.0049 ms of the kernel executing the PoC kernel code.

2) DETECTING MEMORY CORRUPTION OF PAGE TABLE SWITCHING

The attack on page table switching also employed the PoC of the CVE-2017-16995 kernel vulnerability to overwrite the function pointer value of the page table switching to the virtual address of the original kernel module function. The customized PoC code aims to disable sequence 3 (i.e., switching from the kernel to the trampoline page tables). The results indicate that the compromising of `SWITCH_KERNEL_CR3` was ineffective in changing the page table. MKM continued to invoke the kernel observation mechanism that identified the memory corruption and logged the messages "Invalid vmem switching function is detected" and "Virtual Address (Invalid)".

As shown in Figure 8, the kernel observation mechanism used by MKM accurately identified the memory corruption that overwrote the function pointer of `SWITCH_KERNEL_CR3`, pointing it to the kernel module function pointer on the kernel page table.

First, LKM logs the virtual address of the kernel module function and target switching function value (lines 2–6). Second, the kernel observation mechanism does not detect any illegal behavior (line 8). Third, the eBPF kernel attack modifies the function pointer of switching function value to the kernel module function (line 11). Fourth, MKM invokes the kernel observation mechanism to detect and identify the memory corruption of the trampoline and security page tables (lines 14 and 15). Finally, LKM logs the function pointer value of the switching function (line 18).

The detection of the memory corruption occurred within 0.0039 ms of the kernel executing the attack system call.

E. MEASUREMENT OF PERFORMANCE OVERHEAD

1) EXPERIMENTAL RESULTS OF SYSTEM CALL INVOCATION OVERHEAD

To measure the performance overhead incurred by the MKM kernel relative to a vanilla Linux kernel, the LMbench software was run ten times to calculate the average system call overhead score (i.e., the cost of switching page tables for each system call).

The LMbench uses different system calls invocation counts for each system call: 54 invocations for fork+/bin/sh, 4 invocations for fork + execve, two invocations for fork + exit and open/close, and one invocation for all others. Table 2 shows that the fork + exit system call has the highest overhead (0.5445 μs), whereas write has the lowest overhead (0.020 μs).

TABLE 2. System call overhead of MKM mechanism on the Linux (μs).

System call	Vanilla kernel	MKM kernel	Overhead
fork+/bin/sh	517.839	524.383	6.544
fork+execve	133.954	134.823	0.869
fork+exit	120.214	121.303	1.089
open/close	3.070	3.226	0.156
read	0.264	0.285	0.021
write	0.221	0.241	0.020
stat	1.095	1.128	0.033
fstat	0.286	0.306	0.020

2) EXPERIMENTAL RESULTS OF APPLICATION OVERHEAD

An Apache 2.4.25 web server was run on each kernel to measure the user process overhead of the MKM kernel relative to the vanilla kernel. ApacheBench 2.4, run on Windows 10 OS, was used to calculate the HTTP download request average run time for HTTP accesses. ApacheBench sends 100,000 HTTP accesses, and the files downloaded were 1 KB, 10 KB, and 100 KB in size. The tests were run over a single connection. As listed in Table 3, MKM had average overhead of 196.27 μs to 6,685.73 μs for each file download access of 100,000 HTTP sessions.

TABLE 3. ApacheBench overhead of MKM mechanism on the Linux (μs).

File size (KB)	Vanilla kernel	MKM kernel	Overhead
1	1,637.08	1,833.35	196.27
10	1,868.17	2,542.07	673.9
100	3,709.58	1,0395.31	6,685.73

3) EXPERIMENTAL RESULTS OF KERNEL PROCESSING OVERHEAD

The evaluation of kernel processing overhead involved comparison of benchmark scores and kernel compilation times between the vanilla kernel and the MKM kernel.

UnixBench evaluates CPU, inter process communication, and file handling costs in a general application execution

environment. Additionally, both kernels' compilation times were measured by compiling Linux kernel 5.0.0 with the Debian 9.0 default kernel configurations (e.g., .config file). Each kernel's score was the average of five compilation trials to measure the average specific application (e.g., compiler and linker) processing time. The MKM kernel has a 97.65% performance score with UnixBench, and a 99.76% performance score for kernel compilation performance score as presented in Table 4.

TABLE 4. UnixBench and kernel compile performance of MKM mechanism on the Linux.

	Vanilla kernel	MKM kernel	Performance
UnixBench score	1380.060	1347.617	97.65%
Kernel compile time	5977.776 (sec)	5992.326 (sec)	99.76%

VIII. DISCUSSION

A. KERNEL RESILIENCE

The PoC of the eBPF kernel vulnerability attack disables one of the page table switching mechanisms (e.g., the sequence 2) of MKM on the original kernel page table. Therefore, MKM cannot invoke the kernel protection methods after the kernel memory corruption caused by the system call invocation. However, MKM continued to execute the kernel observation mechanism on the security page table from the trampoline page table before the system call invocation.

Vulnerable kernel code can only modify its kernel address space under the restriction of the MKM. This shows that the remaining MKM mechanism did not suffer any further effects of kernel memory corruption. More specifically, MKM enhanced the kernel resilience that protected the specific kernel code and data on the trampoline and security kernel address spaces. Therefore, the implementation of MKM supports the kernel protection methods and the remaining kernel codes assigned to each individual kernel address space. The user process of the adversary cannot evade the entirety of the security capability of MKM when the malicious program has only compromised the host.

B. PERFORMANCE

The benchmark results show that the overhead from MKM handling multiple page tables was suitable for executing processes on a running kernel. To reduce performance overheads, MKM enables tag-based translation lookaside buffers (TLBs). The implementation of Linux KPTI and MKM assign the process-context identifier (PCID) of TLB. Although it requires the CR3 update, the TLB's cache improved physical memory access without a page table walk.

The application process of MKM was without any overhead in user mode. KPTI had already introduced the privileged transition between the user mode and the kernel mode, inducing page table switching overhead at system call invocations. Moreover, MKM has additional overheads when switching between the trampoline and security kernel page tables in kernel mode. The kernel observation mechanism requires additional overhead time of approximately 0.002 μs to 8.246 μs for each system call invocation in [7].

C. LIMITATION

1) DESIGN LIMITATION

MKM provides three sequences for page table switching. Each sequence has a different purpose. MKM uses sequences 1 and 3 before and after the system call invocation, respectively, as MKM's automatic protection mechanisms. Although sequence 2 requires many modifications to the kernel when manually applying MKM, it fits the adequate placement of kernel behavior to call a kernel protection method (e.g., SELinux).

Another limitation for the design of MKM is that a TLB-flush time is considered assuming that the kernel does not use the PCID of TLB. The TLB cache needs to be initialized at every page table switching, which increases the overhead time. During the implementation of MKM, the original kernel TLB-flush timing is retained, and additional TLB-flush timings are activated after page table switching when the CPU no longer supports the PCID feature. It potentially affects the performance overhead for the sequences for page table switching on the MKM kernel.

2) SECURITY LIMITATION

Two security limitations must be considered. First, MKM does not directly prevent the kernel memory corruption. Adversary's user process can still invoke the placement of actual vulnerable kernel code (e.g., CVE-2017-16995); then, other kernel codes and kernel data on the original kernel address space can be defeated.

Another security limitation is that MKM relies on page tables to achieve kernel address space isolation. MKM assumes that the kernel protection methods and vulnerable kernel codes are manually stored on different page tables to protect specific kernel codes and kernel data. Therefore, kernel protection methods and kernel drivers should be suitably isolated in the MKM mechanism.

D. PORTABILITY

Kernel address space isolation is already implemented in FreeBSD, Windows, and macOS in [23]–[25]. It indicates that Linux running with x86_64 architecture can be implemented on these OSes.

In this study, the feasibility of porting MKM is carefully examined in comparison to other OS kernels (Table 5). The MKM implementation for Linux is a reference that supports KPTI and MKM at the kernel layer. FreeBSD has already implemented kernel address space isolation method that has two page tables for user and kernel modes in [26]. Moreover, the XNU kernel has the `no_shared_cr3` flag that supports page table switching between user and kernel modes in [27]. Both approaches keep the entire kernel text and data on a one page table. MKM is needed for additional separations at the kernel layer. The trampoline page table is available and

TABLE 5. Portability consideration of MKM mechanism for OSes (✓ is supported; • is available on x86_64).

OS	Page table isolation	MKM
Linux	✓	✓
FreeBSD	✓	•
XNU kernel	✓	•

remains as a small set of kernel features. The original kernel page table supports full kernel for the FreeBSD and XNU kernels.

Additionally, other CPU architectures can be used to validate the use of multiple kernel page tables (Table 6). The x86_64 architecture indicates that certain hardware specifications are needed to support the implementation of page table isolation. ARM has translation table base registers (TTBR0 / TTBR1) that store page table addresses for address translation. Additionally, the exception level flags (EL0 / 1 / 2 / 3) specify the separation of accessible virtual address ranges and hardware execution identification of multiple address spaces for the OS, hypervisor, and secure monitor in [28].

TABLE 6. Portability consideration of MKM mechanism for architectures (✓ is supported; • is available).

Arch	Page table register	Memory separation	MKM
x86_64	CR3	SMEP / SMAP	✓
ARM	TTBRs	TTBRn_ELm	•

E. HARDWARE SUPPORT

Trusted execution environments (TEEs) provide hardware-assisted trusted computing platforms with TCB (i.e., ARM TrustZone, Intel SGX, and AMD SEV in [29]–[31]). TEEs ensure that the attack surfaces are only the host applications and OS. Therefore, the CPU and TCB are trusted. The TCB of a TEE should be constructed with minimal code set with verification. Thus, the TCB of a TEE still requires modifications to the existing kernel to allow for privilege handling (e.g., a kernel module or specific register management). Additionally, the TCB of a TEE requires a high overhead time to interact with non-TEE environments.

MKM achieves isolation of kernel components for different kernel address spaces at the kernel layer. In this study, it is necessary to consider the requirements such as separation granularity, strength of the security perimeter, and the performance cost to determine the adoption of hardware support dependency for the isolation mechanism.

IX. RELATED WORK

Research into kernel security mechanisms have yielded multiple software and hardware security techniques against potential threats. Figure 9 shows an overview of the kernel protection taxonomy, summarizing previous security mechanisms.

A. USER PROCESS

User processes are the starting points for kernel attacks, and the kernel directory provides MAC and restriction

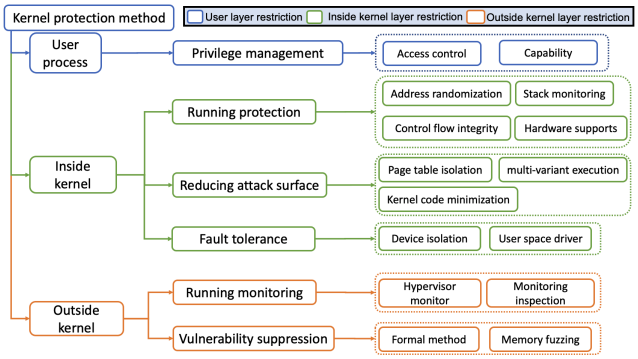

FIGURE 9. Overview of the taxonomy of kernel protection.

mechanisms at the user layer in accordance with the principle of least privilege.

1) PRIVILEGE MANAGEMENT

Security mechanism implementations (e.g., SELinux in [12]) manage access control models and polices that relate the privilege relationships among users, processes, and resources as mentioned earlier [20]. Additionally, capability restricts the granularity of root privilege between user processes and host resources, reducing the risk of a springboard case on a compromised host as shown by Linden [32].

B. INSIDE KERNEL

The inside kernel, the kernel protection mechanism for running kernels, provides a variety of security capabilities for each attack method.

Moreover, the reduction in the attack surface supports page table isolation and monitoring and hardware layer (e.g., CPU) protection. This ensures integrity and enhances security assurance. Fault tolerance is another aspect of kernel structural enhancement to be tackled given the existence of buggy software.

1) RUNNING KERNEL PROTECTION

In [4], the authors have proposed the software-based countermeasures to mitigate actual attack methods from arbitrary programs. Stack monitoring protects against buffer overflow attacks. Randomization of kernel memory layout via KASLR for kernel attack hardening has been proposed in [5]. The authors in [6] designed the kernel CFI that prevents ROP code snippets from invoking illegal kernel code execution. In [33], the authors have proposed the randomization of the page table position to protect against data only attack. Additionally, another kernel protection was developed in [34]. The kRˆX is an exclusive privilege management method that directly protects kernel code and data on the kernel memory. Moreover, hardware support is useful for kernel protection. TCB ensures that a small set of firmware and the kernel are trustworthy. Secure boot verifies system images with tamper-proof features during boot in [9], [10]. The study conducted in [35] proposed the Sprobes, which determines

TABLE 7. Comparison of the reduction in kernel memory attack surface (✓ is supported; △ is partially supported).

Feature	PerspicuOS [36]	kRazor [37]	KASR [38]	Multik [39]	KMO [7]	MKM
Page table switching protection						✓
Isolated kernel protection method	△			△	✓	✓
Memory corruption mitigation via system call	✓	✓	✓	✓	△	△
Reducing executable kernel code	✓	✓	✓			
Reducing kernel code from page table				✓	△	△

kernel integrity using CPU security features and a trusted execution environment.

2) REDUCING ATTACK SURFACE

PerspicuOS supports kernel feature isolation, enabling minimum privilege assignments for kernel code sets, proposed by Dautenhahn *et al*. [36]. kRazor, managing the list of kernel code visible to user process has been proposed in [37]. The authors in [38] proposed the KASR to provide a handling mechanism that controls page table execution permissions for user processes. In [39], authors designed the Multik that reduces the available kernel code to create the minimum mapping of kernel memory for each application. The study conducted in [8], the authors proposed the KPTI separates the user and kernel memory at the page table. In [40], the authors provide low overhead kernel separation using the extended page table on Intel CPUs. Additionally, in [7], the authors demonstrated the KMO provides a dedicated page table to isolate kernel protection methods from other kernel code. An approach was proposed by Österlund *et al*. [41] showcases that the kernel multi-variant execution supports differential virtual address spaces and stack behaviors to identify the success or failure of attacks from anomalous process.

3) FAULT TOLERANCE

The kernel executes device drivers in separate protection domains isolated from the main kernel (e.g., user space driver) as proposed in [42], [43]. In [44], the authors developed the iKernel, which separates buggy devices on virtual machines. In [45], the authors have leveraged the SIDE to set up a dedicated page table for each driver with control transfer mechanisms between kernel and drivers. These methods employ virtual address separation or kernel code behavior restrictions to protect the main kernel from malicious or buggy device drivers.

C. OUTSIDE KERNEL

The outside kernel is a runtime monitoring mechanism using hypervisor or monitoring from other hardware without attack threats from the kernel or user layers. Moreover, kernel vulnerability suppression automatically complements the additional kernel features. Fuzzing techniques identify misimplementations of additional features, and the formal verification approach subsequently ensures that the anomalous control flow behavior is excluded.

1) RUNNING KERNEL MONITORING

SecVisor and TrustVisor preserves the integrity of kernel code and data with a hypervisor that executes the monitored kernel as a guest OS as mentioned earlier [46], [47]. In an approach proposed by Sharif *et al*. [48] SIM achieves real-time inspection of kernel behavior that depend on the insertion of a monitoring mechanism into a guest OS's memory space. Recently, the authors in [49] designed the ED-Monitor as a type of kernel module that monitors hypervisor behavior using captures of register controlling. In [50], the authors demonstrated the GRIM that employs a graphics processing unit to execute protection mechanisms to monitor kernels running on the CPU.

2) VULNERABILITY SUPPRESSION

The seL4 micro-kernel adopts a formal verification scheme to ensure the functionality of memory management that contains a small kernel-level set that does not to cause a memory invalidation and other vulnerabilities as shown by Klein *et al*. [51]. In addition, kmemcheck and KASAN, with syzbot and syzkaller, automatically inspect the memory handling mechanism. Kernel memory fuzzing reveals mis-implementations that lead to complex vulnerabilities as demonstrated in [52]–[54].

D. COMPARISON

1) FEATURE COMPARISON

Table 7 shows a comparison of the security features of MKM and those of five attack surface reducing mechanisms in [7], [36]–[39]. To satisfy most attack mitigation requirements, MKM provides attack mitigation method, whereby the kernel memory is separated or minimized to mitigate the attack code and protect the kernel.

In [37], kRazor must initially collect necessary kernel features for a targeted program requirement, and it subsequently deploys a set of kernel features during user process execution. Moreover, KASR offline trains the targeted program behavior to make a kernel code database; thereafter, the hypervisor employs the executable kernel codes for the execution of the user process in [38]. Although kRazor and KASR forcibly create a minimum set of executable kernel function for the running kernel to reduce the kernel's attack surface that is exploitable by user processes, these approaches do not provide separation mechanisms for the kernel memory at each kernel feature.

In [36], PerspicuOS provides privilege separation mechanisms at the kernel layer. PerspicuOS contains a nested, trusted kernel that supports a small part of the kernel code and data, and an outer, untrusted kernel supports the rest of the kernel with de-privileging. To prevent illegal memory corruption, the nested kernel completely drops hardware control instructions (e.g., memory management unit and

TABLE 8. Gap comparison of the attack mitigation technique.

Feature	PerspicuOS [36]	kRazor [37]	KASR [38]	Multik [39]	KMO [7]	MKM
Mitigation coverage	Hardware control	Kernel feature			Security feature	Kernel feature invocation
Mitigation design	Kernel component separation	Kernel reduction	Kernel minimization	Kernel behavior	Kernel monitor	Kernel memory separation
Mitigation implementation	Hardware control wrapping	Kernel code hooking	Hypervisor monitoring	Stack and heap wrapping	Domestic page table	Page table separation
Limitation	Hardware dependency	Kernel version dependency		User application dependency	Security feature dependency	

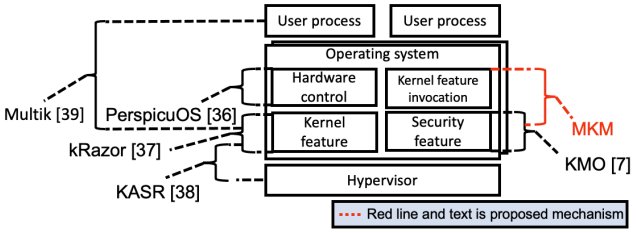

FIGURE 10. Comparison of the cover range in kernel.

CPU registers). Although MKM does not support hardware privilege deduction, MKM provides the Application Binary Interface to user processes and can be easily ported to other OSes at the kernel layer.

In [39], Multik generates customized kernel images based on the profiling of the necessary kernel codes. Applications use modified kernel page tables containing this minimum kernel at the runtime. KMO dedicates independent page tables to specific kernel codes to isolate them from the rest of the kernel codes. Multik and KMO ensure that the separated page tables are protected from illegal modification as a result of memory corruption in [7]. However, the kernel page table remains vulnerable because of vulnerable kernel codes and the invocation placement of the kernel protection method (e.g., page table switching function)

2) SECURITY TECHNIQUE COMPARISON

MKM's security capabilities are similar to those of kernel attack mitigation approaches that use memory management mechanisms. Figure 10 shows a comparison of MKM and other kernel security mechanisms proposed in [7], [36]–[39]. The earliest stage of a kernel protection method is still an attack target kernel code. To mitigate the mapping of specific kernel code and vulnerable kernel code to the same page table, the MKM architecture strongly separates the specific kernel codes of system call or kernel feature processing invocations.

Additionally, Table 8 presents a comparison of the gap of security techniques of MKM and previous security mechanisms in [7], [36]–[39]. These designs and implementations support setting a suitable attack mitigation point of a kernel component, and the users can manage the effective separation of the kernel considering each threat model on the running kernel.

To complement the remaining attack mitigation of kernel components, the security techniques of MKM enables the separated mapping of page table switching kernel codes and vulnerable kernel codes. It reduces the actual attack surface of previous kernel protection methods.

The limitation of MKM must correspond with the updating of security features' kernel code for the mapping of the suitable page table. It is a necessary process to keep the isolation between attack target kernel codes and vulnerable kernel codes. For the future approach, MKM does not provide an executable kernel code reduction mechanism for user processes. MKM's approach needs to be combined with a kernel code reduction approach to achieve a more flexible means of kernel attack surface minimization.

X. CONCLUSION

The modern OS kernel should be able to mitigate the kernel memory corruption of kernel vulnerabilities. To minimize the attack surface and prevent attacks, kernels use stack monitoring, CFI, KASLR, KPTI. However, kernel protection methods and vulnerable kernel codes share the same kernel address space; This leads to the subversion of the kernel protection methods via kernel memory corruption.

In this paper, a novel security mechanism that contains the trampoline and security kernel address spaces, i.e., MKM, is proposed. MKM improves the resilience of the kernel that relies on the separation of kernel page table. MKM assigns the switching function of page table and kernel protection methods (e.g., function pointers and kernel codes) to the trampoline and security page tables. Subsequently, vulnerable kernel codes are encapsulated in the original kernel page table. Therefore, vulnerable kernel codes of the trampoline and security kernel address spaces are cannot be targeted for kernel memory corruption. Based on the evaluation result of the Linux kernel with MKM, it is evident that MKM can mitigate the subversion of page table switching capability. Additionally, kernel observation mechanism can be implemented with the MKM to detect illegal memory modifications. The performance overhead of the system call invocations was between 0.020 μs to 0.5445 μs. The overhead to a web client program averages from 196.27 μs to 6,685.73 μs for each download access of 100,000 HTTP sessions. MKM achieved a 97.65% score for system benchmark score and a 99.76% score for kernel compilation time.

XI. FUTURE WORK

In future studies, following three research approaches should be considered. First, vulnerable kernel code execution should be restricted. Kernel protection methods detect or prohibit the unauthorized behavior of the adversary's user process. When using kernel vulnerability, the kernel protection method can not prohibit the invocation of vulnerability kernel code. To solve the granularity execution management due to the principle of kernel code risk and stability, researchers can

provide the design of kernel resilience which can not only protect attack mitigation but also ensure sufficient attack prevention on the running kernel.

The second approach is the security boundary collaboration between software and hardware. Modern CPUs have provided hardware security mechanisms (e.g., TEE in Sec. VIII-Ei). It would be better to extend the security boundary for software security mechanisms. However, the latest side-channel attack has become a new threat for hardware (e.g., Meltdown in [8]) that makes it difficult to update the user's environment. Software security approaches can mitigate side-channel attacks and complement the hardware countermeasures in [8], [40]. To tackle malicious activity, a strong and flexible security boundary should be constructed with software and modern hardware mechanisms.

Finally, the performance issue of the security mechanism must be addressed. A drawback of the security mechanism is that it requires an overhead to solve the security problem. If the security mechanism incurs heavy costs, it is impractical for the actual environment. To satisfy security levels and requirements, a simple design and lightweight implementation to demonstrate the countermeasure based on software and hardware components should be adopted. It extends security researches, so that its usefulness would be further demonstrated with additional experiments.

REFERENCES

[1] V. P. Kemerlis, M. Polychronakis, and A. D. Keromytis, "Ret2dir: Rethinking kernel isolation," in *Proc. 23rd USENIX Conf. Secur. Symp.*, Aug. 2014, pp. 957–972. [Online]. Available: https://dl.acm.org/doi/10.5555/2671225.2671286

[2] 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," in *Proc. 2nd Asia–Pacific Workshop Syst. (APSys)*, Jul. 2011, pp. 1–5, doi: 10.1145/2103799.2103805.

[3] *Linux Vulnerability Statistics*. Accessed: May 5, 2021. [Online]. Available: https://www.cvedetails.com/vulnerabilities-by-types.php

[4] V. P. Kemerlis, G. Portokalidis, and A. D. Keromytis, "Kguard: Lightweight kernel protection against return-to-user attacks," in *Proc. 21st USENIX Conf. Secur. Symp.*, Aug. 2012, pp. 459–474. [Online]. Available: https://dl.acm.org/doi/10.5555/2362793.2362832

[5] R. Hund, C. Willems, and T. Holz, "Practical timing side channel attacks against kernel space ASLR," in *Proc. IEEE Symp. Secur. Privacy*, May 2013, pp. 191–205, doi: 10.1109/SP.2013.23.

[6] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity principles, implementations," in *Proc. 12th ACM Conf. Comput. Commun. Secur.*, Nov. 2005, pp. 340–353, doi: 10.1145/1102120.1102165.

[7] H. Kuzuno and T. Yamauchi, "KMO: Kernel memory observer to identify memory corruption by secret inspection mechanism," in *Proc. 15th Int. Conf. Inf. Secur. Pract. Exper.*, vol. 11879, Nov. 2019, pp. 75–94, doi: 10.1007/978-3-030-34339-2_5.

[8] D. Gruss, M. Lipp, M. Schwarz, R. Fellner, C. Maurice, and S. Mangard, "KASLR is dead : Long live KASLR," in *Proc. 9th Int. Symp. Eng. Secure Softw. Syst.*, Jun. 2017, pp. 161–176, doi: 10.1007/978-3-319-62105-0_11.

[9] *TPM Main Specification*. Trusted Computing Group. Accessed: Aug. 10, 2018. [Online]. Available: https://www.trustedcomputinggroup.org/resources/tpm_main_specification

[10] *Trusted Boot*. Trusted Computing Group. Accessed: Sep. 26, 2020. [Online]. Available: https://trustedcomputinggroup.org/resource/trusted-boot/

[11] D. Mulnix. *Intel Xeon Processor D Product Family Technical Overview*. Accessed: Aug. 10, 2018. [Online]. Available: https://software.intel.com/en-us/articles/intel-xeon-processor-d-product-family-technical-overview

[12] *Security-Enhanced Linux*. Accessed: May 22, 2019. [Online]. Available: http://selinuxproject.org/page/Main_Page

[13] Exploit Database. *Nexus 5 Android 5.0—Privilege Escalation*. Accessed: May 21, 2019. [Online]. Available: https://www.exploit-db.com/exploits/35711/

[14] Grsecurity. *Super Fun 2.6.30+/Rhel5 2.6.18 Local Kernel Exploit*. Accessed: May 21, 2019. [Online]. Available: https://grsecurity.net/ spender/exploits/exploit2.txt

[15] *CVE-2017-16995*. Accessed: Jun. 10, 2019. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-16995

[16] H. Kuzuno and T. Yamauchi, "MKM: Multiple kernel memory for protecting page table switching mechanism against memory corruption," in *Proc. 15th Int. Workshop Secur.*, vol. 12231, Sep. 2020, pp. 97–116, doi: 10.1007/978-3-030-58208-1_6.

[17] *CVE-2016-4997*. Accessed: Jun. 10, 2019. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-4997

[18] *CVE-2016-9793*. Accessed: Jun. 10, 2019. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-9793

[19] *CVE-2017-1000112*. Accessed: Jun. 10, 2019. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-1000112

[20] B. W. Lampson, "Protection," *ACM SIGOPS Operating Syst. Rev.*, vol. 8, no. 1, pp. 18–24, Jan. 1974, doi: 10.1145/775265.775268.

[21] P. A. Karger and A. J. Herbert, "An augmented capability architecture to support lattice security and traceability of access," in *Proc. IEEE Symp. Secur. Privacy*, Apr. 1984, p. 2, doi: 10.1109/SP.1984.10001.

[22] D. P. Bovet and M. Cesati, "Memory addressing," in *Understanding the Linux Kernel*, 3rd ed. Sebastopol, CA, USA: O'Reilly Media, 2005, pp. 36–63.

[23] G. Tetlow, *Response to Meltdown and Spectre*. Accessed: May 21, 2019. [Online]. Available: https://lists.freebsd.org/pipermail/freebsd-security/2018-January/009719.html

[24] A. Ionescu. *Windows 17035 Kernel ASLR/VA Isolation In Practice (Like Linux KAISER)*. Accessed: Aug. 6, 2020. [Online]. Available: https://twitter.com/aionescu/status/930412525111296000

[25] A. Ionescu. *Say Hello to the 'Double Map' Since 10.13.2- and With Some Surprises in 10.13.3*. Accessed: Aug. 6, 2020. [Online]. Available: https://twitter.com/aionescu/status/948609809540046849

[26] K. Belousov. *Reflections on the Meltdown Fix for FreeBSD*. Accessed: Aug. 21, 2020. [Online]. Available: https://papers.freebsd.org/2018/kib-meltdown.files/meltdown.pdf

[27] Apple Inc. *XNU Source Tree*. Accessed: Aug. 21, 2020. [Online]. Available: https://github.com/apple/darwin-xnu/

[28] *ARMv8—A Address Translation Version 1.0*. ARM Inc. Accessed: Aug. 21, 2020. [Online]. Available: https://static.docs.arm.com/100940/0100/armv8_a_address%20translation_100940_0100_en.pdf

[29] *Architecture Reference Manual; ARMv7A and ARMv7R Edition*. ARM Inc. Accessed: Sep. 8, 2020. [Online]. Available: https://static.docs.arm.com/ddi0406/c/DDI0406C_C_arm_ architecture_reference_manual.pdf

[30] *Intel(R) Software Guard Extensions for Linux OS*. Intel Corporation. Accessed: Sep. 8, 2020. [Online]. Available: https://github.com/intel/linux-sgx

[31] M. Morbitzer, M. Huber, J. Horsch, and S. Wessel, "SEVered: Subverting AMD's virtual machine encryption," in *Proc. 11th Eur. Workshop Syst. Secur.*, Apr. 2018, pp. 1–6, doi: 10.1145/3193111.3193112.

[32] T. A. Linden, "Operating system structures to support security and reliable software," *ACM Comput. Surveys*, vol. 8, no. 4, pp. 409–445, Dec. 1976, doi: 10.1145/356678.356682.

[33] L. Davi, D. Gens, C. Liebchen, and A.-R. Sadeghi, "PT-Rand: Practical mitigation of data-only attacks against page tables," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, Feb. 2017, pp. 1–15.

[34] M. Pomonis and T. Petsios, "kR^X: Comprehensive kernel protection against just-in-time code reuse," in *Proc. 12th Eur. Conf. Comput. Syst.*, Apr. 2017, pp. 420–436, doi: 10.1145/3064176.3064216.

[35] X. Ge, H. Vijayakumar, and T. Jaeger, "Sprobes: Enforcing kernel code integrity on the trustzone architecture," in *Proc. 3rd Workshop Mobile Secur. Technol.*, Oct. 2014, pp. 1–10.

[36] N. Dautenhahn, T. Kasampalis, W. Dietz, J. Criswell, and V. Adve, "Nested kernel: An operating system architecture for intra-kernel privilege separation," in *Proc. 20th Int. Conf. Architectural Support Program. Lang. Operating Syst.*, Mar. 2015, pp. 191–206, doi: 10.1145/2694344.2694386.

[37] A. Kurmus, S. Dechand, and R. Kapitza, "Quantifiable run-time kernel attack surface reduction," in *Proc. 11th Int. Conf. Detection Intrusions Malware, Vulnerability Assessment*, Jul. 2014, pp. 212–234, doi: 10.1007/978-3-319-08509-8_12.

[38] 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," in *Proc. 21st Int. Symp. Res. Attacks, Intrusions, Defenses*, Sep. 2018, pp. 691–710, doi: 10.1007/978-3-030-00470-5_32.

[39] H.-C. Kuo, A. Gunasekaran, Y. Jang, S. Mohan, R. B. Bobba, D. Lie, and J. Walker, "MultiK: A framework for orchestrating multiple specialized kernels," 2019, *arXiv:1903.06889*. [Online]. Available: https://arxiv.org/abs/1903.06889

[40] Z. Hua, D. Du, Y. Xia, H. Chen, and B. Zang, "EPTI: Efficient defence against meltdown attack for unpatched VMs," in *Proc. USENIX Annu. Tech. Conf.*, Jul. 2018, pp. 255–266. [Online]. Available: https://dl.acm.org/doi/10.5555/3277355.3277380

[41] S. Österlund, K. Koning, P. Olivier, A. Barbalace, H. Bos, and C. Giuffrida, "KMVX: Detecting kernel information leaks with multi-variant execution," in *Proc. 24th Int. Conf. Architectural Support Program. Lang. Operating Syst.*, Apr. 2019, pp. 559–572, doi: 10.1145/3297858.3304054.

[42] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum, "Fault isolation for device drivers," in *Proc. IEEE/IFIP Int. Conf. Dependable Syst. Netw.*, Jun. 2009, pp. 33–42, doi: 10.1109/DSN.2009.5270357.

[43] S. Butt, V. Ganapathy, M. M. Swift, and C.-C. Chang, "Protecting commodity operating system kernels from vulnerable device drivers," in *Proc. Annu. Comput. Secur. Appl. Conf.*, Dec. 2009, pp. 301–310, doi: 10.1109/ACSAC.2009.35.

[44] L. Tan, E. M. Chan, R. Farivar, N. Mallick, J. C. Carlyle, F. M. David, and R. H. Campbell, "IKernel: Isolating buggy and malicious device drivers using hardware virtualization support," in *Proc. 3rd IEEE Int. Symp. Dependable, Autonomic Secure Comput. (DASC)*, Sep. 2007, pp. 134–144, doi: 10.1109/DASC.2007.16.

[45] Y. Sun and T.-C. Chiueh, "SIDE: Isolated and efficient execution of unmodified device drivers," in *Proc. 43rd Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw. (DSN)*, Jun. 2013, pp. 1–12, doi: 10.1109/DSN.2013.6575348.

[46] A. Seshadri, M. Luk, N. Qu, and A. Perrig, "SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes," in *Proc. 21st ACM SIGOPS Symp. Operating Syst. Princ. (SOSP)*, 2007, pp. 335–350, doi: 10.1145/1294261.1294294.

[47] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig, "TrustVisor: Efficient TCB reduction and attestation," in *Proc. IEEE Symp. Secur. Privacy*, May 2010, pp. 143–158, doi: 10.1109/SP.2010.17.

[48] M. I. Sharif, W. Lee, W. Cui, and A. Lanzi, "Secure in-VM monitoring using hardware virtualization," in *Proc. 16th ACM Conf. Comput. Commun. Secur. (CCS)*, 2009, pp. 477–487, doi: 10.1145/1653662.1653720.

[49] L. Deng, P. Liu, J. Xu, P. Chen, and Q. Zeng, "Dancing with wolves: Towards practical event-driven VMM monitoring," in *Proc. 13th ACM SIGPLAN/SIGOPS Int. Conf. Virtual Execution Environments*, Apr. 2017, pp. 83–96, doi: 10.1145/3050748.3050750.

[50] L. Koromilas, G. Vasiliadis, E. Athanasopoulos, and S. Ioannidis, "GRIM: leveraging gpus for kernel integrity monitoring," in *Proc. 19th Int. Symp. Res. Attacks, Intrusions, Defenses*, Sep. 2016, pp. 3–23, doi: 10.1007/978-3-319-45719-2_1.

[51] G. Klein, M. Norrish, T. Sewell, H. Tuch, S. Winwood, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, and R. Kolanski, "SeL4: Formal verification of an OS kernel," in *Proc. ACM SIGOPS 22nd Symp. Operating Syst. Princ. (SOSP)*, 2009, pp. 207–220, doi: 10.1145/1629575.1629596.

[52] *Getting Started With Kmemcheck*. Accessed: May 21, 2019. [Online]. Available: https://www.kernel.org/doc/html/v4.14/dev-tools/kmemcheck.html

[53] *The Kernel Address Sanitizer (KASAN)*. Accessed: May 21, 2019. [Online]. Available: https://www.kernel.org/doc/html/latest/dev-tools/kasan.html

[54] *Syzkaller is an Unsupervised, Coverage-Guided Kernel Fuzzer*. Accessed: May 22, 2019. [Online]. Available: https://github.com/google/syzkaller/

HIROKI KUZUNO received the M.E. degree in information science from Nara Institute of Science and Technology, Japan, in 2007, and the Ph.D. degree in computer science from Okayama University, Japan, in 2020.

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 (Member, IEEE) received the 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 with Japan Society for the Promotion of Science. In 2002, he became a Research Associate with the Faculty of Information Science and Electrical Engineering, Kyushu University. Since 2005, he has been an Associate Professor with the Graduate School of Natural Science and Technology, Okayama University. Since 2021, he has been serving as a Professor with Okayama University. His research interests include operating systems and computer security. He is a member of IPSJ, IEICE, ACM, and USENIX.