MKM: Multiple Kernel Memory for Protecting Page Table Switching Mechanism Against Memory Corruption

Hiroki Kuzuno[1,2] and Toshihiro Yamauchi[1]

[1] Graduate School of Natural Science and Technology, Okayama University, Japan
[2] Intelligent Systems Laboratory, SECOM CO., LTD., Japan
kuzuno@s.okayama-u.ac.jp, yamauchi@cs.okayama-u.ac.jp

Abstract. Countermeasures against kernel vulnerability attacks on an operating system (OS) are highly important kernel features. Some kernels adopt several kernel protection methods such as mandatory access control, kernel address space layout randomization, control flow integrity, and kernel page table isolation; however, kernel vulnerabilities can still be exploited to execute attack codes and corrupt kernel memory. To accomplish this, adversaries subvert kernel protection methods and invoke these kernel codes to avoid administrator privileges restrictions and gain complete control of the target host. To prevent such subversion, we present Multiple Kernel Memory (MKM), which offers a novel security mechanism using an alternative design for kernel memory separation that was developed to reduce the kernel attack surface and mitigate the effects of illegal data manipulation in the kernel memory. The proposed MKM is capable of isolating kernel memory and dedicates the trampoline page table for a gateway of page table switching and the security page table for kernel protection methods.The MKM encloses the vulnerable kernel code in the kernel page table. The MKM mechanism achieves complete separation of the kernel code execution range of the virtual address space on each page table. It ensures that vulnerable kernel code does not interact with different page tables. Thus, the page table switching of the trampoline and the kernel protection methods of the security page tables are protected from vulnerable kernel code in other page tables. An evaluation of MKM indicates that it protects the kernel code and data on the trampoline and security page tables from an actual kernel vulnerabilities that lead to kernel memory corruption. In addition, the performance results show that the overhead is 0.020 μs to 0.5445 μs, in terms of the system call latency and the application overhead average is 196.27 μs to 6,685.73 μs , for each download access of 100,000 Hypertext Transfer Protocol sessions.

1 Introduction

Kernel vulnerability attacks are highly consequential and compromising processes in which an adversary takes control of the administrator account

through privilege escalation in the operating system (OS); to prevent this, the kernel must adopt protection methods.

Several kernel protection methods have been implemented, including stack monitoring of the kernel code [1], verification of kernel code control flow integration (CFI) to prevent the use of the return oriented programming (ROP) technique [2], system call isolation (SCI) for the creation of page table that contains necessity only kernel code at the system call invocation to prevent ROP attack selects an execution piece from the entire kernel code [3], kernel address space layout randomization (KASLR) to randomize the layout of the kernel code and data on the kernel memory [4], and the virtual address space separation method isolates the kernel memory to the kernel and the user page tables, to prevent meltdown side channel attacks from a user process (e.g., kernel page table isolation (KPTI) for Linux [5]). At the CPU layer, supervisor mode access prevention (SMAP) prevents access to the user memory region, whereas supervisor mode execution prevention (SMEP) prevents the execution of code in the user memory region of the virtual address space in the supervisor [6].

Privilege escalation leads to kernel vulnerability attacks; it employs an illegal memory corruption effect to overwrite privileged information variables on kernel memory. Although kernel protection methods restrict administrator privileges, the adversary also subverts kernel protection methods (e.g., achieving access control of SELinux [7]) to modify the kernel code and data on the kernel page table [8, 9]. Moreover, kernel memory observer (KMO) involves the segregation of specific kernel codes as dedicated page tables [10]. Although, kernel protection methods calling kernel codes (e.g., the page table switching) need to be assigned to same virtual address space with vulnerable kernel code, the placement of kernel protection methods calling is the remaining of kernel attack surface.

This poses a threat to the kernel protection methods at the kernel layer. To mitigate this and protect specific kernel codes against kernel vulnerability attack targets, another perspective is necessary to ensure that the environment supports the limitations of kernel vulnerability attacks, thereby reducing the damage caused to the target kernel memory region.

In this study, we proposed a novel security mechanism called "Multiple Kernel Memory (MKM)", which enhances the resistance capabilities of the kernel using multiple page tables, thereby mitigating kernel vulnerability attacks that subvert kernel protection methods and these kernel code calling placement. An overview of the proposed security mechanism is presented below:

- MKM introduces an additional boundary of kernel code execution involving two page tables: trampoline and security. The gateway of page table switching feature is assigned and executed on the trampoline, then the kernel protection methods are dedicated and executed on the security page tables. A majority of the kernel code is confined to the kernel page table, and the remainder of the kernel code is stored in the user page table.
- To reduce the kernel attack surface and achieve isolation of the kernel code accessible range, a separation of the virtual address space is required. The MKM mechanism realizes that the potentiality of vulnerable kernel code is enforced to execute to only the virtual address space of the kernel page

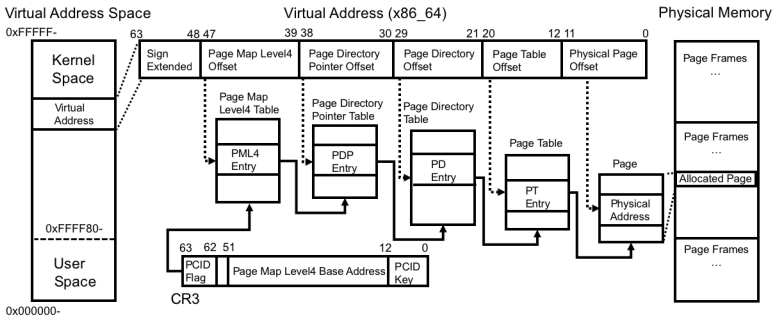

Fig. 1. Overview of page table structure (Linux x86_64) [11]

table. Thus, it eliminates the risk of memory corruption of the gateway of page table switching and the kernel protection methods that are stored on the trampoline and security page tables.

Here, the MKM was implemented on the Linux kernel using KPTI and SCI, and the kernel observation mechanism was executed. During evaluation, MKM mitigated the illegal modifications to the kernel code of the protection methods and page table switching function, and such instances were identified. The main contributions of the study are summarized below:

1. We present MKM, which is a novel kernel memory separation mechanism that is designed to specifically protect kernel protection methods at the kernel layer. We also discuss the threat model, capability, and limitation of MKM, which achieves resilience to kernel vulnerability attacks.
2. We evaluate the efficacy of the implementing MKM based on actual kernel vulnerability CVE-2017-16995 [12] Proof-of-Concepts (PoC) code attacks SELinux and the page table switching kernel code of the MKM. Both cases were identified via the kernel observation mechanism. The performance evaluation results indicate that the MKM overhead is from 0.020 μs to 0.5445 μs for each system call round time, and the application overhead average is from 196.27 μs to 6,685.73 μs for each HTTP download access.

2 Background

2.1 Virtual Address Space and Page Table

The design and implementation of the MKM was realized on a target architecture of is x86_64 and the operating system used is Linux kernel. The page table structure manages multiple tables and pages for handling the virtual address space (Figure 1). The length of the virtual address is 48 bits, the page size is 4 Kbytes, and CR3 register is the physical address of the page table in the Linux x86_64 architecture. The page table maintains the page entry mechanism, which assigns the relationships between the physical and virtual addresses of each page on the page table.

2.2 Kernel vulnerability attack

Kernel memory corruption through kernel vulnerability is a common type of attack [13]. The OS and CPU manages a privilege level to protect the kernel code or data, whereas KASLR / CFI provides hardening of the kernel against exploitation attacks from user processes, and SMAP / SMEP restricts malicious codes in the user memory region from the kernel mode execution.

Privilege escalation is the goal of a process that intends to compromise the kernel. It enables the reading and writing of kernel information by corrupting the kernel memory. To avoid OS and CPU protection methods, the adversary needs to execute the arbitrary program code to gain complete administrator privileges at the kernel layer by exploiting kernel vulnerabilities in the kernel.

To achieve privilege escalation on Linux, the adversary force-calls the kernel functions `commit_creds` and `prepare_kernel_cred` to gain access to root privileges. Moreover, the adversary directory overwrites the `uid` variable of the `cred` structure on the kernel memory. To subvert the Linux kernel protection methods, the adversary directory overwrites the Linux security module (LSM) function pointer value of the `security_hook_list` that invokes different non-checking access control methods in the kernel code or changes the security context variable to escape a mandatory access control (MAC) mechanism (e.g., SELinux) restrictions [8, 9].

2.3 Isolation of Virtual Address Space

The CPU and kernel protection mechanisms prohibit user processes from referring to the virtual address space of the kernel memory. The adversary has to escape these restrictions for the privilege escalation and the subversion of the kernel protection methods.

A meltdown side channel attack indicates that a user process can refer to virtual addresses of the kernel memory without the use of any kernel protection methods (e.g., KASLR). Therefore, an adversary uses this virtual address information to execute an arbitrary program for performing a kernel vulnerability attack, using the ROP technique. Moreover, the meltdown countermeasure method (e.g., Linux KPTI [5]) provides the page table to the user mode and kernel mode for virtual address space isolation.

Additionally, in the ROP countermeasure method (e.g., Linux SCI [3]), an independent page table for the kernel memory with minimum kernel codes is innovated from user process data to execute a system call. SCI limits the ROP technique and creates and executes a malicious code that concatenates code snippets from the complete kernel codes on kernel memory. Moreover, the KMO involves additional virtual address space isolation mechanism that provides security page table for the kernel mode. It is the dedicated memory region for the execution of the kernel protection method and segregation from the kernel page table [10].

2.4 Threat Model

Herein, we postulate that a threat model (i.e., an adversary) executing a kernel vulnerability attack corrupts the kernel memory only in the kernel mode. The adversary's goal is to execute any control of the OS kernel using administrator privileges. Although the adversary is a normal user, the adversary's user process changes the page table switching function on the kernel page table of the MKM, the LSM hook function pointer variable for disabling the MAC, and a credential variable through kernel vulnerability. It enables the adversary's user process to insert malicious code in the kernel memory. Consequently, the adversary gains complete administrator privileges on the OS kernel.

A limitation of a memory corruption is that the kernel vulnerability and the victim kernel code and data must be on the same virtual address space of the page table. This attack cannot overwrite other virtual address spaces of the page tables, which restricts access to the kernel and memory management unit.

3 Design and Implementation

3.1 Goal of Multiple Kernel Memory

The primary goal of the MKM is to prevent the subversion of kernel protection methods that keep to restrict access to complete administrator capabilities at the kernel layer.

- **Concept of protecting kernel protection methods**
  Kernel protection methods face threats on the kernel memory, because the adversary collapses kernel protection methods and bypasses accurate privilege-checking, which is performed for all user processes. To mitigate kernel memory corruption, it is essential to secure memory management and allocation for the execution of kernel code and for storing kernel data. It also necessary to segregate kernel protection methods and these kernel code invocation placement from vulnerable kernel code at the running kernel.

3.2 Challenge and Overview

To achieve the goal of MKM, which makes provision for the challenge of kernel resilience.

- **Securing of kernel protection methods**
  To automatically corresponds to the isolated processing of kernel features, kernel resilience is to manually assign the set of kernel protection methods code, page table function code, and other kernel code are on different virtual address spaces. This mechanism ensures that the kernel code is forcefully accessed and executed. Thereafter, kernel code could only cause a pollution of memory corruption to their virtual address space.

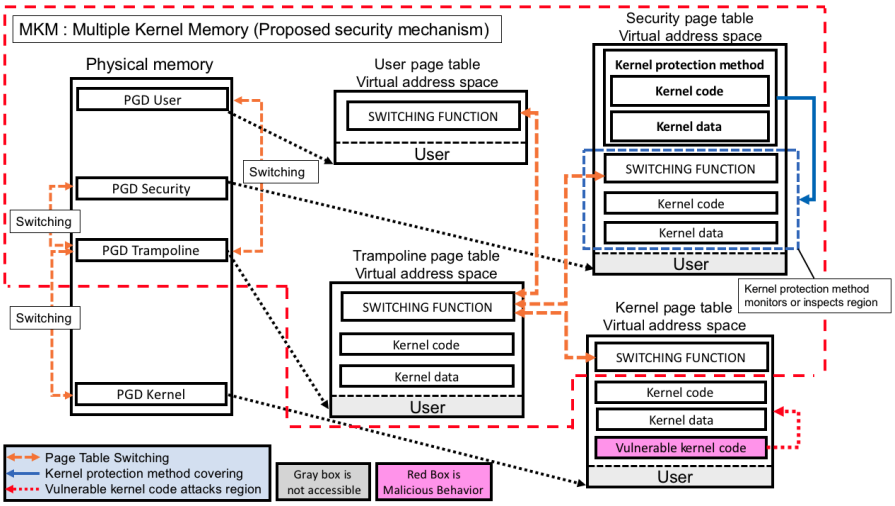

Fig. 2. Overview of multiple kernel memories.

To overcome this challenge, MKM (Figure 2) comprises two dedicated page tables: the trampoline page table and the security page table. The MKM kernel automatically switches to the page table that is suitable for the processing of specific kernel code of the kernel protection method in the kernel mode. The kernel protection method covers the kernel memory region of the security page table to achieve feature responsibilities. These page tables were derived using different memory architectures from the latest kernels such as Linux with KPTI possesses kernel and user page tables.

An overview of the role of the proposed page tables is provided below:

Trampoline: The trampoline page table acts as the gateway of the transition between the user mode and the kernel mode. It causes an invariably switch from other page tables to the trampoline page table. Moreover, it facilitates page table switching functions.

Security: The security page table supports kernel code and kernel data that constructs the features of kernel protection methods. These kernel codes are only executed on virtual address spaces of the security page table, thereby forcing acceptance of the in and out transition with the trampoline page table.

3.3 Page Table Switching Sequence

MKM involves three switching sequences between multiple page tables for processing each kernel feature (Figure 3). Additionally, MKM ensures that the trampoline page table is inserted to a middle position of each sequence, as described below:

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

Sequence 1 is a system call invocation or exception request that

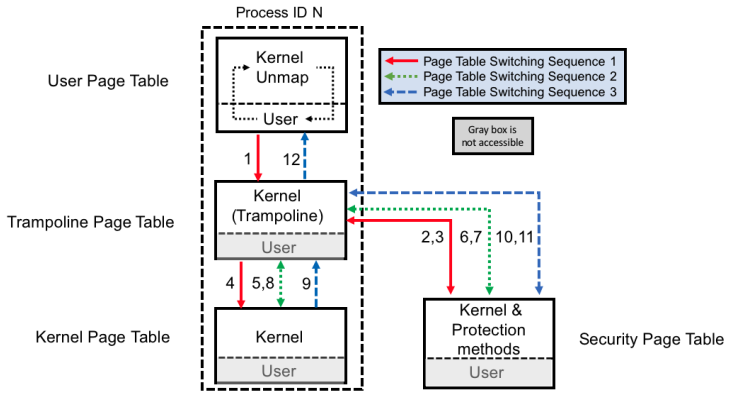

Fig. 3. Overview of page table switching sequences

triggers a transition from the user mode to the kernel mode. Simultaneously, it involves a change from the user to the kernel page tables, through the trampoline and security page tables. Thus, executing the kernel protection methods before the kernel feature deals with the request of user process.

Sequence 2: Kernel $\xrightarrow{5}$ Trampoline $\xrightarrow{6}$ Security $\xrightarrow{7}$ Trampoline $\xrightarrow{8}$ Kernel
Sequence 2 is the invocation of the kernel protection method invocation during kernel processing. It is the switch from the kernel page table to the security page table, through the trampoline page table.

Sequence 3: Kernel $\xrightarrow{9}$ Trampoline $\xrightarrow{10}$ Security $\xrightarrow{11}$ Trampoline $\xrightarrow{12}$ User
Sequence 3 is the return to the user mode from the kernel mode. It involves the switching from the kernel page table to the user page table, through the trampoline and security page tables. It executes kernel protection methods after the kernel features have completed the request of the user process.

3.4 Kernel Attack Surface

A kernel vulnerability attack can result in the corruption of the memory of other kernel code or data stored in the same virtual address space in the kernel mode.

The adversary uses vulnerable kernel code containing adversary-injected attack code disrupts the switching to the security page table. It intercepts the execution of the kernel protection method on the page table switching during sequence 2 and the sequence 3. This occurs during kernel processing and after the system call invocation. Sequence 1 remains unaffected because the page table switching function, kernel protection methods, and data are stored in the trampoline and security page tables. Thus, executing kernel protection method prior to the system call execution enables protection against attacks.

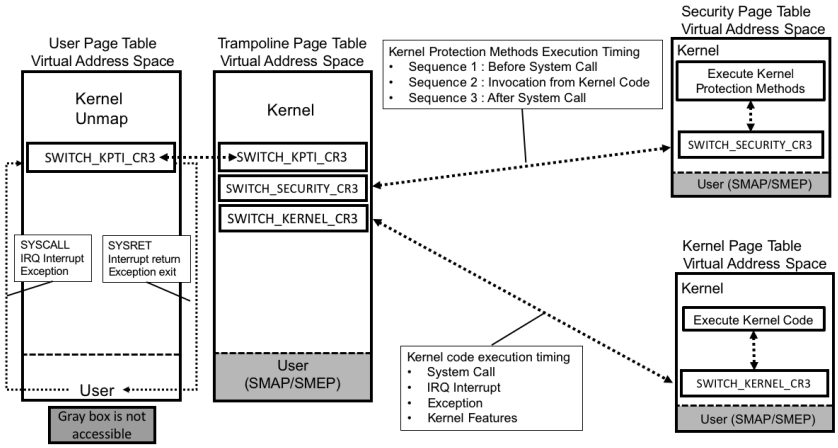

Fig. 4. Timing of switching page table

3.5 MKM Implementation

The proposed MKM was implemented on Linux using KPTI and SCI and the x86_64 CPU architecture.

Page Table Management

The MKM adopts KPTI and SCI, which have pre-assigned virtual address spaces, and assigns them as the user and kernel page tables, respectively, for kernel feature processing. The MKM introduces the trampoline and the security page tables that support the gateway of page table switching and kernel protection methods. Each process shares the security page table. In this study, the proposed kernel memory monitoring feature was executed on the security page table.

MKM employs a variable `pgd` of the structure `init_mm` as the initial value of the trampoline page table. The security page table adopts a four page size OR physical address from the variable `pgd`. The user page table utilizes a one page size XOR (4 Kbytes on x86_64) physical address from the variable `pgd`. MKM sets the kernel page table to the variable `kernel_pgd` of `mm_struct` of the `task_struct` structure.

Switching of Page Table

MKM selects a suitable page tables for the execution of kernel code for specific virtual address spaces (Figure 4). The implementation of switching sequences is described below:

Sequence 1: User $\xrightarrow{1}$ Trampoline $\xrightarrow{2}$ Security $\xrightarrow{3}$ Trampoline $\xrightarrow{4}$ Kernel
To transition from the user mode to the kernel mode. MKM uses the `SWITCH_KPTI_CR3` function, which writes the physical address of the trampoline page table to the CR3 register. MKM also employs `SWITCH_SECURITY_CR3` to write the physical address of the security page table to the CR3 register, and then back to the trampoline page table after executing the kernel protection methods. Moreover,

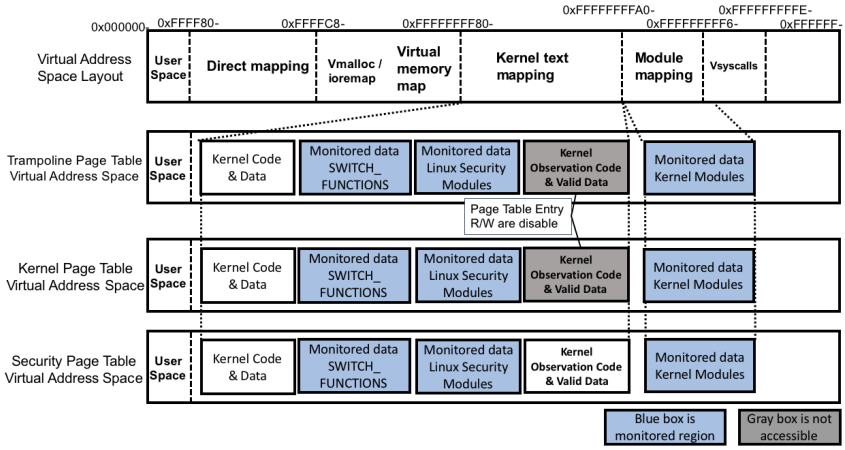

Fig. 5. Monitoring target of kernel code on the MKM

`SWITCH_KERNEL_CR3` writes a physical address of the kernel page table on the trampoline page table

Sequence 2: Kernel $\xrightarrow{5}$ Trampoline $\xrightarrow{6}$ Security $\xrightarrow{7}$ Trampoline $\xrightarrow{8}$ Kernel
During kernel processing, MKM utilizes `SWITCH_KERNEL_CR3` and `SWITCH_SECURITY_CR3` to switch to the security page table through the trampoline page table, for the execution of kernel protection methods.

Sequence 3: Kernel $\xrightarrow{9}$ Trampoline $\xrightarrow{10}$ Security $\xrightarrow{11}$ Trampoline $\xrightarrow{12}$ User
For the transition from the kernel mode to the user mode, MKM uses `SWITCH_KERNEL_CR3` to write the physical address of the trampoline page table to the CR3 register. Moreover, MKM also uses `SWITCH_SECURITY_CR3` and `SWITCH_KPTI_CR3` that to writes the user page table to the CR3 register through the security and trampoline page tables, for the execution of kernel protection methods.

Monitoring of Virtual Address Space

The kernel observation mechanism monitors the kernel module, LSM variables, and page table switching functions for MKM (Figure 5). To ensure accuracy of the monitoring data, the kernel observation mechanism identifies the virtual addresses of the target kernel code and data containing the `SWITCH_KERNEL_CR3` virtual address was specified on the kernel page table. Subsequently, it copies these monitoring data to the security page table as valid data, at the time of booting.

MKM enables timing the execution of the kernel protection method before and after the invocation of the system call and to interrupt kernel processing. The kernel observation mechanism involving MKM begins monitoring and compares the target data with the valid data on the security page table to determine if memory corruption has occurred.

Case Study of Page Table Switching Attack

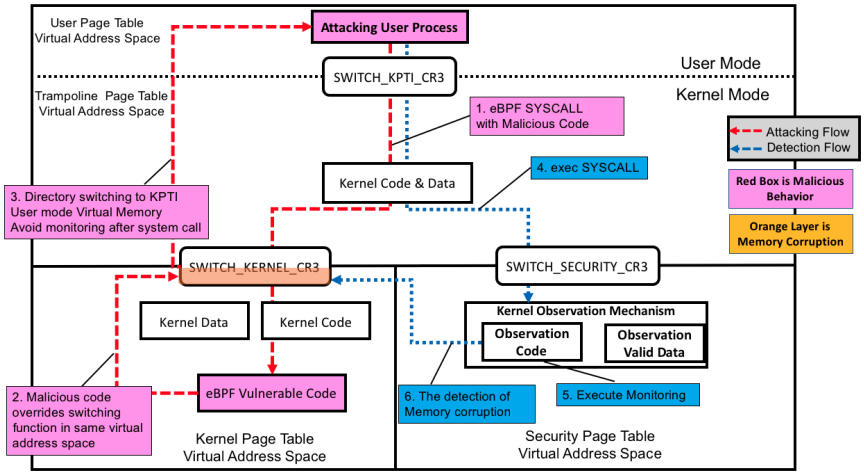

Fig. 6. Attack and detection flow of attacking user process using kernel vulnerability

The attack on the MKM kernel aims to completely disrupt the entire kernel protection method. In this study, the eBPF kernel vulnerability CVE-2017-16995 [12] PoC code that employs the `map_update_elem` function of `kernel/bpf/syscall.c` to exploit the kernel is considered. The adversary is able to write any restricted virtual address space of the kernel page table. It is considered that the attacking user process only succeeds in corrupting the `SWITCH_KERNEL_CR3` switching function of the kernel page table (Figure 6). This indicates that the MKM kernel directory switches from the kernel page table to the user page table.

4 Evaluation

4.1 Purpose and Environment of the Evaluation

The evaluation items and objectives are described as follows:

E1: Monitoring memory corruption of LSM with MKM
The identification and measurement times of the memory corruption overwriting the function pointer of LSM were evaluated. The kernel observation mechanism determined if the target memory region was valid, based on the security page table of MKM.

E2: Monitoring memory corruption of the page table switching feature
The identification and measurement times of the memory corruption of the page table switching function on the kernel page table were evaluated. The kernel observation mechanism preserves detection capability and then inspects whether the target memory region is valid after memory corruption.

E3: Measurement of system call invocation overhead
We measure the effect of kernel feasibility. A benchmark software is used to calculate the overhead of system call latency.

E4: Measurement of application overhead
We measure the performance overhead of a web application process using a benchmark software on MKM, which adopts several page tables switching.

The effectiveness of MKM was evaluated on the target implementation, Linux kernel 4.4.114 was used for monitoring and Linux kernel 5.0.0 was used for the performance evaluation. The Linux distribution used was Debian 9.0; the SCI was ported to kernel 4.4.114, and the CVE-2017-16995 PoC code was modified to handle any virtual addresses. The evaluation environment for stand alone and the server is a physical machine equipped with an Intel (R) Core (TM) i7-7700HQ (2.80 GHz, x86_64) processor and 16 GB DDR4 memory. The client machine is an Intel(R) Core(TM) i5 4200U (1.6 GHz), with 8 GB of memory and running Windows 10. The network environment for the application benchmark uses 1 Gbps hub supporting different ports for server and client physical machines.

4.2 Monitoring memory corruption of LSM with MKM

The eBPF kernel attack uses CVE-2017-16995 [12] to disable the LSM feature on Linux, it modifies the LSM hook function pointer of `selinux_hooks` to the virtual address of the original kernel module function at the `sys_bpf` system call invocation. MKM allows the kernel observation mechanism stores the valid data at the kernel boot and runs an inspection scheme before and after the system call invocation. It compares the target virtual address with the valid virtual address on the security page table; thereafter, it outputs the result to log messages.

On identifying memory corruption, log messages are presented as "Invalid LSM function is detected" and "Virtual Address (Invalid)." The kernel observation mechanism employing MKM accurately identifies the invalid LSM function pointer (Figure 7). The attack occurs on the virtual address space of the kernel page table; subsequently, MKM switches to the security page table through the trampoline page table, enabling the kernel protection method to detect and identify the actual attack within 0.0049 ms after the kernel executes the PoC kernel code.

4.3 Monitoring memory corruption of the page table switching

The eBPF kernel attack also uses CVE-2017-16995 [12] to modify the page table switching function pointer `SWITCH_KERNEL_CR3` to the virtual address of the kernel module function. It compromises sequence 3 of switching from the kernel to the trampoline page table. The kernel observation mechanism with MKM identifies the memory corruption by presenting the log messages of "Invalid vmem switching function is detected" and "Virtual Address (Invalid)."

The kernel observation mechanism with MKM also accurately identifies and shows that the attack overwrites the function pointer of `SWITCH_KERNEL_CR3` to the kernel module function pointer on the kernel page table (Figure 8).

After the attack, the MKM was unable to switch to the security page table from the kernel page table through the trampoline page table. Moreover, the

```
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 trampoline, security page table, and monitoring
8.  [  100.767961] Address ffffffff812f3f20 (Valid), ffffffff812f3f20 (Valid)
9.  // Switching to trampoline page table

10. // Switching to kernel page table
11. // CVE-2017-16995 attack overrides LSM function address via sys_bpf()
12. // Switching to trampoline page table

13. // Switching to security page table and monitoring after sys_bpf()
14. [  100.772834] Invalid lsm function is detected
15. [  100.772854] Address ffffffff812f3f20 (Valid), ffffffffa0000000 (Invalid)
16. // Switching to trampoline page table
```

Fig. 7. Monitoring result for the memory corruption of LSM function pointer

```
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
```

Fig. 8. Monitoring result for the memory corruption of page table switching function

MKM directory is changed to the user page table from the kernel page table. Although the inspection was successfully prevented after the attack, the other inspections prior to system call invocation remain unaffected and detected it within 0.0039 ms after the kernel executed the attack system call.

4.4 Measurement of System Call Invocation Overhead

We measured the performance overhead that compares the Linux kernel using the MKM mechanism and a vanilla Linux kernel. We executed the lmbench software ten times to determine the system call overhead effect from the average score. The result is the switching cost of the page tables for each system call invocation (Table 1). The result of lmbench contains different counts of system calls invoked for each system call. fork+/bin/sh has 54; fork+execve has four; fork+exit and open/close have two invocations; and the others have one invocation. Table 1 demonstrates that the system calls with the highest overhead are fork+exit (0.5445 μs, 100.91%) and the lowest overheads are write (0.020 μs, 109.05%).

Table 1. Overhead of MKM mechanism on the Linux kernel (μs)

System call	Vanilla kernel	MKM kernel	Overhead
fork+/bin/sh	517.839	524.383	6.544 (101.26%)
fork+execve	133.954	134.823	0.869 (100.65%)
fork+exit	120.214	121.303	1.089 (100.91%)
open/close	3.070	3.226	0.156 (105.08%)
read	0.264	0.285	0.021 (107.95%)
write	0.221	0.241	0.020 (109.05%)
stat	1.095	1.128	0.033 (103.01%)
fstat	0.286	0.306	0.020 (106.99%)

Table 2. ApacheBench overhead of MKM mechanism on the Linux kernel (μs).

File size (KB)	Vanilla kernel	MKM kernel	Overhead
1	1,637.08	1,833.35	196.27 (111.99%)
10	1,868.17	2,542.07	673.9 (136.07%)
100	3,709.58	1,0395.31	6,685.73 (280.23%)

4.5 Measurement of Application Overhead

We compared the user process overhead between the vanilla kernel and MKM kernel. The user process used here is the Apache 2.4.25 web server. The benchmark software is ApacheBench 2.4. The ApacheBench calculates a download request average of 100,000 HTTP accesses to file sizes of 1 KB, 10 KB, and 100 KB in one connection. Table 2 demonstrates that MKM has an average overhead of 196.27 μs (111.99%) to 6685,73 μs (280.23%) for each file download access of 100,000 HTTP sessions. ApacheBench relies on the total count of system call invocations in the user process. The ApacheBench result shows that a small file requires a low overhead factor whereas large files increase the overhead factor.

We consider that the file transfer depends on the number of system call invocations. Numerous system call invocations increase the switching of page tables and cause additional processing time.

5 Discussion

5.1 Evaluation Consideration

During the evaluation, our kernel observation mechanism used the MKM security page table. Although the eBPF kernel vulnerability attack successfully modified the page table switching function on the kernel page table, thereby disabling one of the MKM mechanisms, it did not affect the rest of the MKM mechanisms. The MKM continued to run our kernel observation mechanism on the security page table from the trampoline page table before the system call invocation at the kernel layer. Moreover, it is difficult for the adversary to evade the inspection timing before the system call invocation of the kernel observation mechanism, after the adversary program has already compromised the host.

5.2 Limitations

When an actual kernel vulnerability PoC (e.g., CVE-2017-16995) attempts to defeat part of the MKM mechanism, MKM can not invoke kernel protection methods after the memory corruption caused by the system call invocation.

To enhance kernel resilience and protect specific kernel code and data, MKM provides two dedicated page tables at the kernel layer to restrict the attack to the entire kernel memory. The MKM supports the kernel protection method and the normal feature kernel code assigned on each individual page table. Therefore, the MKM complicates the access to the virtual address spaces of trampoline and security page tables when the vulnerable kernel code affects its virtual address space. The MKM relies on the accomplishing memory isolation through page tables. Thus, kernel protection methods or the kernel driver should satisfy the requirements for creating suitable isolation granularity on the MKM mechanism.

5.3 Performance Consideration

We consider the performance overhead resulting from the MKM handling multiple page tables for executing suitable processes on the running kernel. MKM enables tag-based TLBs, which reduces the performance overhead. The Linux KPTI mechanism, SCI, and MKM use the PCID of TLB. The cache on TLBs improves the physical memory access without a page table walk to identify the targeted page, which only requires the CR3 update.

The performance effect involves the switching of page tables, followed by the CR3 register access. MKM maintains the application process without overhead in the user mode. An overhead cost appears when the switching occurs in the kernel mode after the system call invocation from the user mode. In addition, the kernel typically only switches the user page table at the context switch of each process, whereas KPTI switches between the kernel and user page tables at each transition between the user mode and the kernel mode. The trampoline and security kernel page tables provided by MKM require additional overheads to switch page tables in the kernel mode.

5.4 Portability Consideration

We consider the capability of the MKM mechanism on other OSs. FreeBSD, which adopts page table isolation [14], is an accepted Linux implementation similar to the MKM approach. Therefore, additional CPU architectures can be realized in the future, to validate the use of multiple kernel page tables.

6 Related Work

Kernel protection for privilege management. SELinux [7] and Capability [15] restrict the granularity of root privilege of user process, which reduces the harmful effects of a compromised host.

Kernel Protection for running kernel. Software based running kernel protection involves stack monitoring [1], randomization of kernel memory layout using KASLR [4], kernel control flow integrity [2], randomize the virtual address of the page table position [16], to mitigate a kernel attack with arbitrary program or ROP code snippets execution. In addition, kRˆX are exclusive management methods that directly protect kernel code and kernel image data on the kernel memory [17]. Moreover, hardware based running kernel protection adopts trusted computing base (TCB) verifies the firmware and kernel validation at start up and protects them using tamper-proof features [18]. Sprobes uses the CPU security features and provides a trusted execution environment [19].

Kernel monitoring. SecVisor and TrustVisor are hypervisors that monitor the kernel as the guest OS preserves the integrity of the kernel code and data [20, 21]. SIM inserts the monitoring mechanism into a guest OS memory space to achieve real-time kernel behavior inspection [22]. ED-Monitor is a kernel module that handles register management to enable hypervisor monitoring [23]. GRIM supports a kernel protection mechanism on the GPU device [24].

Kernel vulnerability suppression. The seL4 micro-kernel provides a small set of kernel-level functionality with formal verification of memory management to restrain memory invalidation and other vulnerability [25]. In addition, kernel memory fuzzing is the technique of discovering mis-implementations that result in vulnerabilities; kmemcheck [26] and KASAN [27] with syzbot and syzkaller [28] automatically inspect the memory handling processes on the kernel memory mechanism.

Reducing kernel memory attack surface. Separation of user and kernel memory using KPTI [5] and separation of the kernel memory using the extended page table are available on Intel CPUs [29]. KRAZOR reduces the visible kernel code list to the user process [30], KASR handles an execution permission at page granularity for user process [31], Additionally, PerspicuOS provides intra-kernel privilege separation to support isolation management at the kernel layer [32]. Multik prepares the minimum kernel code mapping of kernel memory for each application [33], and KMO provides the dedicated page table for executing specific kernel codes and data, to prevent failure of the kernel protection method [10]. Kernel multi-variant execution (kMVX) provides differential virtual address space and stack behaviors; anomalous process behavior is identified based on the success or failure of attacks in these environments [34].

The security features of MKM and previous research mechanisms were considered. Kernel protection for privilege management and running kernel provides effective kernel protection at each layer. Moreover, kernel monitoring using Hypervisor or a hardware layer (e.g., CPU) protection ensures integrity and enhances monitoring capability. By reducing the kernel memory attack surface, MKM provides an alternative attack mitigation method that separates or minimizes the kernel memory, to mitigate the attack code execution through kernel vulnerability and protect security features in the kernel memory mechanisms. Kernel vulnerability suppression automatically complements the additional kernel features. The fuzzing technique identifies mis-implementations

Table 3. Reducing kernel memory attack surface comparison (✓ is supported; △ is partially supported).

Feature	KRAZOR[30]	KASR [31]	PerspicuOS [32]	Multik [33]	KMO[10]	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				✓	△	△

of additional features and the formal verification approach subsequently ensures that anomalous control flow behavior is excluded. Kernel security capabilities are enhanced and restricted to realize multiple layer security quality using a combination of these research methods.

6.1 Comparison with Related Work

Based on a comparison of the security features of MKM and those of five reducing kernel attack mechanisms (Table 3) [30–33, 10] , MKM satisfies a majority of the attack mitigation requirements for the running kernel.

KRAZOR [30] initially collects necessary kernel features for a targeted program and subsequently enforces the result at the deployment phase during user process execution. Moreover, KASR [31] builds up a kernel code database during the offline training of the targeted program; thereafter, the executable kernel codes are employed for the execution of the user process from the hypervisor layer. Although KRAZOR forcibly minimizes callable kernel functions and KASR executes kernel functions at the running kernel to mitigate kernel attacks from user processes, these approach do not separate the kernel memory at each kernel feature.

PerspicuOS [32] supports isolation techniques of privilege for multiple kernel components. PerspicuOS ensures that nested kernel (trusted) contains a small part of kernel code and data, and the outer kernel (untrusted) contains the rest of the kernel with de-privileging. The nested kernel exclusively manages hardware privilege operations (e.g., MMU and CPU registers) for the protection of illegal memory corruption. Although MKM does not support hardware privilege deduction, MKM completely separates the virtual address space as a page table for the kernel code. We consider that MKM covers the entry gate of the Application Binary Interface (ABI) to a user process, and porting to other OSes at the kernel layer.

Multik [33] profiles the necessary kernel codes that are generated for a customized kernel image and then allocates a minimized kernel as the page table of each application. KMO [10] assigns specific portions of kernel codes to dedicated page tables for the isolation from complete kernel codes. Multik and KMO ensure that the independent page table is safeguarded from the effects of memory corruption. However, the invocation codes of the kernel protection method (e.g., page table switching function) are still assigned to the kernel page table with vulnerable kernel code.

These kernel memory layers attack mitigation approaches, similar to the capability of MKM. The MKM architecture focuses on strongly separating

specific kernel codes at the earliest stage of the kernel protection method, through the invocation of system calls or kernel feature processing. Additionally, MKM does not support the kernel code reducing method for the user process; it needs to be combined with the MKM approach to achieve a more flexible adjustment of the page table assignment.

7 Conclusion

An OS kernel should be able to mitigate various attacks that exploit the kernel vulnerabilities. In general, kernels adopt stack monitoring, CFI, KASLR, KPTI, and KMO to minimize the attack surface and prevent the kernel vulnerabilities from being attacked. However, adversaries could utilize privilege escalation to subvert kernel protection methods and these kernel codes invocation by executing arbitrary code and exploiting the vulnerabilities at the kernel layer.

In this study, a novel security mechanism Multiple Kernel Memory (MKM) is proposed to provide two dedicated page tables: the trampoline and security page tables. MKM encapsulates the vulnerable kernel code in the kernel page table; thereafter, it assigns the page table switching function and kernel protection methods in the trampoline and security page tables to reduce the potential kernel attack surface. It ensures that kernel protection methods and vulnerable kernel code are executed on different virtual address spaces; this improves the resilient against memory corruption because kernel attacks can not target to the trampoline and security page tables. The evaluation of Linux using MKM could prevent the memory corruption of kernel protection methods. Additionally, our kernel observation mechanism works on the MKM implementation, which detects the memory corruption of the LSM hook function and page table switching function. Based on the performance evaluation, the overhead was 0.020 μs to 0.5445 μs for each system call invocation on the proposed kernel; moreover, the web client program overhead average for MKM was 196.27 μs to 6,685.73 μs,for each download access of 100,000 HTTP sessions.

Acknowledgment

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

References

1. Kemerlis, P, V., Portokalidis, G. and Keromytis, D, A.: kGuard: lightweight kernel protection against return-to-user attacks. In: Proceedings of the 21st USENIX conference on Security symposium, USENIX (2012). https://dl.acm.org/doi/10.5555/2362793.2362832
2. Abadi, M., Budiu, M., Erlingsson, U., Ligatti, J.: Control-flow integrity principles, implementations. In: Proceedings of the 12th ACM Conference on Computer and Communications Security, pp. 340-353, ACM (2005). https://doi.org/10.1145/1609956.1609960

3. Rapoport, M.: x86: introduce system calls address space isolation. https://lwn.net/Articles/786894/. Accessed 22 May 2019
4. Hund, R., Willems, C., Holz, T.: Practical timing side channel attacks against kernel space ASLR. In: Proceedings of the 2013 IEEE Symposium on Security and Privacy, pp. 191–205, IEEE (2013). https://doi.org/10.1109/SP.2013.23
5. Gruss, D., Lipp, M., Schwarz, M., Fellner, R., Maurice, C., Mangard, S.: KASLR is dead : long live KASLR, In: Bodden E., Payer M., Athanasopoulos E. (eds) ESSoS 2017. LNCS, vol. 10379, pp. 161-176, Springer, Cham (2017). https://doi.org/10.1007/978-3-319-62105-0_11
6. Mulnix D.: Intel® Xeon® Processor D Product Family Technical Overview, https://software.intel.com/en-us/articles/intel-xeon-processor-d-product-family-technical-overview. Accessed 10 August 2018
7. Security-enhanced Linux. http://www.nsa.gov/research/selinux/. Accessed 22 May 2019
8. Exploit Database, Nexus 5 Android 5.0 - Privilege Escalation. https://www.exploit-db.com/exploits/35711/. Accessed 21 May 2019
9. grsecurity: super fun 2.6.30+/RHEL5 2.6.18 local kernel exploit. https://grsecurity.net/~spender/exploits/exploit2.txt. Accessed 21 May 2019
10. Kuzuno, H., Yamauchi, T.: KMO: kernel memory observer to identify memory corruption by secret inspection mechanism. In: Heng SH., Lopez J. (eds) ISPEC 2019. LNCS, vol. 11879, pp. 75-94, Springer, Cham (2019). https://doi.org/10.1007/978-3-030-34339-2_5
11. Bovet, P, D., Cesati, M.: Understanding the Linux kernel, 3rd edition. O'Reilly Media, (2005).
12. CVE-2017-16995. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-16995. Accessed 10 June 2019
13. Chen, H., Mao, Y., Wang, X., Zhow, D., Zeldovich, N., Kaashoek, F, M.: Linux kernel vulnerabilities - state-of-the-art defenses and open problems. In: Proceedings of the Second Asia-Pacific Workshop on Systems, pp. 1-5, ACM (2011). https://doi.org/10.1145/2103799.2103805
14. Tetlow, G.: Response to Meltdown and Spectre. https://lists.freebsd.org/pipermail/freebsd-security/2018-January/009719.html. Accessed 21 May 2019
15. Linden, A. T.: Operating system structures to support security and reliable software. ACM Computing Surveys, vol. 8, no. 4, pp. 409–445. ACM (1976). https://doi.org/10.1145/356678.356682
16. Davi, L., Gens, D., Liebchen, C., Sadeghi, A.-R.: PT-Rand: practical mitigation of data-only attacks against page tables. In: Proceedings of the 23th Network and Distributed System Security Symposium, Internet Society (2016).
17. Pomonis, M., Petsios, T.: kRˆX: comprehensive kernel protection against just-in-time code reuse. In: Protection of the twelfth European Conference on Computer Systems, pp. 420-436, ACM (2017). https://doi.org/10.1145/3064176.3064216
18. Trusted computing group. tpm main specification. http://www.trustedcomputinggroup.org/resources/tpm_main_specification. Accessed 10 August 2018
19. Ge, X., Vijayakumar, H., Jaeger, T.: Sprobes: enforcing kernel code integrity on the trustzone architecture. In: Proceedings of the third Workshop on Mobile Security Technologies, ACM (2014).
20. Seshadri, A., Luk, M., Qu, N., Perrig, A.: SecVisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In: Proceedings of the 21st ACM

SIGOPS symposium on Operating systems principles, pp. 335-350, ACM (2007). https://doi.org/10.1145/1294261.1294294
21. McCune, M, J., Li, Y., Qu, Z., Zhou, A., Datta, V., Gligor, D., Perrig A.: TrustVisor: efficient tcb reduction and attestation. In: Proceedings of the 2010 IEEE Symposium on Security and Privacy, pp. 143–158, IEEE (2010). https://doi.org/10.1109/SP.2010.17
22. Sharif, I, M., Lee, W., Cui, W., Lanzi, A.: Secure in-VM monitoring using hardware virtualization. In: Proceedings of the 16th ACM Conference on Computer and Communications Security, pp. 477–487, ACM (2009). https://doi.org/10.1145/1653662.1653720
23. Deng, L., Liu, P., Xu, J., Chen, P., Zeng, Q.: Dancing with Wolves: towards practical event-driven VMM monitoring. In: Proceedings of the 13th ACM SIGPLAN / SIGOPS International Conference, pp. 83-96, ACM (2017). https://doi.org/10.1145/3050748.3050750
24. Koromilas, L., Vasiliadis, G., Athanasopoulos, E., Ioannidis, S.: GRIM: leveraging gpus for kernel integrity monitoring. In: Monrose F., Dacier M., Blanc G., Garcia-Alfaro J. (eds) RAID 2016. LNCS, vol. 9854, pp. 3-23. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-45719-2_1
25. Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: seL4: formal verification of an OS kernel. In: Proceedings of the 22nd ACM Symposium on Operating Systems Principles, pp. 207–220, ACM (2009). https://doi.org/10.1145/1629575.1629596
26. Getting started with kmemcheck. https://www.kernel.org/doc/dev-tools/kmemcheck.html. Accessed 21 May 2019
27. The Kernel Address Sanitizer (KASAN). https://www.kernel.org/doc/dev-tools/kasan.html Accessed 21 May 2019
28. syzkaller is an unsupervised, coverage-guided kernel fuzzer. https://github.com/google/syzkaller/. Accessed 22 May 2019
29. Hua. Z., Du, D., Xia, Y., Chen, H., Zang, B.: EPTI: efficient defence against meltdown attack for unpatched VMs. In: Proceedings of the 2018 USENIX Annual Technical Conference, pp. 255–266, USENIX (2018). https://dl.acm.org/doi/10.5555/3277355.3277380
30. Kurmus, A., Dechand, S., Kapitza, R.: Quantifiable Run-Time Kernel Attack Surface Reduction. In: Dietrich S. (eds) DIMVA 2014. LNCS, vol. 8550, pp. 212-234, Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08509-8_12
31. Zhang, Z., Cheng, Y., Nepal, S., Liu, D., Shen, Q., Rabhi, F.: KASR: a reliable and practical approach to attack surface reduction of commodity os kernels. In: Bailey M., Holz T., Stamatogiannakis M., Ioannidis S. (eds) RAID 2018. LNCS, vol 11050. pp. 691-710, Springer, Cham (2018). https://doi.org/10.1007/978-3-030-00470-5_32
32. Dautenhahn, N., Kasampalis, T., Dietz, W., Criswell, J., Adve, V.: Nested Kernel: an operating system architecture for intra-kernel privilege separation. In: Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 191-206, ACM (2015). https://doi.org/10.1145/2694344.2694386
33. Kuo, H, C., Gunasekaran, A., Jang, Y., Mohan, S., Bobba, B, R., Lie, D., Walker, J.: MultiK: a framework for orchestrating multiple specialized kernels. https://arxiv.org/abs/1903.06889v1. Accessed 16 May 2019
34. Österlund, S., Koning, K., Olivier, P., Barbalace, A., Bos, H., Giuffrida, C.: kMVX: detecting kernel information leaks with multi-variant execution. In: Proceedings

of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 559-572, ACM (2019). https://doi.org/10.1145/3297858.3304054