Kobe University Repository : Kernel

PDF issue: 2025-12-04

Prevention of Kernel Memory Corruption Using Kernel Page Restriction Mechanism

Kuzuno, Hiroki
Yamauchi, Toshihiro

(Citation)
Journal of Information Processing,30:563-576

(Issue Date)
2022-09-15

(Resource Type)
journal article

(Version)
Version of Record

(Rights)
© 2022 by the Information Processing Society of Japan
Notice for the use of this material The copyright of this material is retained by the Information Processing Society of Japan (IPSJ). This material is published on this web site with the agreement of the author (s) and the IPSJ. Please be complied with…

(URL)
https://hdl.handle.net/20.500.14094/0100483237

Journal of Information Processing Vol.30 563–576 (Sep. 2022)

[DOI: 10.2197/ipsjjip.30.563]

Recommended Paper

Prevention of Kernel Memory Corruption Using Kernel Page Restriction Mechanism

Hiroki Kuzuno[1,a)] Toshihiro Yamauchi[2,b)]

Received: December 8, 2021, Accepted: June 14, 2022

Abstract: An adversary's user process can compromise the security of the operating system (OS) kernel, and subsequent invocation of the vulnerable kernel code can cause kernel memory corruption. The vulnerable kernel code could overwrite the kernel data containing the privilege information of user processes or the kernel data related to security features (i.e., mandatory access control). As a means of kernel protection, OS researchers have proposed the multiple kernel address space approach that partitions the kernel address space to protect the kernel memory from memory corruption (e.g., process-local memory and system call isolation). However, in the previous approach, the vulnerable kernel code and the kernel data targeted for attack still reside in the same kernel memory. Consequently, to compromise the kernel, adversaries simply focus on calling the latest vulnerable kernel code, which relies on the starting points of the kernel attack process. With the aim of preventing such subversion attacks, this paper proposes the kernel page restriction mechanism (KPRM), which employs an alternative design and method to obviate kernel memory corruption. The objective of the KPRM is to prohibit vulnerable kernel code execution and prevent writing to the kernel data from an adversary's user process. KPRM ensures the unmapping of vulnerable kernel code or kernel data to prevent the exploitation of the kernel due to kernel vulnerability. Therefore, an adversary's user process is obstructed from executing vulnerable kernel code and overwriting kernel data on the running kernel. Evaluation results indicate that actual proof-of-concept attacks on vulnerable kernel code resulting in kernel memory corruption can successfully be prevented by KPRM. Moreover, the implementations of KPRM indicate that the maximum latency for system calls is 0.703 μs, while the overhead for 100,000 Hypertext Transfer Protocol (HTTP) downloads via a web client program ranged from 1.188% to 4.093% of the access overhead. In addition, KPRM implementations achieved acceptable overheads of 2.459% and 2.193% for the kernel compile time.

Keywords: memory corruption, kernel vulnerability, system security, operating system

1. Introduction

Kernel vulnerability attacks aim to invoke malicious kernel code that can cause corruption of the kernel memory (vulnerable kernel code). They can leverage the security features of the kernel and modify the privilege information of the running kernel [1]. The vulnerable kernel code relies on the user processes that share the kernel address space in the kernel mode. Consequently, an adversary's user process can execute vulnerable kernel code to overwrite existing kernel features [2].

To prevent kernel memory corruption, researchers have proposed that the kernel protection mechanism should provide security features for each attack method. Kernel control flow integrity (CFI) supports kernel behavior verification [3]. Kernel address randomization (KASLR) provides the identification hardening of kernel code and kernel data [4]. Moreover, process-local memory (Proclocal) allocates a specific kernel address space to a user process [5]. System call isolation (SCI) creates a dedicated kernel address space for the processing system call's routines [6]. In addition, a CPU privilege mechanism provides supervisor mode access prevention (SMAP) and supervisor mode execution prevention (SMEP) to forcibly deny access and execution permission between the user mode and the kernel mode [7].

These countermeasures can effectively obviate and mitigate the kernel memory corruption, but leave two problems unsolved. First, the running kernel still requires the full kernel page mapping of the kernel code and kernel data. Second, Proclocal can protect the kernel data of only specific kernel components and SCI creates a statically duplicated kernel address space for each user process owing to stable behavior for kernel processing.

However, these mechanisms place potentially vulnerable kernel code and kernel data targeted for attacks in the same kernel address space. Thus, invoking the vulnerable kernel code and overwriting the kernel data at the kernel layer remains an unaddressed threat. Moreover, two internet articles (grsecurity and database exploit) have already shown that the latest kernel attack ideas and methods can invoke the vulnerable kernel code to escalate privileges and bypass security features (i.e., mandatory access control

1 Graduate School of Engineering, Kobe University, Kobe, Hyogo 657–8501, Japan
2 Faculty of Natural Science and Technology, Okayama University, Okayama 700–8530, Japan
a) kuzuno@port.kobe-u.ac.jp
b) yamauchi@okayama-u.ac.jp

This paper is an extended version of a paper published in the 16th International Workshop on Security (IWSEC 2021) [8].
The preliminary version of this paper was published at the 16th International Workshop on Security (IWSEC 2021), September 2021. The paper was recommended to be submitted to Journal of Information Processing (JIP) by the Program Co-Chairs of IWSEC 2021.

© 2022 Information Processing Society of Japan

(MAC)) of SELinux [9]) for the running kernel [10], [11].

This paper describes the features of a novel security design known as the kernel page restriction mechanism (KPRM). KPRM is a security capability enhancement that mitigates kernel memory corruption due to kernel vulnerabilities. It has a restriction mechanism that can extend invocation control of vulnerable kernel code as well as access to kernel data by an adversary's user process on the running kernel. KPRM uses two types of kernel pages, namely, normal and restricted kernel pages, to run the kernel and user processes. It first assigns vulnerable kernel code and protected kernel data (e.g., user identifiers) to a restricted kernel page; then, it stores the remaining kernel code and kernel data in normal kernel pages.

KPRM achieves two objectives. First, to prevent memory corruption, it provides data security during kernel page handling by ensuring that the adversary's user process cannot access restricted kernel page references in its own kernel address space to prevent memory corruption. Second, it dynamically unmaps restricted kernel page references to prevent the invocation of vulnerable kernel code for the adversary's user process during the system call processing.

KPRM performs its procedures on all user processes that have not been identified as benign but are manually registered in the running kernel's benign user process list. It ensures that the kernel can dynamically reduce its attack surface where the kernel address space is shared by both vulnerable kernel code and attack targeted kernel data. KPRM assumes that the common vulnerabilities and exposures (CVE) list that provides kernel vulnerabilities. KPRM manually extracts target kernel code from kernel vulnerabilities for the restriction target of vulnerable kernel code.

The implementation of KPRM provides two prototypes on the latest Linux kernel. This is equivalent to the user of KPRM adjusting the protection of kernel code and kernel data owing to the cost of processing the kernel address space for the running kernel. The first implementation provides an additional kernel address space to improve security. It reserves the vulnerable kernel code and protected kernel data for restricted kernel pages so that the invocation of vulnerable kernel code and access of kernel data by the adversary's user process are prevented. The additional kernel address page is managed as a dedicated kernel page table. KPRM prepares process-context identifiers (PCIDs) of the translation lookaside buffer (TLB) for each page table. It reduces the cost of flushing the TLB when the kernel performs the switching of kernel address spaces. The second implementation aims to reduce the overhead. It is possible to reserve protected kernel data for restricted kernel pages because user processes share kernel address space. KPRM only handles the kernel page references of kernel code related to the adversary's user process and the protected kernel data for reducing kernel misuse during the interruption processing.

In summary, the primary contributions of this study are summarized as follows:

(1) KPRM, a novel approach for mitigating kernel memory corruption, is proposed. The key objectives of KPRM are restriction of the execution of vulnerable kernel code and access to kernel data in KPRM implementations. This paper also discusses the threat model, security features, limitations, software portability, and hardware considerations of KPRM.

(2) The effectiveness of the KPRM implementations is based on the efficacy with which they prevent both the invocation of vulnerable kernel code and the illegal modification of the kernel data using actual proof-of-concept (PoC) code. Performance measurement of KPRM requires software benchmarks and applications such as the Apache web server and compiler. Its performance evaluation results show that the maximum round time overhead for a system call is 0.703 μs. The overhead for 100,000 Hypertext Transfer Protocol (HTTP) downloads via a web application range from 1.188% to 4.093% of the access overhead. Moreover, the implementations of KPRM achieved acceptable performance scores indicating overheads of 2.459% and 2.193% for the Linux kernel compile time.

2. Background

2.1 Address Space and Page Table

Modern kernels and hardware support the page table structure to provide virtual memory that is larger than the physical memory. **Figure 1** shows the page table structure that creates virtual address spaces. Moreover, the page table structure (e.g., 4 / 5 level paging on the x86_64 CPU architecture (x86_64)) maintains page and page table entries, which shows the relationships between

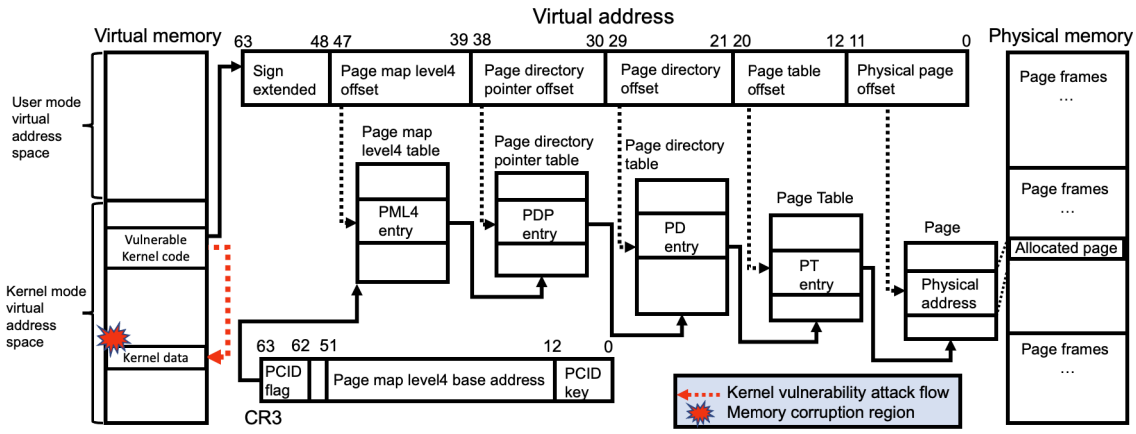

Fig. 1 Page table management of virtual address [12] and physical address with attack regions.

© 2022 Information Processing Society of Japan

the virtual and physical addresses. Additionally, the CR3 register stores the physical address of the page table on the x86_64 [13].

Linux allocates the virtual address space, which is separated into two regions for user and kernel modes [12]. Linux maps a virtual address (48 bits on the x86_64) to a physical address on the page table. The smallest set is a page (4 kB on the x86_64). The Linux kernel page stores the kernel code and kernel data in the specific virtual address space of the kernel page for the kernel mode.

2.2 Memory Corruption Vulnerability

The ten types of kernel vulnerabilities are types of improper implementations that lead to kernel attacks [1]. The CVE list contains 2,708 Linux kernel vulnerabilities [14]. A typical kernel vulnerability discovered is memory corruption, which has been presented in 140 reports [14]. The illustration of the memory corruption mechanism demonstrates that a vulnerable kernel code directly modifies the kernel data in the kernel address space to take the root privilege capability (e.g., administrator account) [1]. Figure 1 shows the attack regions of the memory corruption that performs a privilege escalation attack or security features subverting. An adversary's user process requires the overwriting of the kernel data that contains the `UID` variable of the user identifier in the kernel address space.

The latest Linux kernel uses the MAC of the Linux security module (LSM) to restrict the privilege capability of the administrator. To bypass the MAC restriction, the adversary's user process utilizes the memory corruption vulnerability to replace the LSM's kernel code of `security_hook_list` (e.g., function pointer) with the non-checking access control kernel code [10], [11].

3. Threat Model

3.1 Attack Environment

The attack environment of the threat model for KPRM requires preparation to counter the adversary's intentions during an attack scenario. The attack environment available to the adversary and the kernel capabilities are as follows:

- **Adversary**: An adversary takes normal user privileges and controls the shell command with the PoC code that exploits kernel vulnerability to perform the kernel memory corruption.
- **Kernel**: A kernel provides the sharing of the kernel address space for every user process. The kernel address space contains the vulnerable kernel code and the kernel data targeted by the attack.
- **Kernel vulnerability**: This is a registered kernel vulnerability [14]. A vulnerable kernel code is identified as a known piece of kernel vulnerability. The adversary's shell executes the PoC code as a user process that invokes the vulnerable kernel code. It modifies any kernel data that is present in the kernel address space to lead the kernel memory corruption.
- **Attack target**: This contains the security feature of the kernel data of the kernel code (i.e., an access control policy or a function pointer of MAC) and kernel data privilege information (i.e., user identifier). These are the key points of the administrator's privilege restrictions on the kernel.

3.2 Attack Scenario

The adversary's attack scenario is the approach used to induce the privilege escalation. First, the PoC code of the adversary's user process must be accessed and executed to call the vulnerable code at the attack's starting point. Subsequently, the adversary's user process can access any kernel virtual address from the vulnerable kernel code in the same kernel address space. Therefore, the adversary's user process modifies the security features of the kernel code to bypass the administrator privilege restriction. Then it forcibly invokes the kernel code that modifies the privilege information of the kernel data for privilege escalation. Finally, the adversary's user process now has administrator privileges, hence they can control the compromised kernel in the attack environment.

3.3 Preparation

KPRM's resilience preparation requires the two building processes to protect the security features and the privilege information from the attack scenario in the attack environment.

In the first building process, KPRM requires the manual preparation of the restricted kernel code list (e.g., virtual address ranges and kernel function names) and the protected kernel data list for the additional information that is accessed. In the second building process, KPRM has registered kernel code and kernel data to restrict kernel pages during its kernel booting. To cover the kernel attack from any user process, it restricts all user processes on the running kernel. KPRM also supports the benign user process list that reduces the performance overhead for non-malicious applications. The administrator can manually register the benign flag to avoid KPRM after the execution of the user process.

Additionally, the use case of KPRM assumes that both building processes are performed on the testing environment before the user deploys KPRM to the production environment.

4. Design of KPRM

4.1 Design Concept

In this study, KPRM was designed to achieve the primary requirements for the reduction of the kernel attack surface. More specifically, it is the ability of the kernel protection mechanism to prevent the invocation of vulnerable kernel code and the illegal modification of kernel data.

- **Security requirement of restriction for kernel page references**
  All the kernel code and kernel data are shared in the kernel address space as the kernel page table for every user process. In the kernel address space, kernel features are used and privilege information is stored, simultaneously (e.g., MAC and user identifiers). The design of the mechanism has the aim of preventing the invocation of vulnerable kernel code and prohibiting illegal kernel data modification. This ensures that the adversary's user process on the running kernel is prevented from exhibiting malicious behavior by exploiting kernel vulnerabilities.

© 2022 Information Processing Society of Japan

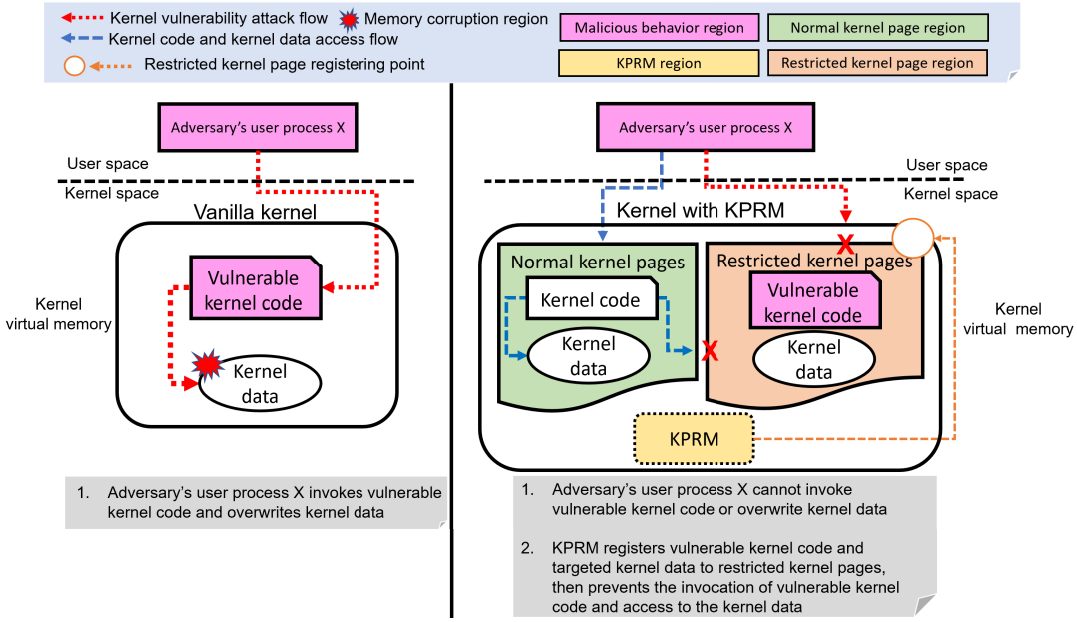

Fig. 2 Overview of the kernel page restriction mechanism (KPRM).

4.2 Kernel Security Capability Challenge

Improving the kernel security capability is the objective of KPRM. The following design rules for the restriction must be fulfilled:

- **Dynamic kernel page reference management**
  The design for KPRM introduces normal and restricted kernel pages to control the kernel address space. Specifically, restricted kernel pages support the assignment of vulnerable kernel code and the kernel data (e.g., function pointer and privilege information). To prevent the execution of vulnerable kernel code, KPRM forcibly unmaps the references of restricted kernel pages for the adversary's user process. This mechanism ensures that normal pages and restricted pages are not in the same kernel address space. This aids in maintaining kernel resilience and prevents the adversary's user process from causing memory corruption using KPRM.

Figure 2 provides an overview of how KPRM implements restriction by allocating vulnerable kernel code and kernel data to restricted kernel pages.

4.3 Restriction Design

KPRM assigns vulnerable kernel code and kernel data to the restricted kernel pages for the adversary's user process. Figure 2 depicts the adversary's user process X on the KPRM kernel. KPRM uses unmapping management from the kernel address space to handle the kernel page references to control the execution and data access privileges on the restricted kernel page.

The adversary's user process is prevented from invoking the vulnerable kernel code and accessing the kernel data on restricted pages.

4.3.1 Kernel Page Types

KPRM provides two types of kernel page structures:

- **Normal kernel page**: This is shared by every user process and kernel task. A normal page contains the kernel code and kernel data.
- **Restricted kernel page**: This is assigned to an adversary's user process. During kernel execution, KPRM forcibly unmaps restricted kernel page references for an adversary's user process.

KPRM also creates a restricted kernel page list and a benign user process list for the kernel. To generate a restricted kernel page list, it can automatically compute a valid page frame number from the virtual address of the kernel code and kernel data. The virtual address of the kernel code is derived from a kernel vulnerability and target kernel data are statically implemented in the kernel. KPRM sets restricted kernel pages when all the vulnerable kernel code and kernel data are identified at the time of kernel booting. Additionally, KPRM manages the restricted kernel page list that stores and deletes a restricted kernel page. The user process also has the benign identifier flag for management by the benign user process list to avoid the restricted kernel page management in the running kernel.

4.3.2 Restricted Kernel Page Object

The restricted kernel page is responsible for assigning the kernel code and kernel data to ensure security capability. The definitions of these two restricted kernel page objects are as follows:

- **Kernel code**: This is a kernel feature component.
- **Kernel data**: This is a function pointer of kernel code and a privilege information variable.

Specifically, KPRM assumes that kernel code is an already known kernel vulnerability that is discovered or reported [14], and that kernel data contains the credential variables of the running user processes.

4.3.3 Timing of Restricted Kernel Page Management

KPRM requires the handling of accessible kernel pages for both the user process and kernel. The design of the handling timing of KPRM in the kernel layer is based on the assumption that KPRM interrupts the user process behavior to unmap restricted kernel pages prior to the system call invocation. Moreover, KPRM maintains all the page table entries of the page table

© 2022 Information Processing Society of Japan

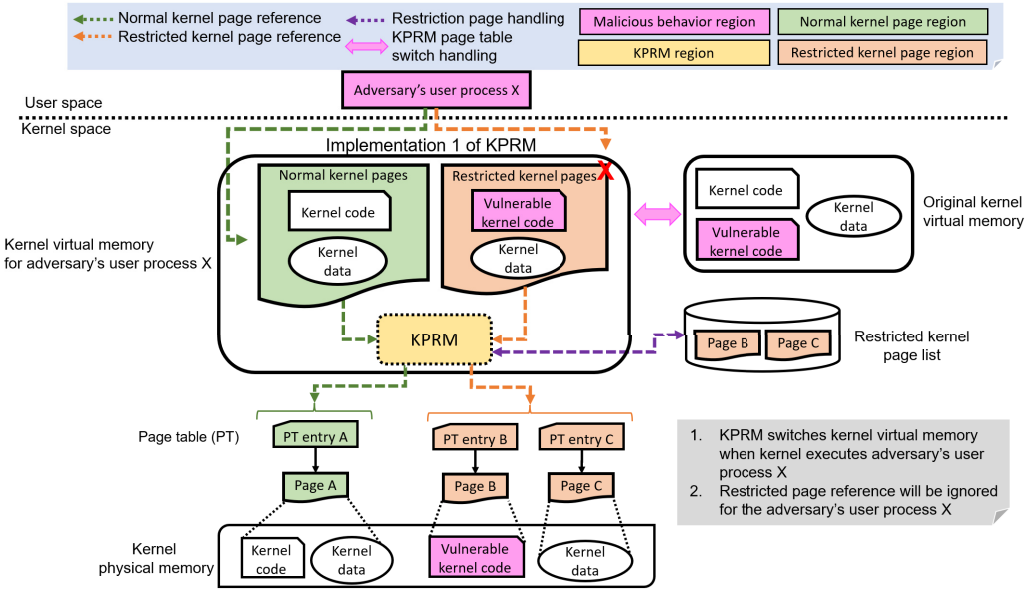

Fig. 3 Implementation 1 of KPRM.

to check whether a kernel page reference matches the restricted kernel page.

4.4 Attack Situations

KPRM manages vulnerable kernel code for the protection of the kernel from memory corruption in the following attack situations.

- **Situation 1**: KPRM does not maintain a restricted kernel page. The vulnerable kernel code and attack targeted kernel data reside on a normal kernel page. The adversary's user process can execute the vulnerable kernel code that can overwrite any kernel data on the normal kernel page.
- **Situation 2**: KPRM assigns the vulnerable kernel code to a restricted kernel page, which the adversary's user process cannot access. When it attempts to execute the vulnerable kernel code, the kernel issues a page fault; KPRM prevents the execution of the vulnerable kernel code by the adversary's user process, which is killed after completion of the page fault handler process.
- **Situation 3**: KPRM registers the attack targeted kernel data on a restricted kernel page. The vulnerable kernel code, however, is on a normal kernel page. If the adversary's user process executes the vulnerable kernel code that tries to override the targeted kernel data, the kernel issues a page fault; KPRM intercepts this page fault, and then kills the adversary's user process to prevent access to the restricted kernel page.

5. Implementation

5.1 Restriction Implementation

The implementations of KPRM on a Linux kernel with the x86_64 are described in this section. KPRM manages the kernel page table that controls the visible kernel pages for an adversary's user process.

Table 1 compares the characteristics, stability, and performance of the KPRM implementations. Stability means the quality of kernel behavior without crashes or errors. **Figure 3** shows that KPRM implementation 1 can prevent the invocation of vulnerable kernel code and protect kernel data. It controls the references of restricted kernel pages on the additional kernel address space of the kernel page table for the adversary's user process. **Figure 4** shows that KPRM implementation 2 can prevent kernel data memory corruption. This requires complex reference handling of restricted kernel pages on the shared kernel address space of one kernel page table.

Table 1 Implementations of KPRM (○ is covered; ● is non-covered).

Item	Implementation 1	Implementation 2
Kernel data protection	○	○
Kernel code restriction	○	●
Stability	Low	High
Performance	High	Low

5.1.1 Restricted Kernel Page Management

Both KPRM implementations on Linux handle the benign identification `benign` flag based on the struct `task_struct` to user process. The KPRM enables the `benign` flag to refer to the application binary's absolute path within the `benign_user_process_list`. This list is manually created in the kernel source code. Additionally, the administrator handles the `benign` flag management via `/proc/pid/kprm`. KPRM also manages the restricted kernel page list `restricted_page_list` that stores the restricted kernel page information. It includes the list of kernel data and virtual addresses, and the list of kernel code with the virtual addresses and function names from kernel vulnerability.

5.1.2 Implementation 1

KPRM implementation 1 adopts an additional kernel page table with the Linux kernel page table structure. Figure 3 shows that KPRM implementation 1 creates the kernel address space of the page table for the kernel and it is restricted to system calls for the user process. The additional kernel page table is the variable `kprm` of `mm_strct` on the struct `task_struct`.

The additional kernel page table duplicates the initial value

© 2022 Information Processing Society of Japan

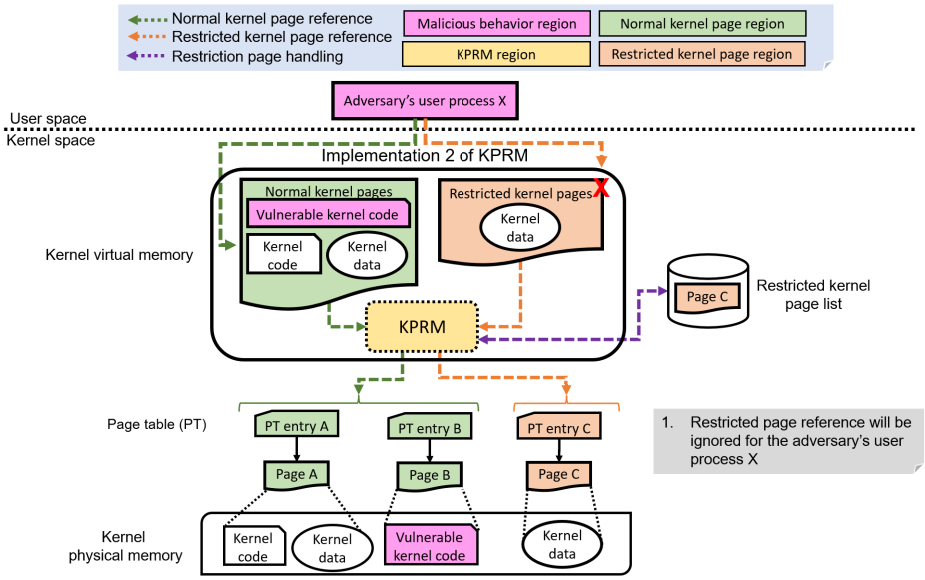

Fig. 4 Implementation 2 of KPRM.

`pgd` of `init_mm` to `kprm` for the user process creation. The KPRM kernel prepares the PCID of TLB and then writes `kprm` in `current` to the CR3 register for system call invocation during the running of the kernel. KPRM also applies the timing of restricted kernel page handling.

The Linux kernel executes a task under the kernel address space constructed from the variable `pgd` of `current` when the kernel receives an asynchronous interruption. To overcome the interruption issue, KPRM implementation 1 switches to the kernel address space from the additional kernel address space `kprm` of `current`. This requires the writing of the CR3 register with the kernel page tables as the variable `pgd` of `current` with PCID of TLB.

5.1.3 Implementation 2

KPRM implementation 2 uses the directory management of the original kernel page table (Fig. 4). The KPRM kernel uses the variable `pgd` of `current` at the system call invocation for the timing of restricted kernel page handling.

The KPRM kernel directory modifies the original kernel page table, which can lead to unstable kernel behavior during interruption processing. For handling an interruption, KPRM implementation 2 affixes the restricted kernel page references to the original kernel page table for kernel tasks execution.

The restriction of implementation 2 cannot prevent the invocation of kernel code because implementation 2 only unmaps the page of kernel data from the original kernel page table.

5.1.4 Page Fault

During the execution of user processes on the Linux kernel with both implementations of KPRM, Linux kernels intercept the page fault with the `do_page_fault` or the `do_double_fault` function. These functions indicate the virtual address of the origin of the page fault. Then, the KPRM kernel inspects the details of the page fault to determine whether the virtual address is available to the user process. It further determines the access decision and maps the restricted kernel page to the current kernel page table when the user process is valid. Otherwise, it uses `force_sig_info` to send a SIGKILL to force termination of the user process.

5.1.5 Restricted Kernel Page Handling

The restricted kernel page handling process is common to both implementations of KPRM. The KPRM kernel automatically adopts the handling for the adversary's user process. The handling timing is hard-coded into the KPRM kernel that uses the handling steps before the system call invocation in the `entry_SYSCALL_64` function.

The handling mechanism identifies the page number from the virtual address and subsequently unmaps the reference of the restricted kernel page from the target kernel page table with the `remove_pagetable` function. The restricted kernel page is also unmapped from the direct mapping region. Additionally, the KPRM kernel manages the page fault and trap handling related to a restricted kernel page.

Figure 5 shows the handling mechanism for an adversary's user process. The restricted kernel page handling process is as follows.

(1) The preparation of KPRM that kernel creates the restricted kernel page list and the benign user process list at the time of kernel booting.
  (a) The KPRM kernel registers the virtual address of kernel codes and kernel data into the restricted kernel page list from the list of restricted kernel code and the list of protected kernel data.
(2) An adversary's user process starts the system call execution, following which the KPRM kernel traps the system call routing and moves to KPRM kernel processing.
(3) The KPRM kernel determines the adversary's user process identity with the `benign` flag set to off, and then the KPRM kernel restores the restricted kernel pages from the restricted kernel page list and subsequently unmaps all the references of the kernel pages in the kernel page table.
(4) The system call is invoked along with access to the kernel code of the system call routine, following which the kernel

© 2022 Information Processing Society of Japan

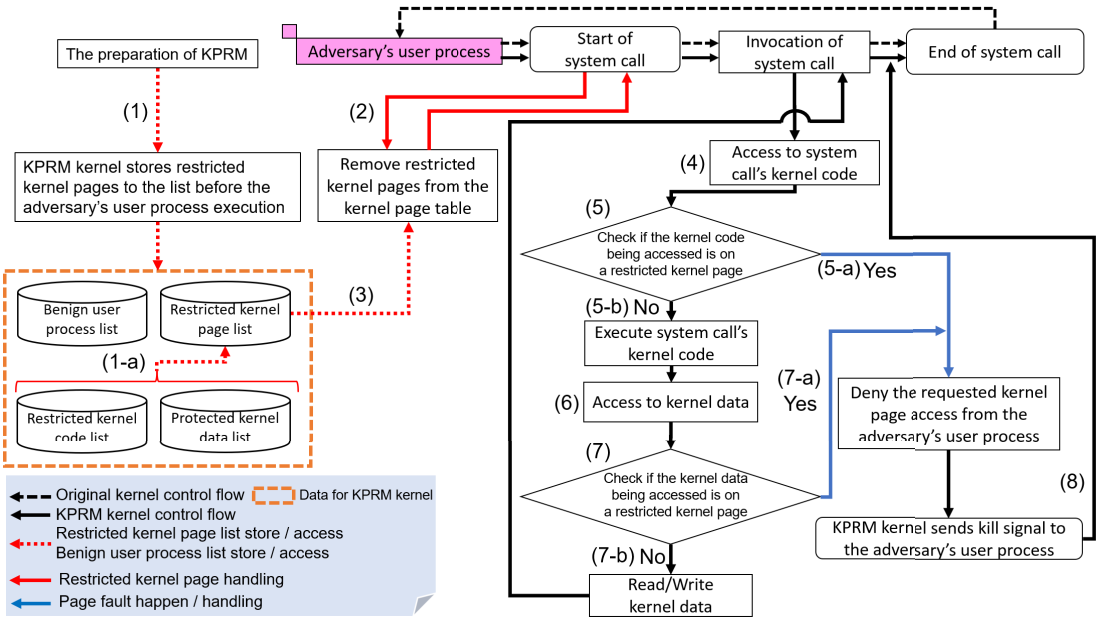

Fig. 5 KPRM for the adversary's user process.

issues the page fault, which the KPRM kernel traps.

(5) The KPRM kernel identifies the virtual address of the page fault that indicates the virtual address of the kernel code on the restricted kernel page.
  (a) In the case of invalid access to the restricted kernel page, the KPRM kernel denies access to the user process.
  (b) If the access is valid, the KPRM kernel maps the reference of restricted kernel page of the kernel code to the kernel page table for the user process to continue.

(6) The system call routine's kernel code accesses kernel data; then, the kernel also issues the page fault, and the KPRM kernel traps the page fault.

(7) The KPRM kernel identifies the virtual address of the page fault that indicates the virtual address of kernel data on the restricted kernel page.
  (a) In the case of invalid access to kernel data on the restricted kernel page, the KPRM kernel denies access to the user process.
  (b) If the access is valid, KPRM kernel maps the restricted kernel page of kernel data to the kernel page table for the user process to continue.

(8) If the KPRM kernel identifies an adversary's process, access is not allowed on the restricted kernel page, and KPRM kernel sends a signal to the user process.

5.2 Case Study

5.2.1 Vulnerable Kernel Code Invocation and Memory Corruption

For a case study of kernel memory corruption exploiting kernel vulnerability, **Fig. 6** shows the adversary's user process employing the CVE-2017-16995 [15] PoC code to invoke the vulnerable kernel code during the extended Berkeley Packet Filter (eBPF) system call. The vulnerable kernel code is the `map_update_elem` function of `kernel/bpf/syscall.c`. The adversary's user process tries to invoke the vulnerable kernel code to modify the virtual address of the privilege information of kernel data on the kernel address space. After successfully inducing privilege escalation, the adversary's user process executes the shell with administrator privileges.

To prevent such attacks, KPRM implementation 1 assigns the `map_update_elm` function as vulnerable kernel code to the restricted kernel page. Then, the adversary's user process cannot invoke the vulnerable kernel code. Additionally, both KPRM implementations assign kernel data (e.g., privilege information) to the restricted kernel page. The adversary's user process utilizing the PoC code cannot modify the restricted kernel page. Next, a page fault of the restricted kernel pages occurs. Both KPRM implementations can intercept this page fault to determine whether to send the SIGKILL signal to the attack process in the page fault handler.

To summarize, Fig. 6 shows how the KPRM implementations prevent the adversary's user process as follows:

(1) The KPRM kernel starts the construction process of a restricted kernel list.
  (a) The KPRM kernel prepares a restricted kernel list containing restricted kernel pages with vulnerable kernel code and kernel data before adversary process execution.

(2) The adversary executes CVE-2017-16995 PoC code as a user process on the KPRM kernel.
  (a) The KPRM kernel removes the references of the restricted kernel page on the kernel address space. This step is also inserted into steps (3-a) and (4-a) by the KPRM kernel.
  (b) The kernel starts the fork system call and invokes the fork function.
  (c) The kernel creates user, kernel, and restricted virtual memory for the user process and finishes the fork system call.

(3) The user process invokes the first eBPF system call with the eBPF map creation and managing option.
  (b) The kernel starts the eBPF system call that creates the ker-

© 2022 Information Processing Society of Japan

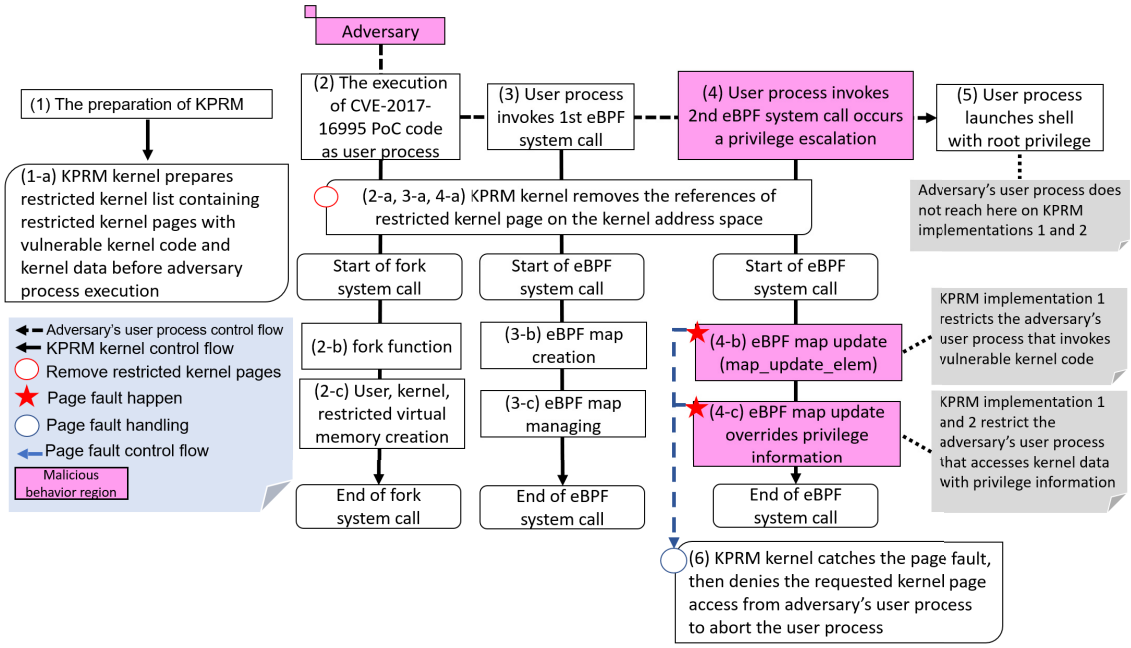

Fig. 6 Handling of restricted kernel page reference for the actual kernel vulnerability.

nel data of the eBPF map.

(c) The kernel sets the kernel data of the eBPF map using the managing option and finishes the first eBPF system call.

(4) The user process invokes the second eBPF system call that causes privilege escalation.

(b) The kernel initiates the eBPF system call that tries to execute the eBPF map update contained map_update_elem kernel code. It is vulnerable kernel code that is unmapped by the KPRM kernel at step (3-a). Then, a page fault is happen. KPRM implementation 1 restricts the adversary's user process that invokes the vulnerable kernel code.

(c) The kernel invokes the eBPF map update that overrides privilege information, however, privilege information is unmapped by the KPRM kernel at step (3-a), then the page fault is happen. Both KPRM implementations restrict the adversary's user process trying to accesse kernel data with privilege information.

(5) The user process attempts to launch a shell program with root privilege if the privileges escalation succeeds. However, the adversary's user process does not reach this point with either KPRM implementations owing to the page fault is happen at step (4-b) in KPRM implementation 1 or at step (4-c) in KPRM implementations 1 and 2. The KPRM kernel kills the adversary's user process at step (6).

(6) The KPRM kernel catches the page fault, then denies the requested kernel page access from the adversary's user process to abort the user process.

6. Evaluation

6.1 Security Capability

The security capability assessment of KPRM is validated by the prevention of kernel memory corruptions.

(1) **Prevention of vulnerable kernel code access and kernel memory corruption**: The evaluation of the security capability of KPRM was based on whether the KPRM kernel can prevent the execution of vulnerable kernel code and protect kernel data of privilege information when an adversary's user process tries to exploit an actual kernel vulnerability when invoking the system call.

6.2 Performance Measurement

The objectives of the performance evaluation were to measure the overhead cost for the user process, a vanilla kernel, and the KPRM kernel.

(1) **Measurement of the system calls invocations overhead**: To measure the implementation effect for determining the feasibility of the KPRM kernel, LMbench was used to calculate the overhead of system call invocation latency.

(2) **Measurement of application overhead**: The performance overhead for the application was measured using ApacheBench.

(3) **Measurement of kernel processing overhead**: The processing performance overhead of the KPRM kernel was measured using the Linux kernel compile time.

6.3 Evaluation Environment

6.3.1 Implementation

KPR was implemented for the Linux x86_64 kernel. To evaluate the practical security capability of KPRM, an actual kernel vulnerability was customized to allow memory corruption of Linux kernel 4.4.114 with CVE-2017-16995 [15]. The performance measurement requires stable behavior for the entire Linux kernel 5.0.0. There is no difference in performance overhead between Linux kernel 4.4.114 and 5.0.0. The Linux distribution used was Debian 10.0; the CVE-2017-16995 PoC code was modified to handle any kernel address space. The KPRM implementations required 40 source files and 1,832 lines for Linux kernels 4.4.114 and 5.0.0.

© 2022 Information Processing Society of Japan

```
// PoC code running, process id is 1642
1.  www-data$ ./cve-2017-16995
2.  [*] creating bpf map
3.  Killed

// Kernel log message
4.  [  91.425545] target system call, pid : 1642
5.  [  91.425884] sysnum: 0000000000000141
// ffffffff81131e00 is the virtual address of map_update_elem
6.  [  91.426586] remove_exclusive_pages() : ffffffff81131e00
//  page fault is happen
7.  [  91.426925] error_handling() pid: 1642,
8.  [  91.426966] killing target pid: 1642
```

Red text is the protection and prevention point of kernel memory corruption

Fig. 7 Attack prevention case of vulnerable kernel code invocation.

```
1.  www-data$ ./cve-2017-16995
2.  [*] creating bpf map
3.  [*] sneaking evil bpf past the verifier
4.  [*] creating socketpair()
5.  [*] attaching bpf backdoor to socket
6.  [*] skbuff => ffff88001d3c8b00
7.  [*] Leaking sock struct from ffff88001d1b6f00
8.  [*] Sock->sk_rcvtimeo at offset 472
9.  [*] Cred structure at ffff88001c46df00
10. [*] UID from cred structure: -33686019,
        matches the current: -33686019
11. [*] hammering cred structure at ffff88001c46df00
12. [*] credentials patched, launching shell...
13. $ id
14. uid=4261281277 gid=4261281277 groups=4261281277,
15. 33(www-data)
```

Red text is the protection and prevention point of kernel memory corruption

Fig. 8 Attack prevention case of kernel data access.

6.3.2 Equipment

The evaluation environment for the system performance and a web server was executed on a physical machine equipped with an Intel (R) Core (TM) i7-7700HQ (2.80 GHz, x86_64) processor with 16 GB memory. The web client machine was equipped with an Intel (R) Core (TM) i5-4200U (1.6 GHz, x86_64) processor with 8 GB memory, running Windows 10. The network environment for the application benchmark was a 1 Gbps hub supporting different ports for the server and client machines.

6.4 Security Capability Experiment

6.4.1 Prevention of Vulnerable Kernel Code Access and Kernel Memory Corruption

The evaluation of practical security capability is based on whether the invocation of vulnerable kernel code is prevented and the privilege kernel data are protected in KPRM implementations from the eBPF kernel attack with CVE-2017-16995 [15]. The modified PoC code can overwrite any kernel virtual address. The PoC code invokes the `sys_bpf` system call that calls the `map_update_elem` kernel code to execute the malicious code; then, it tries to modify the `cred` variable of the privilege kernel data of the running user process. KPRM Implementation 1 uses the modified PoC code that executes the restricted kernel page containing the vulnerable kernel code `map_update_elem`. KPRM Implementation 2 uses the same PoC code that accesses targeted privilege kernel data of the `cred` variable.

Figure 7 shows that the adversary's user process attempts to invoke the `map_update_elem` function during the `sys_bpf` system call processing at line 5. KPRM proceeds with the restricted kernel page handling at line 6, after which it can intercept the page fault that contains the virtual address of the vulnerable kernel code. In this situation, the KPRM kernel determines the running process that requests invalid access to the restricted kernel page and then sends SIGKILL to terminate the adversary's user process.

Figure 8 shows the prevention success case of memory corruption. The user `www-data` with user id 33 also executes the PoC code at line 1. The adversary's user process tries to modify the `cred` struct of the privilege kernel data to the root with user id 0 in lines 9 to 11. The KPRM kernel automatically restricts the access of the adversary's user process to the privilege kernel data on the restricted kernel page. Finally, the adversary's user process runs the shell program without administrator privileges in line 14.

From the results, KPRM implementations can prevent vulnerable kernel code invocation and protect the privilege kernel data to prevent memory corruption from the eBPF kernel attack with CVE-2017-16995 with stable behavior when running the kernel and user process.

Table 2 One time system call invocation overhead of KPRM kernel (μs).

System call	Vanilla kernel	Implementation 1	Implementation 2
open / close	0.532	1.187 (123.12%)	1.119 (110.34%)
read	0.276	0.896 (224.64%)	0.838 (203.63%)
write	0.238	0.856 (259.66%)	0.796 (234.45%)
stat	0.547	1.251 (128.70%)	1.173 (114.44%)
fstat	0.291	0.938 (222.34%)	0.873 (200.00%)

Table 3 ApacheBench overhead of KPRM kernel (μs).

File size (kB)	Vanilla kernel	Implementation 1	Implementation 2
1	599.143	623.667 (4.093%)	617.167 (3.008%)
10	764.250	784.250 (2.617%)	773.333 (1.188%)
100	2,443.714	2,509.167 (2.678%)	2502.667 (2.412%)

6.5 Measurement of Performance Overhead

6.5.1 Measurement of System Calls Overhead

For the measurement of the performance overhead, a comparison between a vanilla kernel and the KPRM kernel was conducted. The benchmark software LMbench was executed 10 times to determine the average system call overhead. The result obtained was that the overhead time of the KPRM kernel incurs a kernel page handling cost for each system call invocation.

LMbench invokes the various system call counts (i.e., open / close is invoked two times and the remaining system calls are invoked only once). **Table 2** shows that the stat system call has the highest overhead for KPRM implementations 1 and 2 (0.703 μs and 0.626 μs), whereas the write system call has the lowest overhead (0.617 μs and 0.557 μs).

6.5.2 Measurement of Application Overhead

Measurement of the web application process overhead for the vanilla kernel and KPRM kernel was conducted. The web application process used was an Apache 2.4.25 web server. The benchmark software was ApacheBench 2.4 for the web client. The network environment was 1 Gbps. ApacheBench 2.4 calculated the HTTP download request average for HTTP accesses. The benchmark configuration of ApacheBench sent 100,000 HTTP accesses, then the downloaded file for one connection. The benchmark adopted files sizes of 1 kB, 10 kB, and 100 kB. The list in **Table 3** shows that KPRM implementation 1 has an average overhead of 2.617% to 4.093% and KPRM implementation 2

© 2022 Information Processing Society of Japan

Table 4 Kernel building overhead of KPRM kernel (s).

Vanilla kernel	Implementation 1	Implementation 2
5926.644 (s)	6072.413 (2.459%)	6056.629 (2.193%)

has an average overhead of 1.188% to 3.008% for each file download access.

6.5.3 Measurement of Kernel Processing Overhead

To evaluate kernel processing overhead, the kernel compile times of the vanilla kernel and the KPRM kernels were compared. The kernel compile overhead measures the processing time of specific applications (i.e., compiler and linker) in a general application execution environment. The compile target was Linux kernel 5.0.0 source code with Debian 10.0 kernel configuration (e.g., default `.config` file). The kernel compilations were executed five times to determine the average kernel processing time. The performance scores are presented in **Table 4**. KPRM implementations 1 and 2 had kernel compile overheads of 2.459% and 2.193%, respectively.

7. Discussion

7.1 Kernel Resilience

The KPRM kernel successfully registered the vulnerable kernel code and kernel data to the restricted kernel page. Then, the KPRM kernel prevented the PoC of the eBPF kernel vulnerability attack that tried to invoke the vulnerable kernel code and modify the credential information. The page fault indicated the access violation from the access and execution request of the user process at the kernel layer. It is known that the KPRM kernel intercepts the page fault before the malicious program can exploit the kernel vulnerability to modify the kernel data. Therefore, the KPRM kernel can protect the kernel data and disable invocation of the vulnerable kernel code by the malicious program.

Additionally, KPRM deals with the threat of already known kernel vulnerabilities by maintaining kernel memory integrity before memory corruption occurs owing to kernel subverting.

7.2 Performance Evaluation

The benchmark measurement results indicate that the KPRM implementations increase the performance cost by requiring additional processing time from the running kernel. To measure these results, LMbench, ApacheBench, and a kernel compile that correctly calculate the cost of the overhead of the two KPRM implementations were used.

The different overhead costs of the two KPRM implementations show that KPRM implementation 1 requires page table switching, CR3 register update, and TLB flush cost. KPRM implementation 2 does not require page table switching. Although KPRM implementation 1 adopts the PCID of TLB for the overhead reduction, it retains the CPU cycles for the CR3 update without a TLB flush. The common overhead cost of the KPRM implementations arises from the requirement of searching for the restricted kernel pages of the kernel page table during page table walk. In addition, the KPRM implementations forcibly unmap restricted kernel pages, which leads to additional processing time at the kernel layer.

The inspection of KPRM implementations helped to achieve better performance. Although the kernel page table size is not reduced at the kernel, the improvement of implementations removes restricted kernel pages from the kernel page table at the time of process creation. It avoids the page table walk at the time of the system call invocation timing. Additional improvements will need to consider suppressing the number of restricted kernel pages and types of system calls to achieve a reduction in overhead.

Table 5 Portability consideration of KPRM for OSs (✓ is supported; • is available on the x86_64).

OS	Page table structure	KPRM
Linux	✓	✓
FreeBSD	✓	•
XNU kernel	✓	•

7.3 Limitation

7.3.1 Design Limitation

The design limitations of KPRM must be considered. The primary limitation is that the registration of the vulnerable kernel code and protected kernel data are statically managed at kernel boot time.

To ensure quick response for the kernel vulnerability, dynamic registration of vulnerable kernel code and the control of the benign user process is required in the design.

7.3.2 Security Limitation

The security limitations must also be considered. KPRM is assumed that kernel vulnerabilities should be registered in the CVE list to restrict vulnerable kernel code. The adversary's user process can still invoke unknown vulnerable kernel code and modify the kernel data on the normal kernel page. KPRM does not directly protect the original kernel address space from attack. Instead, it relies on the restricted kernel page to achieve the prevention of kernel memory corruption. Therefore, KPRM assumes that the vulnerable kernel codes and attack targeted kernel data are suitably registered on the restricted kernel page.

7.4 Portability

7.4.1 OS Kernel

The Linux implementations of KPRM with the x86_64 are a reference at the kernel layer. The applicability of KPRM in comparison with other OS kernels is limited by the condition that the OS kernel must satisfy the page table structure to manage virtual memory (**Table 5**). Modern OS kernels manage the page table entries and page table combination by handling the user and kernel virtual memory region. FreeBSD has a virtual memory system [16] and XNU kernel has kernel memory management [17].

In this study, the portability of KPRM was carefully examined in relation to other OS kernels. FreeBSD has implemented virtual address space (e.g., `vmspace`), comprising `vm_map` and `vm_map_entry` which constructs the page table, then `vm_page` is the page object at the kernel layer [18] The XNU kernel also has similar page table structure implementation for user and kernel modes [17]. Therefore, the restricted kernel page approach is available and provides support for both OS kernels.

7.4.2 Architecture

The consideration of other CPU architectures can be used to

© 2022 Information Processing Society of Japan

Table 6 Portability consideration of KPRM for architectures (✓ is supported; • is available).

Arch	Page table register	Paging	KPRM
x86_64	CR3	4 / 5 level paging	✓
ARM	TTBRs	TTBRn_ELm	•
RISC-V	satp	Sv32 / Sv39 / Sv48	•

validate the use of page table structure for the virtual memory (**Table 6**).

The x86_64 shows that certain hardware specifications (e.g., 4 / 5 level paging) support the restricted kernel page of KPRM implementations. ARM has translation table base registers (TTBR0 / TTBR1) and the exception levels (EL0 / 1 / 2 / 3) that provide page table structure for each privilege level [19]. Additionally, RISC-V has supervisor address translation and protection (satp) register that specifies the virtual address length and translation modes (Sv32 / Sv39 / Sv48) for page table structure [20]. Therefore, KPRM can be realized in both CPU architectures that are available with two types of kernel page structures (e.g., normal and restricted kernel pages).

8. Related Work

Kernel security research has yielded multiple memory isolation and memory protection mechanisms against potential threats and practical attack techniques. **Figure 9** gives an overview of the kernel memory security research taxonomy, summarizing previous security mechanisms.

8.1 Memory Isolation

Kernel memory isolation is a runtime memory separation technique that uses hardware or software against directory attack threats from the kernel or user layers.

Isolation at Hardware: The hardware features support a memory isolation mechanism. A trusted execution environment (TEE) executes the kernel in the secure memory region to mitigate kernel attacks from a non-secure memory region [21], [22]. Additionally, the hardware includes the core code integrity measurement architecture Sprobes that adopts CPU security features and a TEE that ensures kernel integrity [23]. IskiOS uses a CPU memory protection feature that restricts a specific memory region related to the kernel virtual memory [24].

The approach with hardware virtualization is available for the separation of the kernel virtual memory to achieve low overhead [25]. KHide restricts the granularity of software diversity techniques for kernel code and kernel data with hardware virtualization [26]. Moreover, xMP provides switching of the visible virtual memory region between the user and the kernel modes for the guest OS with the hypervisor [27].

Isolation at Software: The software approach also supports the memory isolation in the kernel layer. Kernel page table isolation (KPTI) that provides dedicated page tables for the user and kernel modes to mitigate meltdown side-channel attacks [28]. Moreover, Proclocal allocates dedicated pages to preserve the kernel data for each user process [5]. A third software feature is SCI, which creates an isolated page table to execute the kernel codes of system calls during kernel processing [6].

Fault Tolerance: The fault tolerance mechanism separates

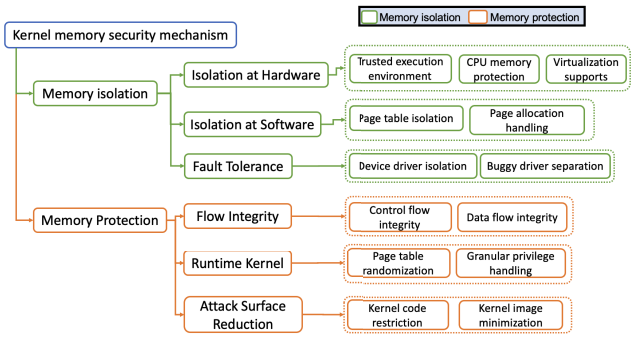

Fig. 9 Overview of the kernel memory security mechanism taxonomy.

the execution of the device drivers (e.g., user space driver) from main kernel processing [29], [30]. iKernel adopts the virtual machine monitor feature that separates buggy devices on virtual machines [31]. SIDE creates a dedicated page table for each driver with an interaction mechanism between kernel and drivers. These methods focus on the stability of kernel behavior to protect the main kernel feature from malicious or buggy device drivers [32].

8.2 Memory Protection

Flow Integrity: Memory protection of the kernel requires the flow integrity to ensure prevention of memory corruption. The CFI prevents malicious behavior behind benign program flow [3]. KCoFI corresponds with CFI for the kernel processing that requires the asynchronous behavior to handle the interruption and context switch [33]. Additionally, the data flow integrity verifies the benign data flow to prevent memory corruption [34].

Runtime Kernel: To mitigate memory corruption, randomization and granular privilege handling is implemented to achieve kernel resilience and enforce the runtime kernel protection. Randomization of the kernel page table position protects the structure from malicious activity in the kernel layer [35]. XPFO manages the attribute separation of the page between the user and kernel modes to protect direct mapping region attacks [2]. Additionally, kRˆX controls the exclusive mechanism along with the access and the execution privilege of the kernel code and kernel data [36].

Attack Surface Reduction: Reducing the kernel attack surface restricts the visible virtual memory region for user processes. PerspicuOS demonstrates privilege minimization to isolate the kernel mechanism for hardware management [37]. kRazor manages the list of kernel codes visible for user processes [38]. Kernel attack surface reducing (KASR) handles and controls the kernel page table to reduce the set of kernel codes and kernel data for each user program execution [39]. Additionally, Multik customizes the kernel image to create the profile that contains the necessary kernel code for each application [40].

8.3 Comparison

With the aim of determining which mechanism fulfills the maximum kernel protection requirements, **Table 7** compares the security features of KPRM and types of target vulnerability with those of other kernel memory protection mechanisms [5], [6], [26], [27], [36], [37]. KPRM satisfies most of the security requirements for kernel code and kernel data, by providing the page reference management to prevent the actual attack from the user

© 2022 Information Processing Society of Japan

Table 7 Comparison of kernel memory protection features (✓ is supported; △ is partially supported) and types of target vulnerability (C: code execution, M: memory corruption, B: bypass feature, I: gain information, P: gain privileges) [14] (Table A·1 lists the types of target vulnerability).

Feature	**PerspicuOS** [37]	**KHide** [26]	**kRˆX** [36]	**xMP** [27]	**Proclocal** [5]	**SCI** [6]	**KPRM**
Kernel data protection	△	△	✓	✓	✓		✓
Kernel code restriction	△	✓	✓			✓	✓
Page reference management		△			✓	△	✓
Access restriction for user process			✓	✓	△	✓	✓
Types of target vulnerability	B	C	C, I	M, B, P	I	C	C, M, B, P

process for the running kernel.

PerspicuOS implements a nested kernel components model of privilege deduction mechanisms that ensures isolation between trusted and untrusted kernel components [37]. KHide adopts hardware virtualization features to enforce the granularity of diversification for kernel code and kernel data for the kernel deployment on the guest OS environment with hypervisor [26]. Moreover, kRˆX supports the design of exclusive privilege management that directly controls the separation of readable and executable privileges for the kernel code and kernel data in the kernel memory [36]. These approaches provide static customized kernel page tables. To dynamically prevent illegal memory corruption, KPRM manages the kernel page references to isolate the vulnerable kernel code and the targeted kernel data from the adversary's user processes in the running kernel.

In another approach, xMP provides dynamic switching of the customized visible memory region between the user mode and the kernel mode for the guest OS with a hypervisor [27]. KPRM restricts the adversary's user processes that refer to a limited region of kernel memory at the kernel layer without hypervisor.

Proclocal provides a dedicated kernel page to allocate the reserved kernel memory region for the user process [5] and SCI prepares temporary page tables to execute the kernel code during system call processing [6]. Although a combination of Proclocal and SCI offers security capabilities similar to that of KPRM, the protection provided by Proclocal is limited to the kernel data of specific kernel components and the memory isolation feature of SCI requires full kernel page mapping. The combination of Proclocal and SCI cannot, however, prevent the vulnerable kernel code invocation when the user process is able to begin the attack at the kernel layer.

In contrast, the design and architecture of KPRM is such that it supports finer granularity for the control of the kernel memory. KPRM completely isolates vulnerable kernel code from attack targeted kernel data. It relies on the unmapping of the kernel page of vulnerable kernel code at the beginning of the kernel attacking flow to ensure that no access is allowed to the adversary's user process. However, KPRM requires the manual identification process of vulnerable kernel code from kernel vulnerabilities using a CVE list.

In addition, Table 7 lists KPRM mitigation capabilities for most types of target vulnerability. However, KPRM cannot mitigate the gain information that forcibly accesses kernel memory. Therefore, the combination of other kernel memory protection mechanisms is necessary to mitigate other types of target vulnerability.

9. Conclusion

Kernel memory corruption leads to privilege escalation and the bypass of security features, and thus necessitates the design of kernel protection mechanisms. These protection mechanisms focus on mitigating and preventing the actual threat to the running kernel. Stack monitoring, CFI, KASLR, and KPTI are kernel attack prevention countermeasures. Two other important protection mechanisms are Proclocal and SCI. However, although they reduce the kernel attack surface available to the adversary's user process, vulnerable kernel code and kernel data still share the same kernel address space, which can lead to security being compromised.

In this paper, the proposed KPRM novel security mechanism provides dynamic restriction of kernel code and kernel data for the adversary's user process. KPRM provides the feature of restricted kernel pages to manage vulnerable kernel code and attack targeted kernel data. It dynamically unmaps restricted kernel pages from the kernel page table for the adversary's user process. Subsequently, vulnerable kernel code and the code or kernel data are isolated in the kernel address space to reduce the kernel attack surface. Therefore, the adversary's user process cannot invoke vulnerable kernel code and access restricted kernel data on the running kernel. Consequently, kernel memory corruption is completely prevented. The evaluation result of the KPRM kernel demonstrated that it can prevent vulnerable kernel code invocation, and finally protect privilege data from kernel memory corruption. In addition, the performance overhead of the maximum system call invocations was 0.703 μs. The overhead of a web client program averages between 1.188% to 4.093% for HTTP download access of 100,000 HTTP sessions. The implementations of KPRM recorded kernel compile time overheads of only 2.459% and 2.193%.

Acknowledgments This work was partially supported by the Japan Society for the Promotion of Science (JSPS) KAKENHI Grant Number JP19H04109 and JP22H03592. Hiroki's contributions contained in the paper was done when he belonged to SECOM Co., Ltd.

References

[1] Chen, H., Mao, Y., Wang, X., Zhow, D., Zeldovich, N. and Kaashoek, F.M.: Linux kernel vulnerabilities - state-of-the-art defenses and open problems, *Proc. 2nd Asia-Pacific Workshop on Systems*, pp.1–5 (online), DOI: 10.1145/2103799.2103805 (2011).

[2] Kemerlis, P.V., Polychronakis, M. and Kemerlis, D.A.: ret2dir: Rethinking Kernel Isolation, *Proc. 23rd USENIX Conference on Security Symposium*, pp.957–972 (online), DOI: 10.5555/2671225.2671286 (2014).

[3] Abadi, M., Budiu, M., Erlingsson, U. and Ligatti, J.: Control-flow in-

© 2022 Information Processing Society of Japan

tegrity principles, implementations, and applications, *Proc. 12th ACM Conference on Computer and Communications Security*, pp.340–353 (online), DOI: 10.1145/1609956.1609960 (2005).

[4] Shacham, H., Page, M., Pfaff, B., Goh, E., Modadugu, N. and Boneh, D.: On the effectiveness of address-space randomization, *Proc. 11th ACM Conference on Computer and Communications Security*, pp.298–307 (online), DOI: 10.1145/1030083.1030124 (2004)

[5] Hillenbrand, M.: Process-local memory allocations for hiding KVM secrets, LWN.net (online), available from ⟨https://lwn.net/Articles/791069/⟩ (accessed 2019-08-08).

[6] Rapoport, M.: x86: Introduce system calls address space isolation, LWN.net (online), available from ⟨https://lwn.net/Articles/786894/⟩ (accessed 2019-08-08).

[7] Mulnix, D.: Intel® Xeon® Processor D Product Family Technical Overview, Intel (online), available from ⟨https://software.intel.com/en-us/articles/intel-xeon-processor-d-product-family-technical-overview⟩ (accessed 2018-08-10).

[8] Kuzuno, H. and Yamauchi, T.: KPRM: Kernel Page Restriction Mechanism to Prevent Kernel Memory Corruption, *Proc. 16th International Workshop on Security*, LNCS, Vol.12835, pp.45–63 (online), DOI: 10.1007/978-3-030-85987-9_3 (2021).

[9] Security-enhanced Linux, NSA (online), available from ⟨http://www.nsa.gov/research/selinux/⟩ (accessed 2019-05-22).

[10] Nexus 5 Android 5.0 - Privilege Escalation, Exploit Database (online), available from ⟨https://www.exploit- db.com/exploits/35711/⟩ (accessed 2019-05-21).

[11] super fun 2.6.30+/RHEL5 2.6.18 local kernel exploit, grsecurity (online), available from ⟨https://grsecurity.net/~spender/exploits/exploit2.txt⟩ (accessed 2019-05-21).

[12] Bovet, P.D. and Cesati, M.: Understanding the Linux kernel, O'Reilly Media, Inc., 3rd edition (2005).

[13] Intel Corporation: Intel(R) 64 and IA-32 Architectures Software Developer's Manual, Intel (online), available from ⟨https://www.intel.co.jp/content/www/jp/ja/architecture-and-technology/64-ia-32-architectures-software-developer-system-programming-manual-325384.html⟩ (accessed 2021-05-05).

[14] Linux Vulnerability Statistics, CVE detail (online), available from ⟨https://www.cvedetails.com/vulnerabilities-by-types.php⟩ (accessed 2021-05-05).

[15] CVE-2017-16995, MITRE (online), available from ⟨https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-16995⟩ (accessed 2019-07-10).

[16] The FreeBSD documentation project, FreeBSD architecture handbook, FreeBSD (online), available from ⟨https://www.freebsd.org/doc/en_US.ISO8859-1/books/arch-handbook/⟩ (accessed 2019-08-08).

[17] XNU Source Tree, Apple Inc. (online), available from ⟨https://github.com/apple/darwin-xnu/tree/main/osfmk/vm⟩ (accessed 2021-09-16).

[18] McKusick, K.M., Bostic, K., Karels, J.M. and Quarterman, S.J.: The Design and Implementation of the 4.4BSD Operating System, FreeBSD (online) available from ⟨https://docs.freebsd.org/en/books/design-44bsd/⟩ (accessed 2021-09-16).

[19] ARMv8-A Address Translation Version 1.0, ARM Inc. (online), available from ⟨https://static.docs.arm.com/100940/0100/armv8_a_addresstranslation_100940_0100_en.pdf⟩ (accessed 2021-09-16).

[20] Waterman, A. and Asanovic, K.: The RISC-V Instruction Set Manual Volume II: Privileged Architecture, Document Version 20190608-Priv-MSU-Ratified, RISC-V (online), available from ⟨https://riscv.org/technical/specifications/privileged-isa/⟩ (accessed 2021-08-21).

[21] Lee, D., Kohlbrenner, D., Shinde, S., Asanović, K., Song, D.: Keystone: An open framework for architecting trusted execution environments, *Proc. 15th European Conference on Computer Systems*, pp.1–16 (online), DOI: 10.1145/3342195.3387532 (2020).

[22] Marcela, S.M., Michael, J.F. and Mic, B.: EnclaveDom: Privilege separation for large-TCB applications in trusted execution environments, Arxiv (online), available from ⟨https://arxiv.org/abs/1907.13245⟩ (accessed 2020-12-08).

[23] Ge, X., Vijayakumar, H. and Jaeger, T.: Sprobes: Enforcing kernel code integrity on the trustzone architecture, *Proc. 3rd Workshop on Mobile Security Technologies* (2014).

[24] Gravani, S., Mohammad, H., Criswell, J. and Scott, L.M.: IskiOS: Lightweight defense against kernel-level code-reuse attacks, Arxiv (online), available from ⟨https://arxiv.org/abs/1903.04654⟩ (accessed 2020-12-08).

[25] Hua, Z., Du, D., Xia, Y., Chen, H. and Zang, B.: EPTI: Efficient defence against meltdown attack for unpatched VMs, *Proc. 2018 USENIX Annual Technical Conference*, pp.255–266 (online), DOI: 10.5555/3277355.3277380 (2018).

[26] Gionta, J., Enck, W. and Larsen, P.: Preventing kernel code-reuse attacks through disclosure resistant code diversification, *Proc. 2016 IEEE Conference on Communications and Network Security*, pp.189–197 (online), DOI: 10.1109/CNS.2016.7860485 (2016).

[27] Sergej, P., Marius, M., Seyedhamed, G., Vasileios, P.K. and Michalis, P.: xMP: Selective Memory Protection for Kernel and User Space, *Proc. 41st IEEE Symposium on Security and Privacy*, pp.563–577 (online), DOI: 10.1109/SP40000.2020.00041 (2020).

[28] Gruss, D., Lipp, M., Schwarz, M., Fellner, R., Maurice, C. and Mangard, S.: KASLR is dead: Long live KASLR, *Proc. 2017 International Symposium on Engineering Secure Software and Systems*, Vol.10379, No.3, pp.161–176 (online), DOI: 10.1007/978-3-319-62105-0_11 (2017).

[29] Herder, J., Bos, H., Gras, B., Homburg, P. and Tanenbaum, A.: Fault Isolation for Device Drivers, *Proc. 39th Annual IEEE / IFIP International Conference on Dependable Systems and Networks* (online), DOI: 10.1109/DSN.2009.5270357 (2009).

[30] Butt, S., Ganapathy, V., Swift, M.M. and Chang, C.-C.: Protecting Commodity Operating System Kernels from Vulnerable Device Drivers, *Proc. 2009 Annual Computer Security Applications Conference* (online), DOI: 10.1109/ACSAC.2009.35 (2009).

[31] Tan, L., Chan, M.E., Farivar, R., Mallick, N., Carlyle, C.J., David, M.F. and Campbell, H.R.: iKernel: Isolating Buggy and Malicious Device Drivers Using Hardware Virtualization Support, *Proc. 3rd IEEE International Symposium on Dependable, Autonomic and Secure Computing* (online), DOI: 10.1109/DASC.2007.16 (2007).

[32] Sun, S. and Chiueh, T.: SIDE: Isolated and efficient execution of unmodified device drivers, *Proc. 43rd Annual IEEE / IFIP International Conference on Dependable Systems and Networks* (online), DOI: 10.1109/DSN.2013.6575348 (2013).

[33] Criswell, J., Dautenhahn, N. and Adve, V.: KCoFI: Complete Control-Flow Integrity for Commodity Operating System Kernels, *Proc. IEEE Security and Privacy*, pp.292–307 (online), DOI: 10.1109/SP.2014.26 (2014).

[34] Lu, T. and Wang, J.: Data-flow bending: On the effectiveness of data-flow integrity, *Computers & Security*, Vol.84, pp.365–375 (online), DOI: 10.1016/j.cose.2019.04.002 (2019).

[35] Davi, L., Gens, D., Liebchen, C. and Sadeghi, A.-R.: PT-Rand: Practical mitigation of data-only attacks against page tables, *Proc. 23th Network and Distributed System Security Symposium, Internet Society* (2016).

[36] Pomonis, M. and Petsios, T.: kR^X: Comprehensive kernel protection against just-in-time code reuse, *Proc. 12th European Conference on Computer Systems*, pp.420–436 (online), DOI: 10.1145/3064176.3064216 (2017).

[37] Dautenhahn, N., Kasampalis, T., Dietz, W., Criswell, J. and Adve, V.: Nested Kernel: An operating system architecture for intra-kernel privilege separation, *Proc. 20th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp.191–206 (online), DOI: 10.1145/2694344.2694386 (2015).

[38] Kurmus, A., Dechand, S. and Kapitza, R.: Quantifiable Run-Time Kernel Attack Surface Reduction, *Proc. 11th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, LNCS, Vol.8550, pp.212–234 (online), DOI: 10.1007/978-3-319-08509-8_12 (2014).

[39] Zhang, Z., Cheng, Y., Nepal, S., Liu, D., Shen, Q. and Rabhi, F.: KASR: A reliable and practical approach to attack surface reduction of commodity os kernels, *Proc. 21st International Symposium on Research in Attacks, Intrusions, and Defenses*, LNCS, Vol.11050, pp.691–710 (online), DOI: 10.1007/978-3-030-00470-5_32 (2018).

[40] Kuo, H.C., Gunasekaran, A., Jang, Y., Mohan, S., Bobba, B.R., Lie, D. and Walker, J.: MultiK: A framework for orchestrating multiple specialized kernels, Arxiv (online), available from ⟨https://arxiv.org/abs/1903.06889v1⟩ (accessed 2019-06-16).

Appendix

A.1 Types of Target Vulnerability

Table A·1 The descriptions of types of target vulnerability [14].

Types	Description
Code execution	Adversary inserts a malicious program code to the kernel, then executes it.
Memory corruption	Adversary overrides kernel data on the kernel virtual memory.
Bypass feature	Adversary circumvents kernel security restrictions.
Gain Information	Adversary accesses kernel secret information (i.e., side-channel attacks).
Gain Privileges	Adversary takes the administrator privilege on the running kernel.

Editor's Recommendation

This authors of this paper propose the kernel page restriction

© 2022 Information Processing Society of Japan

mechanism (KPRM), which is a novel security design that prohibits vulnerable kernel code execution and prevents writing to the kernel data from an adversary's user process. The proposal will be useful for other researches. The paper gives insights to readers in this research field and thus is selected as a recommended paper.

(Program Co-Chairs of IWSEC 2021, Ryo Nojima)

Hiroki Kuzuno received his M.E. degree in information science from Nara Institute of Science and Technology, Japan, in 2007, and the Ph.D. degree in computer science from Okayama University, Japan in 2020. In 2007, he has been engaged in research on computer security specifically on operating systems and networks. He is an Assistant Professor at Kobe University, Japan since 2022. He is a member of IEICE and IPSJ.

Toshihiro Yamauchi received his B.E. M.E. and Ph.D. degrees in computer science from Kyushu University, Japan in 1998, 2000, and 2002, respectively. In 2001 he was a Research Fellow of the Japan Society for the Promotion of Science. In 2002 he became a Research Associate 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. 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.

© 2022 Information Processing Society of Japan