Protection Mechanism of Kernel Data Using Memory Protection Key

Kuzuno, Hiroki

Yamauchi, Toshihiro

(Citation)
IEICE Transactions on Information and Systems,E106.D(9):1326-1338

(Issue Date)
2023-09-01

(Resource Type)
journal article

(Version)
Version of Record

(URL)
https://hdl.handle.net/20.500.14094/0100485462

IEICE TRANS. INF. & SYST., VOL.E106–D, NO.9 SEPTEMBER 2023

PAPER Special Section on Information and Communication System Security

Protection Mechanism of Kernel Data Using Memory Protection Key*

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

SUMMARY Memory corruption can modify the kernel data of an operating system kernel through exploiting kernel vulnerabilities that allow privilege escalation and defeats security mechanisms. To prevent memory corruption, the several security mechanisms are proposed. Kernel address space layout randomization randomizes the virtual address layout of the kernel. The kernel control flow integrity verifies the order of invoking kernel codes. The additional kernel observer focuses on the unintended privilege modifications. However, illegal writing of kernel data is not prevented by these existing security mechanisms. Therefore, an adversary can achieve the privilege escalation and the defeat of security mechanisms. This study proposes a kernel data protection mechanism (KDPM), which is a novel security design that restricts the writing of specific kernel data. The KDPM adopts a memory protection key (MPK) to control the write restriction of kernel data. The KDPM with the MPK ensures that the writing of privileged information for user processes and the writing of kernel data related to the mandatory access control. These are dynamically restricted during the invocation of specific system calls and the execution of specific kernel codes. Further, the KDPM is implemented on the latest Linux with an MPK emulator. The evaluation results indicate the possibility of preventing the illegal writing of kernel data. The KDPM showed an acceptable performance cost, measured by the overhead, which was from 2.96% to 9.01% of system call invocations, whereas the performance load on the MPK operations was 22.1 ns to 1347.9 ns. Additionally, the KDPM requires 137 to 176 instructions for its implementations.

key words: *memory corruption, memory protection, system security, operating system*

1. Introduction

The operating system (OS) kernel encounters threats, in which privileges may be escalated and security mechanisms may be defeated. The user process of the adversary exploits the kernel code containing vulnerabilities (i.e., vulnerable kernel code), thereby corrupting the memory. Kernel vulnerability is reported by the Common Vulnerabilities and Exposures (CVE) [1]. The CVE has the Common Platform Enumeration (CPE) that indicates the naming of systems, software, and packages [2]. Until August 2022, 724 CVE with CPE of Linux (e.g., `linux_kernel`) and CVE description containing specific words (e.g., "memory") were issued in the CVE of National Vulnerability Database (NVD) [3], as shown in Fig. 1.

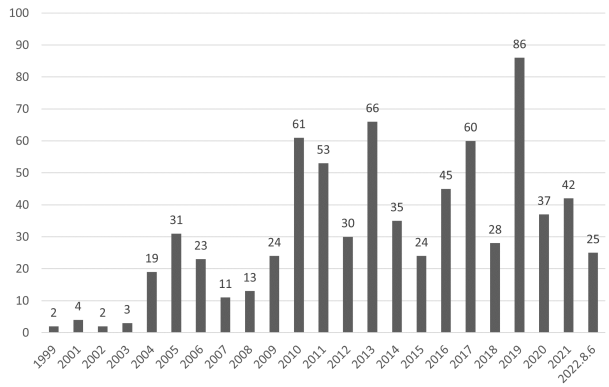

Fig. 1 Statistics of CVE for Linux kernel memory until August 2022 [3]

Therefore, privileged information can be modified and the kernel data of the security mechanism can be altered to gain full administrator privileges. Actual kernel memory corruption cases indicate that modifying the kernel data related to mandatory access control (MAC), the user acquires administrator privileges and circumvents the MAC restrictions [5], [6].

The following are the countermeasures that can prevent kernel attacks via vulnerable kernel code. Kernel control flow integrity (KCoFI) inspects the order of code execution [7] to restrict the kernel code from being illegally invoked [8]. Kernel address space layout randomization (KASLR) randomizes the virtual addresses of the kernel code and kernel data in the kernel memory space to foil attacks [9], whereas the additional kernel observer (AKO) detects unintentional rewriting in response to the changes in the privileged information of user processes against a privilege escalation attack [10].

Research Question. These mitigate the illegal modification of kernel data via kernel vulnerabilities. However, if the kernel memory is successfully corrupted, kernel data can be overwritten. Thus, this paper considers the following: *A running kernel does not restrict the writing of kernel data in the kernel mode. Existing approaches do not control the write restrictions of kernel data related to privileged information and security mechanisms. Therefore, an adversary can gain full administrator privileges.*

Research Contribution. This paper describes a kernel data protection mechanism (KDPM), which is a novel security capability that dynamically controls the write restric-

Manuscript received November 9, 2022.
Manuscript revised March 31, 2023.
Manuscript publicized June 30, 2023.

[†]The author is with Graduate School of Engineering, Kobe University, Kobe-shi, 657–8501 Japan.
[††]The author is with Faculty of Environmental, Life, Natural Science and Technology, Okayama University, Okayama-shi, 700–8530 Japan.

*This paper is an extended version of a paper published in the 16th International Workshop on Security (IWSEC) 2022 [4].

a) E-mail: kuzuno@port.kobe-u.ac.jp
b) E-mail: yamauchi@okayama-u.ac.jp

DOI: 10.1587/transinf.2022ICP0013

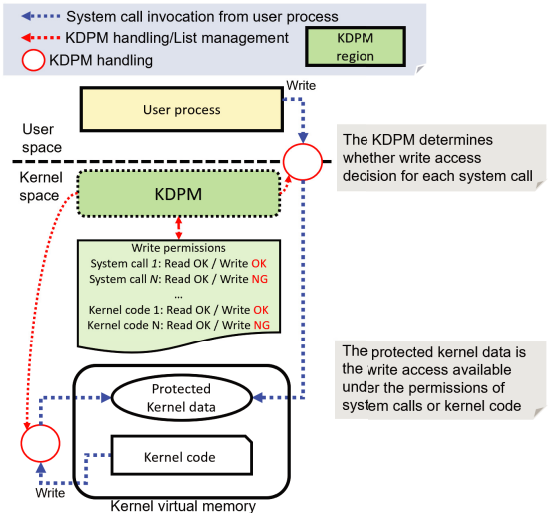

Fig. 2 Overview of the kernel data protection mechanism

tions of specific kernel data as protected kernel data. Figure 2 provides an overview of the KDPM, which determines whether system calls and kernel codes have write permission of the kernel data in the kernel layer. To ensure kernel data protection and manage write restrictions, the KDPM adopts the Intel memory protection key (MPK), which is a protection keys for supervisor (PKS). A PKS provides a protection key that handles write restrictions for each page of kernel data.

Research Objectives. The KDPM assumes that the user process of an adversary invokes a vulnerable kernel code that attempts to modify the kernel data related to privileged information or security mechanism. The KDPM focuses on the mitigation of the illegal overwrite of these kernel data. The privileged information is changed by specific system calls and the policy of MAC is modified by specific kernel codes. Moreover, the function pointers of the MAC are never modified at the running kernel. The KDPM provides a straightforward application of the PKS to maintain simple design of the kernel data protection. Additionally, the KDPM combines the characteristics of system calls, kernel code behavior, and hardware features. The limitation of the KDPM is its inability to support frequently modified kernel data.

Implementations. The KDPM has two implementations that focus on the different types of kernel attacks. Implementation 1 is a general purpose implementation for the protection of privileged information to prevent privilege escalation. This allows user processes to write to protected kernel data only when write-permitted system calls are invoked. Implementation 2 protects the kernel data of the security mechanism (e.g., MAC) from the defeating of security mechanism. This reduces overheads to limit the write restriction timing of protected kernel data. Further, Implementation 2 allows the protected kernel data to be written only when executing a write-permitted kernel code.

- **Implementation 1**: To prevent a privilege escalation attack, Implementation 1 controls the write restriction of privileged information in each write-permitted system call to protect the privileged information of user processes.
- **Implementation 2**: Implementation 2 controls the write restriction of the kernel data related to the security mechanism in each write-permitted kernel code to prevent the defeating by security mechanism attack.

Summary of Contributions. This study is an early application of the forthcoming PKS to protect kernel data. Intel CPUs containing a PKS are not available as of October 2022 and will be implemented on the next generation CPUs; however, a PKS is available in the QEMU environment [12]. The following are the contributions of this study:

1. The proposed KDPM is a novel approach that protects the kernel data in the running kernel to prevent privilege escalation and defeat by security mechanism attacks through vulnerable kernel code. The implementations of the latest Linux kernel use a PKS to handle the write restriction of the kernel code during a specific system call or specific kernel code execution.
2. The security capability evaluation indicates that the kernel with Implementation 1 can prevent the modification of privileged information by the adversary's user process. Additionally, the kernel with Implementation 2 can prevent the defeat of security mechanisms. The overhead of Implementation 1 requires latency of system call ranging from 2.96% to 9.01%, and the processing time for the kernel with Implementation 2 for writing the PKS is 22.1 ns. Furthermore, reading the register operation requires 30.5 ns, and writing the register operation requires 1347.9 ns. Additionally, Implementation 1 requires 176 instructions and Implementation 2 requires 137 instructions.

2. Background

2.1 Memory Protection Key

Intel CPU provides an MPK, which is a security feature provided to control read and write restrictions on a page basis, that is, page table entry (PTE) [13]. The MPK includes protection keys for userspace (PKU) and the protection key right for user mode register (hereinafter, PKRU) for the user mode. In addition, the MPK includes PKS and `IA32_PKRS_MSR` register (hereinafter, PKRS) for the kernel mode.

As shown in Fig. 3, the PTE has 16 4-bit protection keys (Pkeys), and the 32-bit flag (two bits per Pkey: write disable (WD) and access disable (AD)) controls the read and write restriction for each Pkey. The read and write restriction for Pkey i ($0 \le i \le 15$) is performed via the register. If the value of bit AD $i \times 2$ is 0, read is allowed. In contrast, if the value of bit AD $i \times 2$ is 1, read is not allowed. Additionally, if the value of bit WD $i \times 2 + 1$ is 0, write is allowed;

Table 1 Top CWE of kernel memory vulnerability [3]

Type	Content	CVE	PoC
CWE-200	Exposure of Sensitive Information to an Unauthorized Actor	125	6
NVD-CWE-Other	Other	87	5
CWE-119	Improper Restriction of Operations within the Bounds of a Memory Buffer	82	3
CWE-401	Missing Release of Memory after Effective Lifetime	78	0
CWE-399	Resource Management Errors	52	1
CWE-20	Improper Input Validation	44	1
CWE-787	Out-of-bounds Write	31	1
CWE-416	Use After Free	20	0
CWE-125	Out-of-bounds Read	19	1
CWE-362	Concurrent Execution using Shared Resource ('Race Condition')	18	3
CWE-189	Numeric Errors	17	0
CWE-264	Permissions, Privileges, and Access Controls	17	2
CWE-190	Integer Overflow or Wraparound	16	0
CWE-909	Missing Initialization of Resource	14	0
CWE-400	Uncontrolled Resource Consumption	12	0
CWE-772	Missing Release of Resource after Effective Lifetime	11	0
NVD-CWE-noinfo	Insufficient Information	11	1
Other CWE	Under 10 CVE	70	3
Total		724	27

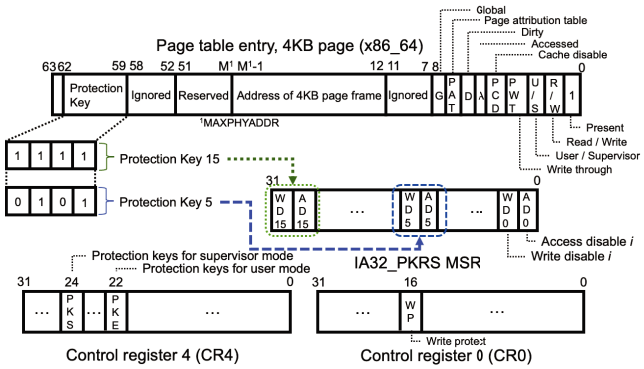

Fig. 3 Intel memory protection key [13]

Table 2 Executable PoC code for Linux kernel memory vulnerability list (✓ is protection available;)

CVE ID	CWE	Description	KDPM
CVE-2016-4997 [15]	CWE-264	Boundary check error in setsockopt function	✓
CVE-2016-9793 [16]	CWE-119	Boundary check error in net/core/sock.c	✓
CVE-2017-16995 [17]	CWE-119	Boundary check error in kernel/bpf/verifier.c	✓
CVE-2017-1000112 [18]	CWE-362	Race condition in net/ipv4/ip_output.c	✓

Table 3 Comparison of between page-level write protection and MPK (✓ is supported;)

		Page-level write protection	MPK
Types of restriction	Read		✓
	Write	✓	✓
Granularity of restriction	Page Table	✓	
	Pkey		✓
Control register		CR0 WP	PKRU/PKRS

and if the value of bit WD $i \times 2 + 1$ is 1, write is not allowed.

In MPK, read and write limits can be controlled separately in the specific register (e.g., PKRU and PKRS). For multiple PTEs, the read and write restriction can be controlled by specifying the Pkey.

2.2 Comparison of Memory Access Rights

Intel CPU also provides page-level write protection. Table 3 summarizes the comparison of memory access rights between the page-level write protection and MPK. The page-level write protection manages the read-only access and read/write (R/W) access rights for each page using the R/W flag and the write protect (WP) flag of control register (CR) 0. MPK manages read/write access rights for multiple pages using Pkey and PKRU/PKRS. The advantage of MPK that supports read restriction and fine-grain access control with Pkeys is that it can make multiple groups of memory access rights for management of data types. In addition, PTE updating requires the Translation Lookaside Buffer (TLB) flush and potentially occurs TLB miss-hit (hereafter, TLB costs). For memory access rights control, the page-level write protection requires TLB costs, however, MPK takes only the PKRS updating without TLB costs except for Pkeys changing.

2.3 Kernel Vulnerability

Kernel vulnerabilities are improper implementations that lead to kernel attacks [14]. Table 1 shows the Common Weakness Enumeration (CWE) ranking for CWE of kernel memory vulnerability. CWE represents the categorized software and hardware weakness type. Table 1 indicates a running kernel can be damaged in a variety of ways through a kernel vulnerability attack. Additionally, Proof of Concept (PoC) codes that achieve the Linux kernel compromising for specific CVE. Table 1 indicates 27 PoC codes of kernel memory vulnerability are available in the Exploit Database [11].

To investigate which CVE are reproducible, PoC codes are introduced from CWE ranking (Table 1). Table 2 shows the result of four PoC codes are reproducible and can compromise the environment for Linux kernel. Three PoC codes forcibly invokes kernel codes that modify privileged information to achieve the privilege escalation [15], [16], [18]. One PoC code overwrites the variable `cred` of the kernel

data that stores privileged information from the normal user to the administrator [17]. The defeat of the MAC forcefully modifies the list of function pointers that manage the access control decisions in the kernel. Meanwhile, the variable `selinux_hooks`, which stores function pointers, is modified to the inserted kernel codes that bypass the access control [5], [6].

Therefore, the combination of privilege escalation and the MAC being disabled provides full administrator capability to the adversary with no restrictions on the kernel.

3. Threat Model

3.1 Environment

This section highlights the assumed a threat model for the KDPM. The adversary acquires administrator privileges and disables the MAC in the target environment as follows:

- Adversary: An adversary gains normal user privileges, attempts privilege escalation, and defeats the MAC via the PoC code that exploits kernel vulnerabilities.
- Kernel: A kernel contains kernel vulnerabilities that can be exploited for privilege escalation and defeating the MAC. Existing security mechanisms (e.g., KCoFI, KASLR, and AKO) are not applied.
- Kernel vulnerability: A kernel vulnerability is the presence of a vulnerable kernel code that exploits kernel memory corruption.
- Attack targets: Attack targets are kernel data related to privileged information of user process (e.g., user id) and kernel data of the MAC (e.g., function pointers and access policies).

3.2 Scenario

The adversary induces the attack that executes the PoC code as the user process exploits the vulnerable kernel code. The following are the details of an attack:

1. Privilege escalation attack
   The user process of the adversary forcefully rewrites user privileges to gain administrator privileges for attaining full control of the computer.
2. Defeating security mechanisms
   The user process of the adversary forcefully disables the MAC by replacing the function pointer of the kernel code with one that does not make access decisions.

4. Design

4.1 Requirement

To manage write restrictions on specified kernel data, the KDPM should satisfy the following requirement:

- **Requirement**: Prevent privilege escalation and defeat of security mechanisms by illegally modifying kernel data via kernel vulnerabilities. The kernel must control the write restrictions of kernel data for specific system calls and kernel codes on the running kernel. The kernel data can be written only when system calls are invoked and the authorized kernel codes are executed.

4.2 Design Overview

KDPM fulfills the requirement for kernel data protection from the invocation of vulnerable kernel codes. Figure 4

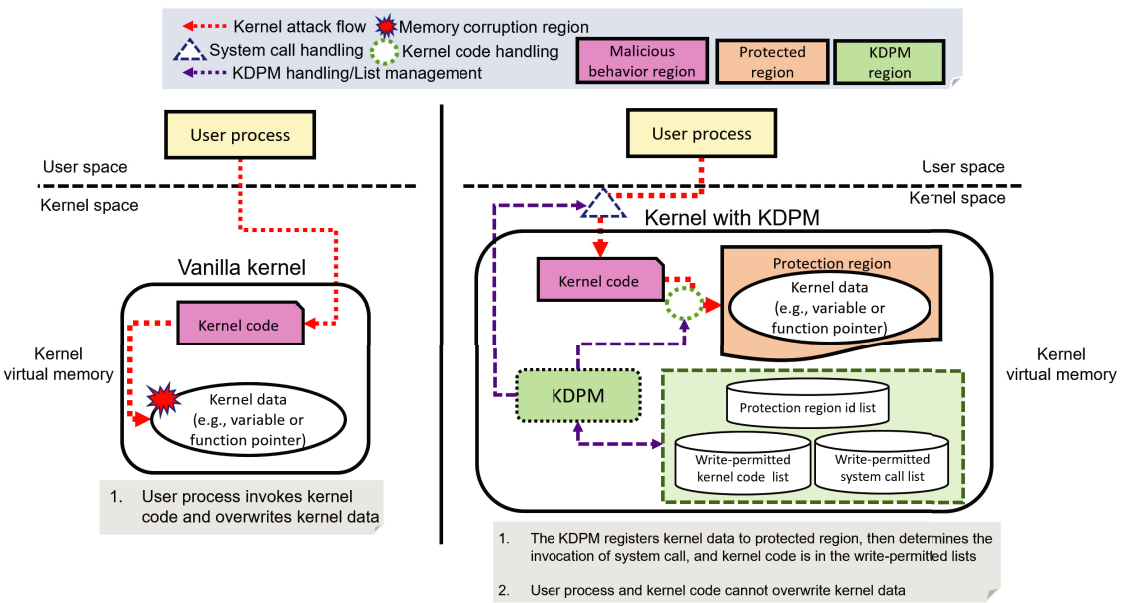

Fig. 4 Design overview of the KDPM

outlines the design overview of KDPM that introduces the protected kernel data, identifier list, the write-permitted system call list and write-permitted kernel code list.

KDPM manages the linking of the identifier that indicates the relationship between the protected kernel data, write-permitted system call, and write-permitted kernel code. KDPM statically registers protected kernel data, identifiers of protected kernel data, write-permitted system call, and write-permitted kernel code lists at the source code. After that, KDPM dynamically controls the write restriction of kernel data using identifier at the invocation of system call or kernel code for the running kernel.

4.3 Approach

The KDPM supports specific kernel data as protected kernel data (e.g., variable or function pointer) and the identifier to handle the write restrictions that manages the write-permitted system calls and write-permitted kernel code.

4.3.1 Protected Kernel Data

The following are the definitions of protected kernel data and identifiers:

- Protected kernel data: The kernel data of the user process (e.g., privileged information) and security mechanisms (e.g., the function pointer and access policy).
- Identifier: The identifier is used to set the write restrictions of the protected kernel data. For controlling the write restriction, the identifier is associated with the protected kernel data, write-permitted system call, and write-permitted kernel code.

The kernel with the KDPM provides a list of protected kernel data and corresponding identifiers in advance at the time of booting. Additionally, the kernel data for each user process generation is assumed to be protected.

4.3.2 Handling of Write Restrictions:

The KDPM handles the write restrictions of the protected kernel data using specific system calls and kernel codes. The KDPM defines and manages the following:

- Write-permitted system call: A system call has write permission for the protected kernel data.
- Write-permitted kernel code: The kernel code is authorized to write to the protected kernel data.

The KDPM disables write restrictions when a write-permitted system call is issued or write-permitted kernel code is executed. At the end of the write-permitted system call or write-permitted kernel code execution, the KDPM enables the write restriction to the protected kernel data.

5. Implementation

In this study, the KDPM is implemented on Linux with the x86_64 CPU architecture. Table 4 presents the protected kernel data and write control timing according to the implementations. The following are the implementation details:

Table 4 Comparison of the implementations of the KDPM

Item	Implementation 1	Implementation 2
Protected kernel data	Privilege information	Function pointer & Access policy
Handling	System call	Kernel code
Mitigation	Privilege escalation	MAC defeating
Performance	High	Low

- Implementation 1: This manages the protected kernel data containing privileged information and write-permitted system calls that change the privileges of the user process. Even if a user process attempts a privilege escalation, the privilege information cannot be written during the execution of another system call.
- Implementation 2: This manages protected kernel data related to the MAC (e.g., the Linux Security Module (LSM)) and write-permitted kernel code that changes the security policy or access control decision. Even if the user process attempts to defeat the MAC, the function pointer of the kernel code related to the LSM and security policy in the kernel data cannot be written during another kernel code execution. It is internal to the kernel and has little impact on the performance of user processes.

5.1 Protected Kernel Data Management

Implementations 1 and 2 equally manage the protected kernel data and the processes that handle page faults.

5.1.1 Protected Kernel Data

A Linux kernel with implementations that support an identifier is set to the protected kernel data, which is arranged on one page (4 KB), and the PKS handles the write restriction.

- Identifier: Implementations control the write restriction of the protected kernel data and identification number i. The identification number i is the same as the value of the Pkey i (4 bit) of PTE.
- Write restriction control: Implementations use the identification number i of the protected kernel data to control Pkey i of the PKRS. If the value of WDi in the PKRS is set to 1, write access is restricted; however, if WDi is set to 0, write access is permitted.

The handling of write restrictions by the implementations with the PKRS is a different process (for details, see Sect. 5.2 and 5.3).

5.1.2 Page Fault Handling

The kernel with implementations supports the page fault handler functions `do_page_fault` and `do_double_fault` to identify illegal page references of protected kernel data by

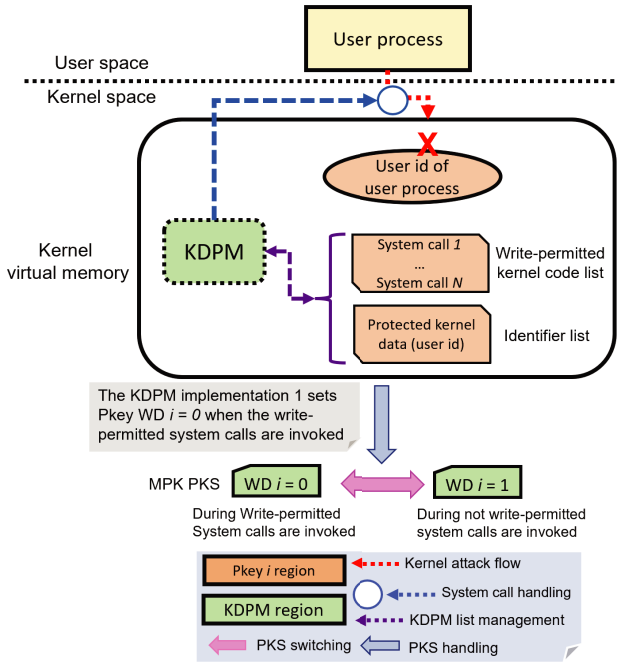

Fig. 5 Implementation 1 of the KDPM

Table 5 Protected kernel data and write-permitted system call of Implementation 1

Item	Description
Protected kernel data	User ID (e.g., uid, euid, fsuid, suid) Group ID (e.g., gid, egid, fsid, sgid)
Write-permitted system call	execve, setuid, setgid, setreuid, setregid setresuid, setresgid, setfsuid, setfsgid

the PKS. In the Linux kernel, a page fault (i.e., error number 35) is a violation of the write protection on a page of Pkey. The implementations do not allow writing to the protected kernel data, and these send a `SIGKILL` to the target user process using the function `force_sig_info`.

5.2 Implementation 1

Figure 5 presents an overview of Implementation 1. Implementation 1 protects the privilege information for each user process. It manages the list of protected kernel data and that of write-permitted system calls.

5.2.1 Protected Kernel Data

Implementation 1 generates a dedicated page (4 KB) as protected kernel data when a user process is created. The dedicated page stores the privileged information of the user process provided in Table 5. The list of write-permitted system calls is also protected and write restriction control is performed by the PKS at the kernel startup.

5.2.2 Handling of Write Restrictions

Implementation 1 admits the system calls that change the privileged information (Figure 5). The process of controlling the write restrictions of the protected kernel data using Pkey is as follows:

1. The kernel identifies a system call invoked by a user process.
2. The kernel determines if the system call number is included in the list of write-permitted system calls.
   a. For write-permitted system calls: the kernel sets the protected kernel data with the write-enable permission by the PKRS.
3. The execution of the system call is continued.
4. After the system call: the kernel restores the protected kernel data and is set to the write-disable permission by the PKRS.

5.2.3 Dedicated Page of Protected Kernel Data

Implementation 1 prepares the dedicated page that stores privileged information. The definition of a dedicated page is the type of `struct` that contains privileged information of `uid` and `gid`. It is the same size as the Linux user process at `struct` of `task_struct` in source code `include/linux/sched.h`. For the running kernel, Implementation 1 allocates the kernel page (4K) as the dedicated page with the zero clear at the user process creation, and stores `uid` and `gid` into the dedicated page. At this time, the user process (e.g., kernel task) of `current` refers to the dedicated kernel page that is one page size (4K). Therefore, Implementation 1 can manage the PKS of the dedicated kernel page for the write protection enabling and disabling at the system call invocation timing (the detail is in Sect. 5.2.4) from the user process.

5.2.4 Kernel Hook Placements

The configuration placements of the PKS are necessary before and after the invocation of the system call. Implementation 1 requires the hook mechanism in the Linux kernel source code `arch/x86/entry/common.c`.

Implementation 1 has two hook points, one is the entering of system call invocation that disables the PKS protection to write the privilege information, other one is the termination of system call invocation that restores the PKS protection to protect the privilege information. Both points are paired and manually implemented into the Linux kernel. Linux kernel forcefully invokes the hook points of Implementation 1 for each system call invocation to disable and restore protected kernel data.

5.3 Implementation 2

Figure 6 presents an overview of Implementation 2. Implementation 2 adopts the write-permitted kernel code of the LSM and supports the list of kernel data related to the LSM

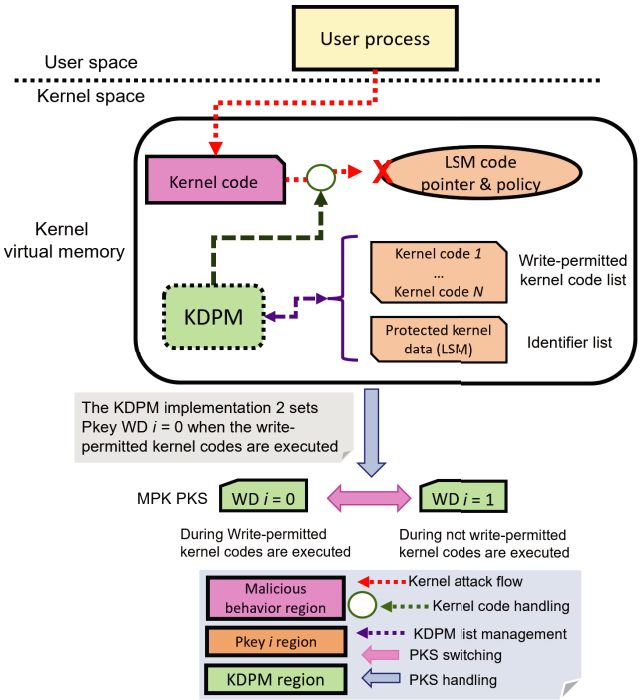

Fig. 6 Implementation 2 of the KDPM

Table 6 Protected kernel data and write-permitted kernel code of Implementation 2

Item	Description
Protected kernel data	Function pointer (e.g., selinux_hooks) Security policy (e.g., selinux_state)
Write-permitted kernel code	Kernel functions in the selinux_hooks avc_init, avc_insert, avc_node_delete, avc_node_replace

and that of the write-permitted kernel codes. Implementation 2 handles the write restrictions for the protected kernel data when executing write-permitted kernel codes that change the access control policy and access control decision.

5.3.1 Protected Kernel Data

Table 6 presents the kernel data to be protected in Implementation 2. Additionally, `selinux_hooks` is a variable that stores function pointers that are part of the kernel data related to the LSM, and `selinux_state` is a variable that stores the access control policy. Furthermore, the list of write-permitted kernel codes is protected by write restriction control using the PKS.

5.3.2 Handling of Write Restrictions

Implementation 2 stores the function pointer of the kernel data related to the LSM, and the list of write-permitted kernel codes is set during the booting of the kernel. Table 6 also presents the kernel codes to be included in the list of write-permitted kernel codes.

The following procedure is used to control the restrictions on the write-permitted kernel code using the PKS:

1. The kernel invokes the kernel code of Implementation 2 during the execution of the write-permitted kernel code.
2. The kernel code of Implementation 2 determines whether the caller belongs to a write-permitted kernel code.
    a. In the case of a write-permitted kernel code, the kernel performs write restriction control using the PKS to set the protected kernel data as write enabled.
3. The kernel continues processing the write-permitted kernel code.
4. Before the end of the write-permitted kernel code, the kernel code of Implementation 2 is called. The kernel performs write restriction using the PKS to set the protected kernel data as write disabled.
5. The kernel finishes the processing of the write-permitted kernel code.

Implementation 2 checks the number of kernel code invocations to determine whether the write restriction enabled and disabled are the same.

5.3.3 Dedicated Page of Protected Kernel Data

Implementation 2 creates a dedicated page that stores function pointers of security mechanisms. The definition of a dedicated page is the type of `struct` that contains function pointers from `security_hook_list selinux_hooks`. It is the same size as Linux MAC (e.g., SELinux) in source code `security/selinux/hooks.c`. For the running kernel, Implementation 2 allocates the kernel page (4K) as the dedicated page with zero clear at the booting time. To store function pointers from `security_hook_list selinux_hooks` into the dedicated page, implementation moves the original function pointers to the dedicated kernel page ones. From this modification, the Linux kernel invokes the MAC functions on the dedicated kernel page. It is statically set at the kernel boot. Therefore, Implementation 2 has to manually insert the PKS write protection enabling and disabling timing into each Linux MAC function entering and exiting placements. It requires the static modification of source code for Linux MAC (e.g., SELinux).

6. Evaluation

6.1 Security Capability

The security capability evaluation validates whether the kernel with the KDPM adequately protects privileged information.

1. Prevention of privilege escalation attack
   A kernel vulnerability that can be exploited for a privilege escalation attack is introduced into the Linux kernel. The evaluation of the kernel with Implementation

1 enables the write restriction of the privileged information of user processes. This prevents an adversary from performing a privilege escalation attack.
2. Preventing the defeat of security mechanism
The evaluation of the kernel with Implementation 2 enables the write restriction of kernel data of the LSM to prevent MAC defeat.

6.2 Performance Evaluation

In performance evaluation, investigation results indicate whether the kernel and user processes are affected by Implementation 1 and the effect of the PKS operations used in Implementation 2.

1. Measurement of the kernel performance overhead
To measure the performance of the Linux kernel with Implementation 1, the benchmark software calculates the overhead of the system call invocation latency.
2. Measurement of PKS performance overhead
To measure the performance of the PKS in the KDPM, the measurement result indicates the processing time of the PKS operations in the Linux kernel with Implementation 2.
3. Measurement of page-level write protection overhead
To compare the performance between page-level write protection and PKS, the measurement result indicates the processing time of the page-level write protection operations in the Linux kernel.
4. Measurement of the kernel instruction increase
To measure the instruction insertion of the Linux kernel with implementations, the disassembling tool indicates the additional instructions.

6.3 Evaluation Environment

6.3.1 Equipment

The evaluation environment for PoC code and kernel was a physical machine equipped with an Intel (R) Core (TM) i7-7700HQ (2.80 GHz, x86_64) processor with 16 GB memory. The security capability evaluation was implemented on a virtual machine because QEMU 6.0.91 supports the PKS. However, the PKS is not available as of November 2022 on the Intel CPU. The guest OS on QEMU was Debian 10.2, and implementations required 15 source files and 431 lines for Linux kernel 5.3.18. The PKS performance for Implementation 2 was evaluated using a measurement program that required 165 lines for Linux kernel 5.3.18.

6.3.2 Implementation

To evaluate the security capability, a kernel vulnerability was introduced into the Linux kernel using a PoC code [17] that leads to privilege escalation via memory corruption through the system call number 350. Additionally, the Linux kernel module (LKM) attempted to overwrite the LSM function pointer to defeat the MAC on the running kernel:

- **Privilege escalation:** Vulnerable kernel code 1 refers to CVE-2017-16995 [17], which was implemented as a system call `sys_kvuln01`. The PoC code exploits the vulnerable kernel code to overwrite the privileged information of a user process for privilege escalation.
- **Defeating security mechanism:** A customized LKM attempts to overwrite the function pointer of the kernel code that manages the LSM file access permission to circumvent the MAC decision.

6.4 Security Capability Evaluation Result

6.4.1 Prevention of Privilege Escalation Attack

The security evaluation result for the adversary's user process is shown in Fig. 7. In line 3, the kernel captures the original system call (i.e., system call number 350) with process ID 1661. The kernel indicates 0x8, which indicates that the write disable (WD) of Pkey 1 is enabled. In line 10, the kernel catches a page fault (i.e., error number 35) when writing to the page that stores the privileged information with Pkey 1. The page fault indicates a write protection violation of a page protected by the Pkey. In line 14, the kernel sends `SIGKILL` to the user process of the adversary.

6.4.2 Preventing the Defeat of Security Mechanism

The security evaluation result of the LKM is shown in Fig. 8.

```
1.  [*] start vulnerable system call (sysnum: 350) invocation
2.  [  364.203190] vulnerable system call invocation
3.  [  364.203227] sysnum: 0x15e (350)
4.  [  364.203275] PID: user process 1661
5.  [  364.203309] PKS PRIV: enable pks currently
6.  [  364.203405] read_pkrs for CPU 0: 0x8
7.  [  364.203496] sys_kvuln01 current privileges 1: uid=33 euid=33 gid=33

8.  [*] process uid, euid, gid, and egid are changed to 0
9.  [*] kernel catches the page fault regarding protection key
10. [  364.204186] PKS: protection keys hw error code 35, pkey 1
11. HW error code 35 (0b100011)
12. Page fault error code bits: from Linux v5.3.18 : arch/x86/include/asm/trap_pf.h
      1.  bit 0 == 1: protection fault, X86_PF_PROT
      2.  bit 1 == 1: write access, X86_PF_WRITE
      3.  bit 5 == 1: protection keys block access, X86_PF_PK
13. [  364.204232] read_pkrs for CPU 0: 0x8
14. [  364.212966] killing target PID: 1661
```

Fig. 7 Prevention of a privilege escalation attack

```
1.  [*] LKM attempts to find the function pointer of the LSM
2.  [  286.118427] selinux_hooks[56].hook.file_permission Address ffffffff81e77c18
3.  [  286.213409] PKS PRIV: enable pks currently
4.  [  286.214513] read_pkrs for CPU 0: 0x8
5.  [*] LKM tries to overwrite the function pointer of the LSM
6.  [*] kernel catches the page fault regarding protection key
7.  [  286.216821] PKS: protection keys hw error code 35, pkey 1
8.  HW error code 35 (0b100011)
9.  Page fault error code bits: from Linux v5.3.18 : arch/x86/include/asm/trap_pf.h
      1.  bit 0 == 1: protection fault, X86_PF_PROT
      2.  bit 1 == 1: write access, X86_PF_WRITE
      3.  bit 5 == 1: protection keys block access, X86_PF_PK
10. [  286.221232] read_pkrs for CPU 0: 0x8
```

Fig. 8 Prevention of a MAC defeat

Table 7 System call invocation overhead of Implementation 1 (μs)

System call	Vanilla kernel	Implementation 1	Overhead
fork+/bin/sh	227111.28	236738.69	9627.41 (4.24%)
fork+execve	12780.0566	13931.6703	1151.6136 (9.01%)
fork+exit	10837.0729	11285.5603	448.4874 (4.14%)
open/close	1302.5639	1334.5312	41.9672 (2.95%)
read	168.8898	180.4594	11.5696 (6.85%)
write	164.2567	176.4273	12.1705 (7.41%)
fstat	195.0063	203.7508	8.7445 (4.48%)
stat	613.7426	631.9393	18.1966 (2.96%)

Table 8 Overhead of PKS operations (ns)

Instruction	Implementation 2
Pkey write	30.5
PKRS read	22.1
PKRS write	1347.9

In line 2, the LKM attempts to find one of the function pointers of `selinux_hooks`. In line 5, LKM attempts to overwrite the function pointer of `selinux_hooks`. In line 7, the kernel catches a page fault (i.e., error number 35) when writing to the page storing the function pointer with Pkey 1. The page fault indicates a write protection violation of a page protected by Pkey.

Implementation 2 focuses on the prevention of kernel memory corruption to security mechanisms. The LKM tries to overwrite function pointers of LSM, then Implementation 2 can handle the page fault of Pkey. From the security evaluation results, the Linux kernel with the KDPM catches privilege escalation attacks is confirmed. The KDPM correctly manages the Pkey and detect memory corruption of the vulnerable kernel code. To stop illegal writing, Implementation 2 requires the double fault that leads the kernel panic. At the evaluation, Implementation 2 disables the PKS protection when the page fault of Pkey has occurred.

6.5 Performance Evaluation Result

6.5.1 Measurement of the Kernel Processing Overhead

The system call overhead was measured using LMbench benchmark software. A vanilla kernel was compared with the kernel with Implementation 1. LMbench was executed 10 times to calculate the average system call latency.

LMbench performs 54 invocations of the system call for fork+/bin/sh, 4 invocations for fork+execve, 2 invocations for fork+exit and open/close, and 1 invocation each of the other system calls. Table 7 shows the overhead of the system call. The highest and lowest overheads are fork+execve with 9.01% and stat with 2.96%, respectively.

6.5.2 Measurement of PKS Operations

The Linux kernel with Implementation 2 invokes the Pkey write of the PTE and read and write of the PKRS. The measurement program was repeated 10,000 times, and the average value was calculated. Table 8 shows the cost of the PKS operations. The write of Pkey required 30.5 ns; PKRS read required 22.1 ns, and PKRS write required 1347.9 ns.

Table 9 Overhead of page-level write protection operations (ns)

Instruction	Page-level write protection
R/W flag write	38.3
R/W flag read	31.6

Table 10 Increase in instructions on the Linux kernel

	Implementation 1	Implementation 2
Instructions	176	137

6.5.3 Measurement of Page-Level Write Protection Operations

The page-level write protection requires the R/W flag read or write of the PTE. The measurement program is LKM which measures 10,000 times and calculates the average value of operation cost on the vanilla Linux kernel. Table 9 shows the cost of the page-level write protection operations. The R/W flag write required 38.3 ns, and the R/W flag read required 31.6 ns.

6.5.4 Measurement of the Kernel Instruction Increase

The Linux kernel with implementations requires protected kernel data management and handling of write restrictions that contain the instructions of PKS operations. For the calculation of the instruction increase, both implementations extract the additional kernel code to the source code file, then the vanilla kernel only contains the invocation placement (e.g., function call) of both implementations. The disassemble tool calculates instructions for the object files of each implementation.

Table 10 shows the increase in instructions. Implementation 1 requires 176 instructions and Implementation 2 requires 137 instructions.

7. Discussion

7.1 Security Capability Consideration

From security capability evaluation results, the kernel with the KDPM can prevent a privilege escalation attack from a PoC code through a kernel vulnerability and an LKM with a defeat of security mechanisms. The careful consideration of kernel with KDPM can cover other PoC codes of kernel vulnerabilities when the vulnerable kernel code is invoked (Table 2). In addition, the kernel and user process operations were not affected by the operation that restricts the writing of kernel data. The evaluation result confirms that the KDPM can dynamically control read restrictions by appropriately setting the PKS. The KDPM only allows system calls for permissions to change privileges and kernel codes for modifying access control information. Therefore, the KDPM prevents the illegal modification of privileged information and kernel code related to access control.

Additionally, the KDPM mitigates the threat from the latest kernel vulnerabilities (e.g., zero-day attack) before the kernel patch is released. Because the KDPM manages a small number of write-permitted system calls and kernel codes. It ensures that the system call or kernel code of a zero-day attack can be manually removed from the write-permitted lists for the protected kernel data to reduce the potential of a kernel attack.

Moreover, analyzing the security capability of the implementations requires the inspection of memory access sequences from the attack of the actual memory corruption kernel vulnerability that performs the illegal modification of kernel data for additional evaluation.

7.2 Performance Consideration

The performance evaluations reveal that the kernel with the implementations requires overhead in kernel processing and read control by the PKS. The duration required for the PKS operations of Implementations 1 and 2 are the same.

Owing to the difference of performance costs for each implementation, Implementation 1 determines whether a system call number is allowed to be written. The user process affects the execution time of the system call and generates privileged information of the user process. Meanwhile, Implementation 2 determines the write-permitted kernel code for processing each access control mechanism. The kernel with Implementation 2 has an impact on kernel processing when access control decisions are necessary.

From the comparison of performance cost between page-level write protection and PKS, the measurement results indicate that PKS has lower processing overhead than page-level write protection in the Linux kernel. The operation cost depends on the number of page management. PKS requires Pkey assignment and PKRS writing for write restriction changing of multiple pages, however, the page-level write protection needs to manage all protection target pages and sequentially increase the write operation cost for each page. KDPM requires the privilege information write restriction for all user processes and security mechanisms. The page-level write protection does not satisfy a lot of pages and multiple types of kernel data protection separately. Therefore, PKS is efficient overhead and reasonable implementation of write restrictions for memory access rights.

To inspect the performance costs from the viewpoint of instruction, Implementation 1 requires 176 instructions and Implementation 2 requires 137 instructions. The instructions for Implementation 1 are more than that for Implementation 2 owing to the number of system call checking for the write-permitted list in the handling of write restriction. If Implementation 2 checks the additional write-permitted kernel code, the instructions are sequentially increased for the running kernel image.

7.3 Limitation

7.3.1 Design Limitation

The performance evaluation results show the PKS is lightweight for protecting kernel data. However, if multiple kernel data share a Pkey, the effects of the write available timing during asynchronous processing should be determined due to interruptions and exceptions in the kernel.

If a kernel vulnerability is discovered and an attack is successful, the vulnerable kernel code may have contained a write-permitted system call or write-permitted kernel code. This is a case of circumventing of the KDPM, which allows the modification of protected kernel data.

The design of the KDPM retains the static information in the list of write-permitted system calls and that of write-permitted kernel codes for the kernel. Customizing both lists is difficult and requires additional permissions for the running kernel or kernel modules. Both lists are modified through a kernel component (e.g., kernel module or extended Berkley Packet Filter).

7.3.2 Implementation Limitation

Although Implementation 1 requires an additional kernel process for the invocation point of system call, Implementation 2 requires an additional kernel process for the restriction of kernel data related functions, which adds to the performance load and requires kernel modifications.

Moreover, the implementation of multi-CPU cores requires the save and restore control for kernel context switching because PKRS is provided for each CPU core. KDPM implementations require the management of the PKRS state for each kernel context. The lock mechanism of PKRS is necessary for irregular write or exception handling to forcibly prohibit the sharing of PKRS state across the multiple kernel task.

7.3.3 Mitigation of Deadlock of Kernel Thread

The consideration of Implementation 1 for mitigating the deadlock of the kernel. Implementation 1 sends a KILL to the malicious user process when a page fault is occurred by illegal writing of protected kernel data. At this time, if the corresponding kernel thread acquired global locks for kernel data, the running kernel takes a dead lock that leads a panic or unstable behavior. It depends on the user process and corresponding kernel thread situation because global locks have a wide variety of types in the Linux kernel (e.g., `DEFINE_MUTEX`). The malicious user process takes the combinations of multiple systems call invocations to acquire global locks before the illegal writing to protected kernel data of Implementation 1.

Implementation 1 requires an additional mechanism to mitigate deadlock to achieve stable kernel behavior. To ensure kernel stability, Implementation 1 should check the ac-

quired status of global locks before the sending KILL signal to the kernel thread. This additional mechanism needs the system call invocation history and manual investigation of the relationship between system calls and global locks. If the malicious user process invokes these system calls, Implementation 1 waits for the sending of the KILL signal to the corresponding kernel thread before global locks are forcibly released. We have considered this mechanism for Implementation 1 to keep kernel stability in the future.

7.3.4 Limitation of Preventing the Defeating of Security Mechanism.

The consideration of Implementation 2 for preventing the defeating of security mechanisms. If the double fault occurs, the kernel is stopped. Implementation 2 requires an additional mechanism to protect the security mechanism with stable kernel behavior. To ensure kernel stability, Implementation 2 disables the Pkey protection when the page fault of Pkey occurs. This mechanism temporary allows the adversary can overwrite the kernel data, however, Implementation 2 should support an additional mechanism that stores the original kernel data at the kernel booting, then writes back it to the modified kernel data after the page fault of Pkey is occurred. It ensures the mitigation of illegal writing of security mechanisms. We have considered this mechanism for Implementation 2 to support preventing security capability with kernel stability in the future.

7.3.5 Hardware Limitation

The limitation of the PKS is that the number of Pkeys is 16. The 0th Pkey is used as the initial value of the PTE. The kernel data to be protected must be managed using 15 Pkeys. As the number of types of kernel data to be protected is limited, an appropriate classification of kernel data should be considered when applying the KDPM.

7.3.6 Affection of Hardware Limitation

The consideration of the KDPM protection policy is a target of kernel memory corruption to the Linux kernel. The adversary focuses its purpose of attack method to write the specific kernel data. The KDPM protection policy covers the two types of kernel data. One is privileged information to mitigate privilege escalation attacks. Another one is access control policy and kernel function pointers to mitigate defeating security mechanisms. The adversary also targets other security features (i.e., cgroups, namespaces, and so on). Therefore, the types of KDPM protection target is affected by the limitation of the number of Pkeys.

7.3.7 Mitigation of Hardware Limitation

We must consider the mitigation of the number of Pkey. One is software mechanism that combines the few protection targets into one protection target. It means that one of Pkeys covers multiple protection targets to mitigate the hardware limitation. The developer have to know kernel implementation and security features have no contradiction for Pkey available and disabling timing at the same time. Another one is hardware mechanism that requires the combination of other hardware mechanisms to increase the number of Pkeys. The previous work [25] has already proposed that the enhancing of the number of Pkey limitation with Extended Page Table (EPT) on the Intel CPU architecture. The combination of the EPT and PKS to support to guest OS (e.g., microkernel) with Pkey transparency. It avoids the limitation of the number of Pkey, however, the implementation of Linux kernel requires VMM feature and complex implementation to separate kernel feature to microkernel architecture.

7.4 Portability

The portability of the KDPM to other OSs must be considered. The KDPM relies on the PKS, which requires the implementation of virtual memory space with a PTE in the OS that supports an Intel CPU.

8. Related Work

User Process Data Protection using the MPK: For data protection using the MPK in applications, libmpk provides a flexible library that supports user processes. This can manipulate the protected data using the PSU [19]. ERIM is proposed as a separation method for the protected user process data into different user processes using PSU [20]. Cerberus is proposed as a sandbox framework for user application using the PSU [21].

Kernel Data Protection using the MPK: To protect the kernel code and kernel data using the MPK in the kernel, xMP proposes a security mechanism that provides multiple domains. These contain pages of kernel memory space that are allocated using the PKU. The virtual machine monitor (VMM) manages domains via Pkeys [22]. Additionally, libhermitMPK proposes a security mechanism to protect against unauthorized reading and writing by dividing and managing the kernel code and data into multiple Pkeys [23]. UnderBridge applies MPK for a microkernel between user space and kernel space at runtime isolation of IPC mechanism [24]. EPK adopts virtualization features to increase the number of Pkeys in MPK for a guest OS of microkernel [25].

Prevention of Malicious Code Execution: To prevent illegal kernel code execution in the kernel or hypervisor, the control flow integrity (CFI) [7], [26], [27], which verifies the order of program function calls, is applied [28]. To apply the CFI to the kernel, KCoFI is proposed as a security mechanism for preserving the integrity of the order of invoking kernel codes as the original architecture [8]. Additionally, pointer authentication based CFI achieves the protection of kernel execution context with low overhead using ARM hardware feature [29].

Table 11 Comparison of kernel protection approaches and types of target vulnerability (C.: code execution, M.: memory corruption) [30] (✓ is supported; △ is partially supported)

Feature	Sub Feature	libhermitMPK [23]	xMP [22]	KCoFI [8]	KDPM
Protection target	Entire kernel	✓	✓		
	Kernel behavior			✓	
	Kernel data				✓
Granularity of MPK separation	Region	✓			
	Domain		✓		
	Kernel data				✓
Granularity of target	System call				✓
	Kernel code	✓		✓	✓
	VM		✓		
Implementation site	In-kernel	✓		✓	✓
	VMM monitoring		✓		
Arbitrary code executions				△	
Limitation of capability		Kernel code security	VMM overhead	Original Architecture	Pkey number
Target Vulnerability		M.	M.	C.	M.

8.1 Comparison

Table 11 presents a comparison of the KDPM with existing security mechanisms [8], [22], [23].

Furthermore, libhermitMPK separates the kernel into two regions (i.e., Safe/Unsafe) using Pkeys [23]. The running kernel code can only read and write to kernel data belonging to the same region. In addition, xMP manages the kernel memory space of the guest OS kernel into multiple domains using Pkeys. The kernel codes and kernel data are assigned forcefully for each domain through the VMM [22]. Although libhermitMPK and xMP show that kernel data can be overridden if the same Pkey is assigned to a vulnerable kernel code and overhead using the VMM, the KDPM assigns Pkeys only to the protected kernel data to separately control system calls and kernel codes from the write restrictions of the kernel data.

KCoFI adopts the CFI for kernel processing that corresponds with the asynchronous behavior to handle the interruption and context switch of tasks [8]. Although KCoFI prevents the invocation of illegal kernel code, kernel memory corruption is not covered. If an attacker executes an arbitrary code in the kernel mode, the KDPM protection may be defeated. The recommendation is the applying of the CFI to the kernel with the KDPM to prevent hardware security defeat. Therefore, the CFI verifies the order of invocation of kernel codes to prevent the illegal execution of the kernel code, which attempts to controls hardware registers. The kernel with the KDPM preserves the kernel data protection.

9. Conclusion

An adversary can achieve privilege escalation and the defeat of security mechanisms by corrupting the kernel memory. KCoFI, KASLR, and AKO are kernel attack countermeasures that mitigate and prevent the threat of kernel attacks. However, vulnerable kernel codes can still modify the kernel data at the kernel layer.

In this paper, a novel security design of a KDPM that manages write restrictions on specific kernel data is proposed. The KDPM enables the kernel to control write privileges on PTEs using the MPK PKS in the running kernel by the CPU. From the two implementations of the KDPM, Implementation 1 protects the privileged information of the user process to prevent privilege escalation, whereas Implementation 2 protects the kernel data of the MAC to prevent the defeat of security mechanisms.

The security capability evaluation indicated a kernel vulnerability that can be exploited for privilege escalation attacks and demonstrated the restriction capability for the writing of privileged information of the user processes. The performance evaluation showed that the overhead for invoking system calls on Linux with Implementation 1 ranged from 2.96% to 9.01%, and the PKS operations overhead on Linux with Implementation 2 ranged from 22.1 ns to 1347.9 ns. Additionally, the increase in number of instructions indicates that implementations require 137 to 176 instructions.

In future studies, to prevent vulnerable kernel code execution and illegal modification of kernel data due to the principle of security risk and performance overhead, researchers can provide the design of lightweight security mechanism that combines the verification of kernel code execution sequence and the write protection of kernel data at the adequate timing to mitigate kernel attacks.

Acknowledgments

This work was partially supported by the Japan Society for the Promotion of Science (JSPS) KAKENHI Grant Numbers JP19H04109, JP22H03592, JP23K16882, and ROIS NII Open Collaborative Research 2022 (22S0302)/2023 (23S0301). Hiroki's contribution contained in the paper is done when he belonged to SECOM Co., Ltd.

References

[1] MITRE, "Common Vulnerabilities and Exposures," https://www.cve.org/, Accessed Aug. 18. 2022.
[2] NIST, "Official Common Platform Enumeration Dictionary," https://nvd.nist.gov/products/cpe, Accessed Aug. 18. 2022.

[3] NIST, "National Vulnerability Database," https://nvd.nist.gov/, Accessed Aug. 18. 2022.
[4] H. Kuzuno and T. Yamauchi, "KDPM: Kernel Data Protection Mechanism Using a Memory Protection Key," Proc. 17th International Workshop on Security (IWSEC), LNCS, vol.13504, pp.66–84, 2022.
[5] Exploit Database, "Nexus 5 Android 5.0 - Privilege Escalation," https://www.exploit-db.com/exploits/35711/, Accessed June 15. 2022.
[6] grsecurity, "super fun 2.6.30+/RHEL5 2.6.18 local kernel exploit," https://grsecurity.net/~spender/exploits/exploit2.txt, Accessed June 15. 2022.
[7] Abadi, M., Budiu, Mihai., Erlingsson, U. and Ligatti, J., "Control-Flow Integrity Principles, Implementations," Proc. 12th ACM Conference on Computer and Communications Security (CCS), pp.340–353, 2005.
[8] J. Criswell, N. Dautenhahn, and V. Adve, "KCoFI: Complete Control-Flow Integrity for Commodity Operating System Kernels," Proc. 35th IEEE Security and Privacy, pp.292–307, 2014.
[9] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh, "On the effectiveness of address-space randomization," the 11th ACM Conference on Computer and Communications Security (CCS), pp.298–307, 2004.
[10] T. Yamauchi, Y. Akao, R. Yoshitani, Y. Nakamura, and M. Hashimoto, "Additional kernel observer: privilege escalation attack prevention mechanism focusing on system call privilege changes," International Journal of Information Security, vol.20, no.13, pp.461–473, 2021.
[11] Exploit Database, "OffSec Services Limited," https://www.first.org/cvss/, Accessed Aug. 18. 2022.
[12] Bonzini, P., "[PATCH] target/i86: implement PKS," https://lore.kernel.org/qemu-devel/20210127093540.472624-1-pbonzini@redhat.com/, Accessed Aug. 18. 2022.
[13] Intel Corporation, "Intel(R) 64 and IA-32 Architectures Software Developer's Manual," https://www.intel.co.jp/content/www/jp/ja/architecture-and-technology/64-ia-32-architectures-software-developer-system-programming-manual-325384.html, Accessed Aug. 18. 2022.
[14] H. Chen, Y. Mao, X. Wang, D. Zhow, N. Zeldovich, and M.F. Kaashoek, "Linux kernel vulnerabilities: state-of-the-art defenses and open problems," Proc. 2nd Asia-Pacific Workshop on Systems (APSys), pp.1–5, 2011.
[15] MITRE, "CVE-2016-4997," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-4997, Accessed Sept. 16. 2022.
[16] MITRE, "CVE-2016-9793," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-9793, Accessed Sept. 16. 2022.
[17] MITRE, "CVE-2017-16995," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-16995, Accessed Sept. 16. 2022.
[18] MITRE, "CVE-2017-1000112," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-1000112, Accessed Sept. 16. 2022.
[19] Park, S., Lee, S., Xu, W., Moon, H., Kim, T., "libmpk: Software Abstraction for Intel Memory Protection Keys (Intel MPK)," Proc. 2019 USENIX Annual Technical Conference (ATC), pp.241–254, 2019.
[20] Vahldiek-Oberwagner, A., Elnikety, E., Duarte, O., N., Sammier, M., Druschel, P., Garg, D., "ERIM: Secure, Efficient In-process Isolation with Protection Keys (MPK)," Proc. 28th USENIX Conference on Security Symposium, pp.1221–1238, 2019.
[21] A. Voulimeneas, J. Vinck, R. Mechelinck, and S. Volckaert, "You shall not (by)pass!: practical, secure, and fast PKU-based sandboxing," Proc. Seventeenth European Conference on Computer Systems, pp.266–282, 2022.
[22] S. Proskurin, M. Momeu, S. Ghavamnia, V.P. Kemerlis, and M. Polychronakis, "xMP: Selective Memory Protection for Kernel and User Space," Proc. 41st IEEE Symposium on Security and Privacy, pp.563–577, 2020.
[23] M. Sung, P. Olivier, S. Lankes, and B. Ravindran, "Intra-Unikernel Isolation with Intel Memory Protection Keys," Proc. 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, pp.143–156, 2020.
[24] Gu, J., Wu, X., Li, W., Liu, N., Mi, Z., Xia, Y., Chen, H., "Harmonizing Performance and Isolation in Microkernels with Efficient Intra-kernel Isolation and Communication," Proc. 2020 USENIX Annual Technical Conference (ATC), pp.401–417, 2020.
[25] Gu, J., Li, H., Li, W., Xia, Y., Chen, H., "EPK: Scalable and Efficient Memory Protection Keys," Proc. 2022 USENIX Annual Technical Conference (ATC), pp.609–624, 2022.
[26] Z. Wang and X. Jiang, "Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity," Proc. 31st IEEE Symposium on Security and Privacy, pp 380–395, 2010.
[27] X. Ge, N. Talele, M. Payer, and T. Jaeger, "Fine-grained control-flow integrity for kernel software," Proc. 1st IEEE European Symposium on Security and Privacy, pp.179–194, 2016.
[28] Edge, J., "Control-flow integrity for the kernel," available from https://lwn.net/Articles/810077/, Accessed Nov. 07. 2022.
[29] Yoo, S., Park, J., Kim, S., Kim, Y., Kim, T., "In-Kernel Control-Flow Integrity on Commodity OSes using ARM Pointer Authentication," Proc. 31st USENIX Conference on Security Symposium, pp.89–106, 2019.
[30] CVE details, "Linux Vulnerability Statistics," https://www.cvedetails.com/vendor/33/Linux.html, Accessed Nov. 05. 2022.

Hiroki Kuzuno received an M.E. degree in Information Science from Nara Institute of Science and Technology, Japan, in 2007, and a Ph.D. in Computer Science from Okayama University, Japan in 2020. Currently, he is an Assistant Professor at Kobe University, Japan. His research focuses on computer security specifically on operating systems and networks. He is a member of IEICE and IPSJ.

Toshihiro Yamauchi received B.E. M.E. and Ph.D. degrees in Computer Science from Kyushu University, Japan in 1998, 2000, and 2002, respectively. In 2001, he was a Research Fellow of the Japan Society for the Promotion of Science. In 2002, he became a Research Associate with the Faculty of Information Science and Electrical Engineering at Kyushu University. In 2005, he became an Associate Professor with the Graduate School of Natural Science and Technology at Okayama University, Japan. He has been serving as a Professor with Okayama University since 2021. His research interests include operating systems and computer security. He is a member of IPSJ, IEICE, ACM, USENIX and IEEE.