Mitigating Foreshadow Side-channel Attack Using Dedicated Kernel Memory Mechanism

Kuzuno, Hiroki

Yamauchi, Toshihiro

(Citation)
Journal of Information Processing,30:796-806

(Issue Date)
2022-12-15

(Resource Type)
journal article

(Version)
Version of Record

Journal of Information Processing Vol.30 796–806 (Dec. 2022)

[DOI: 10.2197/ipsjjip.30.796]

Regular Paper

Mitigating Foreshadow Side-channel Attack Using Dedicated Kernel Memory Mechanism

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

Received: March 8, 2022, Accepted: September 2, 2022

Abstract: New threats to operating systems include side-channel attacks (e.g., Meltdown and Foreshadow) that combine the speculative execution of the central processing unit (CPU) and cache manipulation to facilitate inference of the kernel code and kernel data stored in CPU caches. Side-channel attacks mitigation strategies require kernel memory isolation mechanisms that modify kernel design, such as the kernel page table isolation that separates the kernel memory space for the kernel and user modes to mitigate the Meltdown, and the address space isolation that segregates the virtualization features from the kernel memory space for Foreshadow mitigation. However, user processes still share the remaining kernel feature on the same kernel memory space. The speculative execution of the CPU in a side-channel attack using Foreshadow allows the adversary to refer to the kernel data of the targeted user process with kernel features. This paper presents a dedicated kernel memory mechanism (DKMM), which controls the memory space allocation method for each user process with kernel features. It mitigates Foreshadow side-channel attack (e.g., Foreshadow-OS) with speculative execution. Furthermore, it enables each user process to use its dedicated kernel memory space and suppresses the reference to the kernel data of kernel feature used by the attacked user process attacked by Foreshadow side-channel. We implemented the DKMM on Linux and evaluated its security capability to protect the kernel data of container features against side-channel attack by the Foreshadow proof of concept code. The performance evaluation was reasonable, as the maximum system call overhead was 7.864 μs, the web client program ranged from 0.55% to 0.77% for the 100,000 Hypertext Transfer Protocol sessions, and the benchmark score was 1.06% overhead.

Keywords: side channel attack, system security, operating system, kernel

1. Introduction

A novel threat to operating systems (OS) includes an adversary's user process that refers to data from another user process through the various caches of the central processing unit (CPU) and memory management unit (MMU) by a combination of CPU speculative execution and cache operations (hereafter, "side-channel attacks") [1], [2], [3], [4], [5]. Countermeasures against side-channel attacks are urgently necessary as multiple users share hardware and kernels, which are environments where multiple computing resources are realized by a single computer using virtual machines (VM) or containers (hereafter, "multi-tenant environment").

Side-channel Attacks and Countermeasures. Meltdown enables the adversary's user process to directly refer to the kernel memory space [1]. As a countermeasure against Meltdown, kernel page table isolation (KPTI) divides the kernel memory space into the user and kernel modes [6]. Foreshadow is a form of side-channel attack utilizing the CPU speculative execution of Intel processors [3]. Foreshadow enables the adversary to refer secret information on the computer device without administrator privileges [4], [5]. Foreshadow has several variants, which are named based on the attack targets, such as the CPU security feature, OS kernel, or the VM on the multi-tenant environment (e.g., Foreshadow-SGX [7], Foreshadow-OS [8], and Foreshadow-VMM [9], [10]; the details are given in Section 2.2). More specifically, the adversary's user processes or VM can access the kernel data through the CPU L1 cache using Foreshadow which evades existing CPU, OS, or virtual machine monitor (VMM) security mechanisms [4], [5]. As a countermeasure against Foreshadow at the VMM, the VMM initializes the CPU L1 cache at the time of VM switching [11], [12], and the address space isolation (ASI), which isolates the virtualization feature from the kernel memory space [13] before applying the entire prevention mechanisms of side-channel attack. These software countermeasure approaches are important for quick mitigation. Due to the mitigation strategy of software countermeasures on the multi-tenant environment, it is necessary to support a more general design and safe running of user processes and the kernel.

Problem with Existing Software Cache Defense. All the user processes share the same kernel memory space to reduce the performance cost. ASI ensures the segregation of kernel memory for each user process of executing the VM to protect the kernel data of VM on the VMM from the adversary's user process using Foreshadow-VMM. However, because ASI only isolates the virtualization feature of the kernel, the other kernel features (i.e., container) remain on the kernel memory space. This

[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

ensure that the data of the kernel features, excluding the virtualization feature, continues to be shared among the user processes on the kernel. Therefore, the user process utilizes kernel features that have the kernel data, which cannot be protected against Foreshadow-OS side-channel attack from the adversary's user processes. It becomes necessary to provide countermeasures against the Foreshadow-OS side-channel attack to all user processes with kernel features at the kernel layer.

Research Goals. To address these problems, this paper describes the characteristics of a novel security mechanism called the dedicated kernel memory mechanism (DKMM) to allocate and control the kernel memory space for each user process with kernel features, so that the user process with kernel features can protect its kernel data against Foreshadow-OS side-channel attack (hereafter "Foreshadow side-channel attack"). The proposed mechanism is as follows:

Dedicated kernel memory mechanism. The DKMM provides a dedicated as well as a shared kernel memory space for the user processes. Additionally, the DKMM forcefully unmaps the protected kernel data from the shared kernel memory space. The kernel feature determines that the kernel data to be protected are available on the dedicated kernel memory space when the user process is executing the kernel via a system call. The DKMM design is effective for the following two reasons:

(1) The dedicated kernel memory space is the unit of CPU L1 cache sharing. The DKMM focuses on the CPU L1 cache flushing at the kernel memory switching (e.g., CR3 updating) after a privilege transition into kernel mode. It reduces the range of the CPU L1 cache sharing time during kernel execution for the user processes. The DKMM handles the kernel memory switching at the system call invocation because the Foreshadow side-channel attack is executed from the user process. After successful CPU L1 cache flushing, the kernel cannot refer to any kernel data in the CPU L1 cache except for the system call-related kernel data of the adversary's user process.

(2) The unmapping of protected kernel data from the shared kernel memory space prevents unintentional access to unmapping sensitive data and the moving of protected data into the CPU L1 cache by the adversary.

For the identification of which kernel code access which kernel data, it is difficult to make whole of kernel code control flow and data definition and use relationships. Because the kernel has many pointers (e.g., function pointer and data pointer) are modified variable of its pointer value. Additionally, the kernel has many lines of source code (e.g., Linux kernel is around 27.8 million lines [14]).

Moreover, the kernel with DKMM can handle access of protected kernel data as the page fault. The DKMM can determine whether the page fault occurred owing to malicious behavior because the Foreshadow side-channel attack requires many page faults by one user process.

The naive method covers Foreshadow-OS side-channel countermeasure in the kernel [12]. It induces the initialization of the CPU L1 cache at each user process and kernel task switching. Furthermore, it forcefully applies performance overhead because the CPU L1 cache flush command is issued for every context switching. Therefore, it is necessary to customize the countermeasure against the Foreshadow side-channel attack. DKMM provides the protection target option for the user process using the kernel features based on the administrator's configuration.

Research Contributions. We proposed the DKMM on Linux and evaluated its effectiveness. The results indicated the effectiveness of the DKMM in preventing a Foreshadow side-channel attack on user processes using the container feature, and demonstrated its reasonable performance. The novelty of the DKMM lies in that, it can support a wide range of environments because it focuses on software countermeasures at the kernel layer without additional hardware. Moreover, DKMM serves as the general mitigation approach for kernel memory space isolation. The main contributions and results obtained in this study are as follows:

(1) As a countermeasure against a Foreshadow side-channel attack in a kernel, we propose a security mechanism known as the DKMM to protect kernel data by allocating a dedicated kernel memory space to each user process with kernel features. The implementation supports container features of the user process to improve the security of kernel operations.

(2) We implemented the DKMM on Linux with KPTI and evaluated the effectiveness of kernel data protection for user processes of the container feature against a Foreshadow side-channel attack. Additionally, we evaluated the performance of our method with a Linux kernel wherein the maximum overhead was 7.864 μs for system call processing, web access overhead was between 0.55% and 0.77% for 100,000 Hypertext Transfer Protocol (HTTP) downloads, and the benchmark score was 1.06% overhead for kernel processing performance.

2. Background

2.1 Software Side-channel Attacks

Software side-channel attacks employ a combination of speculative CPU execution and cache manipulation of the CPU and MMU [1], [4], [5]. These attacks force an adversary's user process to refer the data held by other user processes or a kernel. Generally, CPU or OS security mechanisms are not allowed to access such data.

Figure 1 depicts an overview of the side channel attack requiring the sharing of CPU caches between the adversary's user process and attack target user process, on the CPU and MMU [15]. The overview before a side-channel attack involves several steps. First, the adversary's user process ensures the target cache block has been removed or fulfilled from the cache of a target region, and then it waits until the attack target user process accesses these

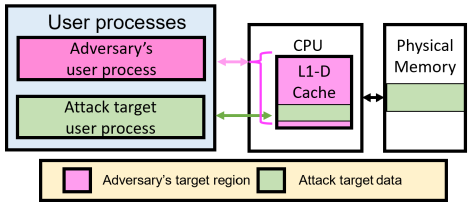

Fig. 1 The target of side-channel attacks.

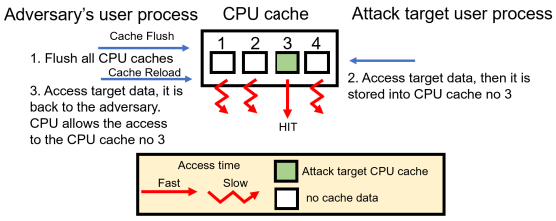

Fig. 2 Steps of FLUSH+RELOAD attack.

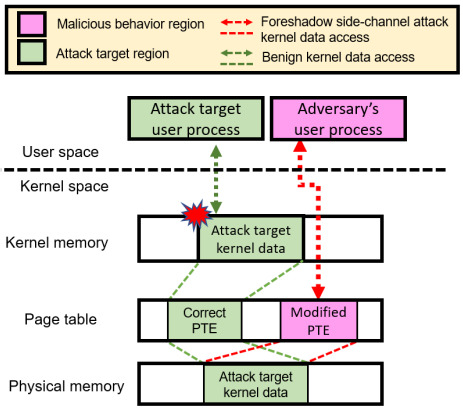

Fig. 3 Overview of Foreshadow attack.

data. Next, the adversary's user process tries to get the cache state by loading or flushing the target cache. Finally, the adversary's user process acquires the attack target data from the cache state.

2.1.1 FLUSH+RELOAD Attack

For the case of side-channel attacks using FLUSH+RELOAD, **Fig. 2** depicts the attacked data that are placed in the cache by the speculative execution after the cache is cleared, and the data that are accessed by the reference speed. The speculative execution is a function that executes software instructions ahead of time in the CPU to increase the performance. Further, when data exists in the CPU cache, the reference speed to the data increases, and the difference in the reference speed is used to determine whether the target data exist in the cache. The data in the cache can therefore be estimated using cache operation commands.

2.2 Foreshadow Attack

Foreshadow is a side-channel attack on the CPU L1 cache. It has several variants and they are named according to the attack target as follows:

(1) Foreshadow-SGX: It targets Intel SGX technology. It is formally denoted CVE-2018-3615 [7].
(2) Foreshadow-OS: It adopts unprivileged applications to access kernel memory. It is formally denoted CVE-2018-3620 [8].
(3) Foreshadow-VMM: It adopts malicious guest VMs to access memory belonging to the hypervisor and other guest machines. It is formally denoted CVE-2018-3646 [9], [10].

The proof of concept (PoC) code for Foreshadow-VMM, which allows user processes on the guest VM to attack the hypervisor [16], **Fig. 3** outlines a Foreshadow side-channel attack.

First, the adversary's user process searches a page table entry for the magic physical address (e.g., `MAGIC_PHY_ADDR` in PoC [16]) from kernel memory (e.g., `/dev/mem` of Linux kernel)

```
1. $ cat /boot/System.map-`uname -r` | grep -i cpt_data
2. ffffffff82b8c3a8  B cpt_data
3. // cat /proc/iomem at the kernel via original system call
4. $ ./read_iomem | grep 'Kernel code'
5. 01000000-01e00efc : Kernel code
6. $ cat /boot/System.map-`uname -r` | grep -i _stext
7. ffffffff81000000 T _stext

8. // cpt data physical address calculation
9. ffffffff82b8c3a8 - ffffffff81000000 = 1b8c3a8
10.1b8c3a8 + 01000000 = 2b8c3a8

11.// PoC execution before container boot
12.$ ./doit 0x2b8c3a8 0x400
13.Looking for the PTE for VA 0x7f1bfa335000 in RAM...
14.Our PTE now mapped. Value: 7badbee225
15.Dumping from VA 0x7f1bfa3353a8
16.00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

17.  // container execute
18.$ ./container cat
19.parent pid: 2459
20.child pid: 2460

21.  // container log
22.[ 151.328798] pid: monitor process 2459
23.[ 151.328799] pid: monitoring process 2460
24.[ 151.328900] cpt data 1st virtual adcress (pB):
0xffff88847646f500

25.  // PoC execution after container boot
26.$ ./doit 0x2b8c3a8 0x400
27.Looking for the PTE for VA 0x7f09e97e3000 in RAM...
28.Our PTE now mapped. Value: 7badbee225
29.Dumping from VA 0x7f09e97e3820
30.00 00 00 00 00 00 00 00 00 f5 46 76 84 88 ff ff
```

Fig. 4 Successful case of a Foreshadow side channel attack.

and overwrites the PTE value that points to the target physical address. Second, a speculative execution is performed in the CPU, if the target data are in a CPU L1 cache for the virtual address specified in the PTE. Third, the corresponding physical address value is obtained via the PTE.

2.2.1 Foreshadow Attack Case

As shown in **Fig. 4**, the modified PoC code indicates the log of a successful Foreshadow side-channel attack (e.g., Foreshadow-OS) on the kernel with the DKMM. The original PoC code [16] executes the adversary's user process on the VM to cause a page fault at the extended page table walk that targets the VMM (e.g., Foreshadow-VMM). However, the modified PoC code executes the adversary's user process in the kernel to cause the page fault at the page table walk that targets the kernel (e.g., Foreshadow-OS). More specifically, the modified PoC code was ported with its attack flows to the kernel and the DKMM disables its protection mechanism for the comparison of security capability evaluation (see Sections 6.3.2 and 6.4).

The physical address `0x2b8c3a8` in lines 1 to 9 is the virtual address where the kernel data `cpt_data` are used by the target user process to be attacked. Before executing the target user process to be attacked, we executed the modified Foreshadow PoC code as the adversary's user process to refer to the physical address of the attack target. Notably, no value could be obtained in line 16.

In line 18, we launched the target user process to be attacked. The kernel stored the virtual address value to `cpt_data` through the `kmalloc` function. From line 24, we confirmed that the value stored in `cpt_data` was `0xffff88847646f500`.

In line 26, the adversary's user process again executed the modified Foreshadow PoC code and attacked the physical address `0x2b8c3a8` of `cpt_data`. The kernel data `cpt_data` was successfully referenced in line 30.

Therefore, using Foreshadow, we were able to refer to the kernel data values. In a multi-tenant environment, the adversary's user process can refer to the values stored in the kernel data of other user processes.

3. Threat Model

The assumed environment of the threat model attempted a Foreshadow side-channel attack on the adversary's user process using a PoC code. Subsequently, the adversary referenced the kernel data on the target user process in the kernel memory space. The environment assumed for deployment was similar to a production environment, in terms of the adversary and kernel capability, and can be described as follows:

- **Adversary**: An adversary uses a normal user account and PoC code that exploits a Foreshadow side-channel attack.
- **Kernel**: A kernel does not contain any countermeasures for a Foreshadow side-channel attack, which are directly used by the PoC codes. (see Sections 2.2.1 and 6.4)
- **Attack target**: It contains the secret information of the kernel data or the parts of kernel code for another user's processes or running kernel.

3.1 Attack Scenario

In the assumed attack scenario, the approach is for the access of the secret information. The adversary's attack flow is described below.

(1) The adversary executed Foreshadow PoC code as a user process.
(2) The adversary's user process launched a Foreshadow side-channel attack that targets specified kernel data.
(3) The adversary's user process waited for the targeted user process to run.
(4) The adversary's user process continued side channeling to the specified kernel data.
(5) The adversary's user process obtained the specified kernel data to access the secret information.

3.2 Limitation of Proposed Methodology

To mitigate a Foreshadow side-channel attack from any user process, DKMM restricts all user processes on the running kernel. For reducing the performance overhead for the user process, DKMM requires the manual configuration of the administrator. The administrator manually sets the group tag (e.g., cgroups of Linux) for user processes to support a multi-tenant environment and avoid the performance overhead.

4. Design

4.1 Design Requirement

The DKMM is proposed to control the allocation of the kernel memory space for each user process with kernel features. The goal of this approach is to prevent a Foreshadow side-channel attack from referencing the kernel data to be protected in each user process with kernel features. The requirements for mitigating such attack using the DKMM can be described as follows.

- **Requirement:** A traditional kernel memory space is shared by various user processes. When an adversary's user process attempts a Foreshadow side-channel attack, it refers to arbitrary kernel data via speculative execution to bypass the CPU and kernel security mechanisms. Further protection against a Foreshadow side-channel attack requires the placement of the kernel data in a different kernel memory space from that of the adversary's user process and a suitable point of the CPU cache handling. These make it difficult for the adversary's user process to refer to the kernel data during such an attack.

4.2 The Overall Design Overview

The configuration of the DKMM before and after the introduction of the shared and dedicated kernel memory spaces were illustrated on **Fig. 5**. Conventionally, the adversary's user process shared the kernel memory space with the attack target user process, and all the kernel codes and kernel data were referenced during the speculative execution of a Foreshadow side-channel attack. After applying the DKMM, the adversary's user process and attack target user process shared only a shared kernel memory space.

The dedicated kernel memory space is the unit of CPU cache sharing. The DKMM handles the CPU cache flushing during kernel memory switching. It limits the range of the CPU cache shar-

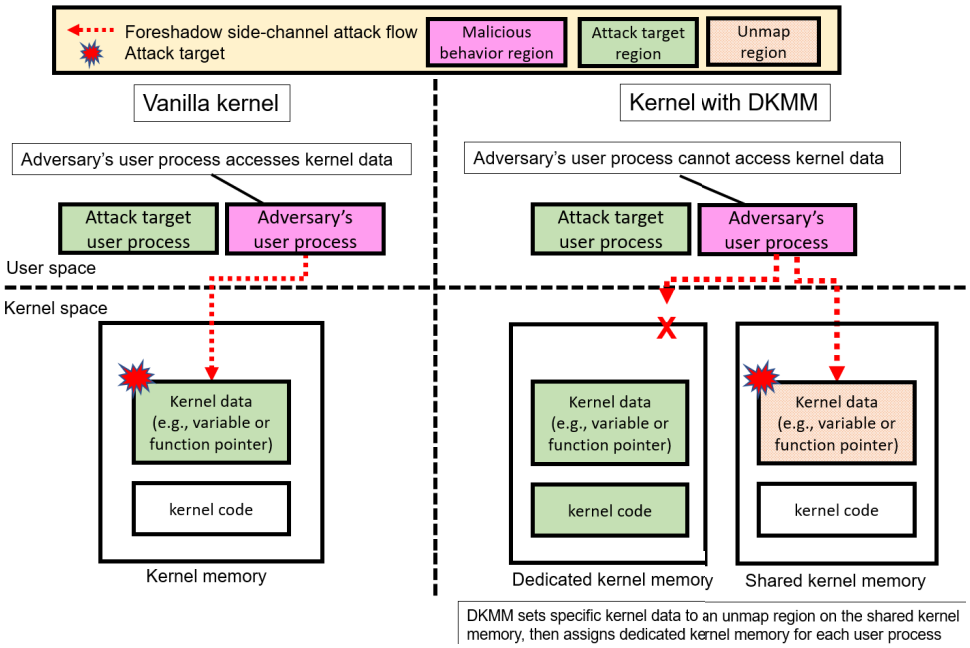

Fig. 5 Overview of the dedicated kernel memory mechanism (DKMM).

ing time during kernel execution of the adversary's user process. Additionally, the DKMM unmaps protected kernel data from the shared kernel memory space. It prevents the unintentional access of protected kernel data into the CPU cache by the adversary's user process. The adversary's user process cannot reference the kernel data located in the kernel memory space dedicated to other user processes with kernel features.

Accessing protected kernel data results in a page fault. The DKMM can determine whether page faults are caused by a malicious behavior. Because the Foreshadow side-channel attack results in numerous page faults by the adversary's user process, DKMM provides two kernel memory spaces and additional kernel data and kernel memory controlling processes of the design are follows.

- Types of kernel memory: DKMM provides a shared kernel memory space per OS and a dedicated kernel memory space per a user process.
- Protected kernel data management: DKMM supports the protected kernel data for each kernel feature.
- User process creation: DKMM requires the provision of the dedicated kernel memory space during user process creation
- User process handling: DKMM requires a kernel memory switching sequence for the accessing of protected kernel data.

4.3 Types of Kernel Memory

In the DKMM, we introduced a shared and a dedicated kernel memory space for all the user processes to meet the requirements.

- **Shared kernel memory space:** The kernel code and kernel data required for kernel operation were placed and used.
- **Dedicated kernel memory space:** The kernel data to be protected were placed in each user process with kernel features and used.

Conventionally, the adversary's user process shared the kernel memory space with the attack target user process, and all the kernel codes and kernel data were referenced during the speculative execution of Foreshadow side-channel attack.

After applying the DKMM, the adversary's user process and attack target user process shared only a shared kernel memory space. Moreover, the adversary's user process could not refer to the kernel data located in the kernel memory space dedicated to other user processes with kernel features.

4.4 Protected Kernel Data Management

DKMM requires the management of protected kernel data for each kernel feature from the adversary's user process. The design of handling timing of protected kernel data is based on the assumption that DKMM prepares the protected kernel data list at the kernel boot or user process creation for each kernel feature. Thereafter, DKMM verifies whether a protected kernel data could be unmapped from the shared kernel memory space after user process creation in the kernel layer.

4.5 User Process Creation

In this method, the dedicated kernel memory space is assigned for each user process. The flow of user process creation can be described as follows.

(1) Creation of a shared kernel memory space after the user process was created.
(2) Creation of a dedicated kernel memory space for the user process.
(3) Placing of the kernel data in a dedicated kernel memory space and its removal from the shared kernel memory space for protection.

4.6 User Process Handling

This was followed by a description of the run-time flow of the user process when using the kernel data to be protected.

(1) Usage of the shared kernel memory space when executing user processes.
(2) Switching to a dedicated kernel memory space when using protected kernel data.
(3) Running of the kernel in the dedicated kernel memory space and use of the kernel data to be protected.
(4) Execution of the kernel in the shared kernel memory space after using the kernel data to be protected.

DKMM therefore clears the CPU cache when the kernel switches between the shared and dedicated kernel memory spaces in the kernel layer.

5. Implementation

The environment assumed for the implementation target was Linux with KPTI and the CPU architecture was x86_64.

5.1 Implementation Overview

The DKMM manages three kernel page tables that control the visible kernel memory space for user processes. **Figure 6** shows the user process when the DKMM was applied to the Linux kernel leveraged KPTI, which provides the user page table and the kernel page table on its implementation. DKMM handles the kernel page table as the shared kernel page table, which was used as the kernel memory space. It was used together with the switching control for the processing of the user process. A dedicated page table was created for each user process, which was used as a dedicated kernel memory space, thereby enabling the protection of the kernel data.

To support the multi-tenant environment, DKMM prepares a `protected_kernel_data_list` and manually creates it in

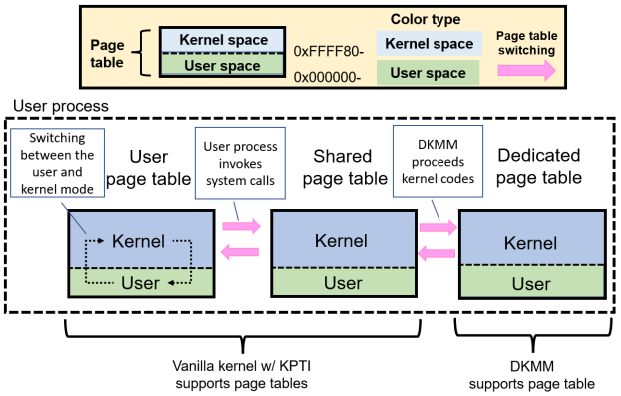

Fig. 6 Overview of user process handling.

the kernel source code. It requires manual work for usability (the details are presented in Section 7.4) It allows the Linux container feature to register its kernel data into the `protected_kernel_data_list` to unmap kernel data from the shared page table. Additionally, DKMM shares a dedicated page table among the parent and child user processes for the container environment, which is managed by the Linux cgroups.

5.2 User Process Creation

In the implementation method, the `new_pgd` variable of the `mm_struct` structure was added as a dedicated page table, and the `group_tag` as a group tag identification for the `task_struct` structure of the user process. The following steps were used to map the kernel code and kernel data.

(1) The shared page table `init_mm` variable was mapped with the kernel code and kernel data. These then run the kernel.
(2) The same kernel code and kernel data were mapped to the `new_pgd` variable from the `init_mm` variable for kernel operation.
(3) The kernel data were restored from the `protected_kernel_data_list` and subsequently unmapped from `init_mm` for the user processes.

DKMM identifies the container environment[*1] creation when the user process invokes the `clone` system call with several options (e.g., `NEWNET`, `NEWUTS`, `NEWNS`, `NEWIPC`, `NEWPID`, `NEWUSER`). Then, DKMM automatically sets the same group tag identification to the `group_tag` of the child user processes from the `group_tag` of the parent user process through the cgroups interface.

5.3 User Process Handling

This section describes the control of user process execution, which involves the following steps.

(1) Supplementation of the system call from the user process.
(2) Writing of the `new_pgd` of the `current` variable of the running user process to the CR3 register and switching it to the dedicated kernel memory space.
(3) Initialization of the CPU L1 cache using the x86_64 feature (e.g., `wrmsrl` function with `MSR_IA32_FLUSH_CMD` and `L1D_FLUSH` options).
(4) The processing of the system call is continued by the kernel.
(5) After the system call is completed, the kernel writes the variable `pgd` of the `current` to the CR3 register and switches it to the shared kernel memory space.
(6) Initialization of the CPU L1 cache using the x86_64 feature again.

To keep the kernel behavior stable, the kernel writes the `pgd` variable of the user process to the CR3 register that switches to the shared kernel memory space and continues the kernel processing. The kernel handles an interruption, exception, or context switch to another user process.

[*1] For the container environment, the cgroups manages the relation between the `UID` of user process identification and the `group_tag` of group tag identification with the read and write interfaces (i.e., `/sys/fs/cgroup/group_name/tasks` and `/sys/fs/cgroup/group_name/dkmm_tag`) of the `cftype` structure.

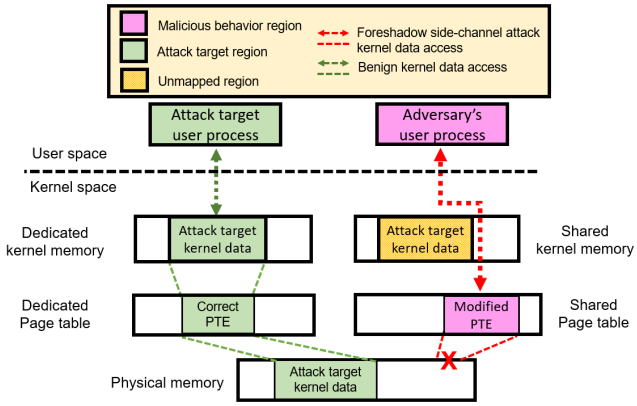

Fig. 7 Mitigation of Foreshadow attack.

5.4 Foreshadow Side-channel Attack Mitigation

DKMM reduces the possibility of sharing the kernel data among user processes in various caches of the CPU, and mitigates the kernel data referenced by a Foreshadow side-channel attack. **Figure 7** outlines a mitigation of a Foreshadow side-channel attack on the kernel with DKMM.

The attack target of kernel data was unmapped from the shared kernel memory space, even though these were located in the dedicated kernel memory space of the attack target user process. In the adversary's user process, the only scope of the shared kernel memory space and its dedicated kernel memory space that could be referenced during the speculative execution was a Foreshadow side-channel attack.

Attack target kernel data were not placed in the shared kernel memory space, because speculative execution in the CPU did not involve switching the kernel memory space. The reference process was performed only on the shared kernel memory or its dedicated kernel memory space. Additionally, the kernel with DKMM clears the CPU L1 cache when the kernel switches among kernel memory spaces. Therefore, the kernel data placed in the dedicated kernel memory space of attack target user process could not be referenced through the CPU L1 cache for a virtual address specified in the modified PTE from the adversary's user process.

6. Evaluation

6.1 Security Capability

To evaluate the security of the DKMM, we assessed the kernel data protection against a Foreshadow side-channel attack in a user process in a multi-tenant environment (e.g., container).

- **Security Capability Evaluation**
  We evaluated whether the DKMM could protect the kernel data of user processes using the container feature from a Foreshadow side-channel attack.

6.2 Performance Measurement

We measured the performance overhead cost for a vanilla kernel, the kernel with the DKMM, and the user process.

- **Measurement of the system calls invocations overhead**
  We ran the benchmark software LMbench in a container

```
1. $ cat /boot/System.map-`uname -r` | grep -i cpt_data
2. ffffffff82b8c3a8 B cpt_data01

3. // cpt data physical address calculation
4. ffffffff82b8c3a8 - ffffffff81000000 = 1b8c3a8
5. 1b8c3a8 + 01000000 = 2b8c3a8

6. // PoC execution before container boot
7. $ ./doit 0x2b8c3a8 0x400
8. Looking for the PTE for VA 0x7f78a6c0b000 in RAM...
9. Our PTE now mapped. Value: 7badbee225
10.Dumping from VA 0x7f9f94dbe800
11.00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

12. // DKMM enable, it requires root privilege
13. % echo 1 >> /sys/fs/cgroup/dkmm/dkmm_tag
14. // container execute
15.$ ./container cat
16.parent pid: 12769
17.child pid: 12770

18.// container log
19. [ 1142.604150] pid: monitor process 12769
20.[ 1142.604150] pid: monitoring process 12770
21.[ 1142.604154] cpd data 1st virtual acdress (pB):
0xffff88847646f500

22. // PoC execution after container boot
23.$ ./doit 0x2b8c3a8 0x400
24.Looking for the PTE for VA 0x7fda4026e000 in RAM...
25.Our PTE now mapped. Value: 7badbee225
26.Dumping from VA 0x7f50d1e9b3b0
27.01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Fig. 8 Prevention of Foreshadow side-channel attack.

environment with the kernel with KPTI after applying the DKMM and measured the overhead of the executing system calls.

- **Measurement of application overhead**
  We measured the performance overhead for the application in a container environment using ApacheBench.
- **Measurement of kernel processing overhead** We measured the processing performance overhead of the kernel with the DKMM using the UnixBench score.

6.3 Evaluation Environment

6.3.1 Equipment

For the evaluation, we used a computer with Intel(R) Core (TM) i7-7700HQ (2.80 GHz, four cores), 16 GB of memory, and Debian 9.0 (Linux Kernel 5.0.0, x86 64) as the OS.

6.3.2 Implementation

The implementation of the DKMM was performed in the Linux kernel 5.0.0 with KPTI, and it was realized in 41 files with 2,794 lines. The PoC code [16] was used for the Foreshadow side-channel attack, and the PTE inversion of the Linux kernel was disabled to enable the attack. To evaluate the security capability, a kernel vulnerability was introduced from the Linux kernel using parts of the PoC code [16] that lead to the Foreshadow side-channel attack via memory reading through the system call.

- Foreshadow side-channel attack: Vulnerable kernel code is ported from a part of the PoC code [16], which was implemented as a system call. The PoC code exploits the vulnerable kernel code to find a PTE for the magic physical address from `/dev/mem` and `/proc/iomem`, then returns the target PTE to the user process.

6.4 Security Capability Evaluation

For security evaluation, we placed the kernel data of a user process that used the container feature in a dedicated kernel memory space and targeted it for an attack. The attack target's user process was launched with normal user privileges, and the adversary executed the Foreshadow PoC code as the adversary's user process. The Foreshadow side-channel attack output guessed the results of the kernel data of the attack target user process.

Figure 8 depicts the kernel data protection log for the container feature for the PoC code [16] using the DKMM.

In lines 1 to 5, the physical address `0x2b8c3a8` can be identified from the virtual address where the kernel data `cpt_data` are used by the container user process. As a preliminary check, we launched side-channel attack using Foreshadow to the physical address `0x2b8c3a8` via the PoC code as the adversary's user process in line 7. Additionally, in line 11, we demonstrated that no estimated values in the CPU L1 cache existed in the form of kernel data.

We enabled the DKMM protection in line 13. We evaluated the kernel data protection for the container feature and initiated the container user process in line 15. After switching to the dedicated kernel memory space, `cpt_data`, the kernel data to be attacked, was placed, and the virtual address was allocated by the `kmalloc` function. In line 18, the kernel outputs the value of the virtual address as `0xffff88847646f500`.

In line 21, we executed the PoC code again as the adversary's user process and performed Foreshadow side-channel attacks with the physical address `0x2b8c3a8`. However, at line 25, the kernel data value is observed to be a sequence of `0`, confirming that the adversary failed to guess the value from the CPU L1 cache.

Table 1 Overhead of DKMM with Linux kernel (μs).

System call	Vanilla kernel	DKMM kernel	Total Overhead	Once Overhead
fork+/bin/sh	525.088	562.496	37.408	0.693
fork+execve	135.527	152.326	16.799	4.200
fork+exit	123.938	139.665	15.727	7.864
open/close	2.985	3.041	0.056	0.028
write	0.222	0.249	0.027	0.027
read	0.263	0.294	0.031	0.031
stat	1.008	1.051	0.043	0.043
fstat	0.285	0.311	0.026	0.026

6.5 Measurement of Performance Overhead

6.5.1 Measurement of the System Calls Invocations Overhead

LMbench was run 15 times as a user process in a container environment on the Linux kernel with KPTI before and after the application of DKMM, and the overhead was calculated from the average value.

Table 1 presents the evaluation results. LMbench called 54 times for fork+/bin/sh, four times for fork+execve, twice for fork+exit and open/close, and once for the others.

Table 1 indicates the total overhead that is entire system call invocations cost and once overhead that is once system call invocation cost for each system call. The fork+bin/sh requires 37.408 μs, which was divided by 54 times invocations yielding 0.693 μs. The fork+execve requires 16.799 μs; this was divided by four times invocations to give 4.200 μs. The fork+exit requires 15.727 μs; this was divided by two times invocations, yielding 7.864 μs. The open/close requires 0.056 μs, which was divided by two times invocations to give 0.028 μs. The other system calls have same cost for total overhead and once overhead because these are one time invocations.

Therefore, the most demanding system call was fork+exit (7.864 μs), and the least demanding was fstat (0.026 μs). The overhead per system call by the DKMM ranged from 0.026 μs to 7.864 μs.

6.5.2 Measurement of Application Overhead

The Apache 2.4.25 web server executed the web application as the user process in the container environment, and the web client was ApacheBench 2.4 for the benchmark software. The network

Table 2 ApacheBench overhead of DKMM with Linux kernel (μs).

File size (kB)	Vanilla kernel	DKMM kernel	Overhead
1	563.667	567.161	3.494 (0.62 %)
10	719.867	725.337	5.47 (0.76 %)
100	2,136.533	2,148.283	11.75 (0.55 %)

Table 3 Unixbench score of DKMM with Linux kernel.

Vanilla kernel	DKMM kernel
2,898.3	2,777.6 (1.06%)

environment was 1 Gbps.

ApacheBench 2.4 calculated the average of the HTTP download requests for the HTTP accesses. ApacheBench sent 100,000 HTTP access requests and then downloaded the file for one connection. The file sizes adopted by the benchmark configuration were 1 kB, 10 kB, and 100 kB. ApacheBench was run 15 times as a user process to calculate the average for the performance overhead cost. The list in **Table 2** indicates that our implementation had an average overhead of 0.55 % to 0.77 % for each file download access.

6.5.3 Measurement of Kernel Processing Overhead

To evaluate the kernel processing overhead, the UnixBench scores of the vanilla kernel with KPTI before and after the applying of DKMM were compared. The UnixBench measured the processing time of the system performance (i.e., calculation, file copy, and process execution) in a general environment. The UnixBench was executed five times to determine the average kernel processing time. **Table 3** presents the performance score, the implementation of our mechanism yielded a score overhead of 1.06 %.

7. Discussion

7.1 Security Capability Consideration

From the security evaluation results, we confirmed that the kernel with DKMM prevented a Foreshadow side-channel attack on the CPU L1 cache from the adversary's user process. We also confirmed that DKMM did not affect the behavior of the container functions and container user processes.

With DKMM, the kernel data of the container function did not share the CPU L1 cache with the kernel data of other kernel functions. It ensured that the kernel data of the container function was in a different kernel memory space and that the CPU L1 cache was cleared at the switching of kernel memory space; thus, it could not be guessed easily by the Foreshadow side-channel attack. Therefore, we can mitigate the threat of a Foreshadow side-channel attack by preventing the PTE used by the user process running the PoC code [16] from referring to the kernel data for each user process with a kernel feature.

7.2 Performance Measurement Consideration

We confirmed that LMbench, ApacheBench, and UnixBench correctly calculate the processing cost of the implementation of DKMM with a kernel.

LMbench measured the time required to issue a system call. The load of DKMM required the overhead of the system call from the user process in the container environment. It also required a write operation to the CR3 register due to the switching of the kernel memory space when the container function was processed as a kernel task. The high load on the fork and exit of the system calls was due to the creation and destruction of user processes in the container environment. DKMM created and released page tables for the creation of the kernel memory space, which led to an increase in the load. Additionally, the creation and release of a page also depend on the amount of protected kernel data. These processes require the sequential page handling of kernel page tables, which increases the overhead of DKMM at the user process creation (e.g., fork system call) stage.

However, ApacheBench and UnixBench indicate that DKMM has nearly the same performance result. DKMM has adopted tag-based translation lookaside buffers (TLBs). The implementation of Linux with KPTI and DKMM allocates the process-context identifier of TLB. The TLB cache improved physical memory access even though the kernel updates the CR3 register. Thus, calculation of the application performance benchmark is not significantly affected by the switching of kernel memory in the kernel mode. In addition, DKMM can be implemented for Linux regardless of KPTI. Linux with DKMM handles two page tables to support the shared and dedicated kernel memory space that requires a similar performance overhead as that of the Linux with KPTI.

7.3 Limitation

To implement DKMM, it was necessary to switch the kernel memory space, manage the kernel data, and clear the CPU L1 cache. These tasks increase the overhead for each kernel data of kernel feature to be protected.

Although the kernel memory space was allocated per user process of kernel features, it was difficult to exchange or share the kernel data to be protected. The implementation of DKMM supports the Linux cgroups that can allow the sharing of the kernel data in the dedicated kernel memory space for the user processes of the same container environment.

However, we needed to select the kernel data that is the protective object and redesign and implement the kernel components so that they worked after the kernel memory space was allocated.

Additionally, the criteria of protection data are security mechanisms related to information that contains the value of the access control policy and the function pointer of the access decision kernel code (e.g., cgroup and mandatory access control). We consider that these security mechanisms related information have to be protected from the adversary's attack. The adversary tries to collect the actual security parameter and virtual address of kernel data on the kernel memory. It prepares the attack step of forcefully overwriting kernel memory to disable the limitation of security mechanism on the multi-tenant environment.

Moreover, the level of selection difficulty of protection kernel data must be considered. It requires the combination of three factors: security, kernel stability, and performance cost. The DKMM protection mechanism unmaps the protection kernel data from the shared kernel memory space. Firstly, the user of DKMM has to determine the protection target of the kernel data on the kernel

with DKMM at the source code. Next, is the consideration of stability and performance cost. If the kernel data are referenced across the whole of the kernel, it is difficult to apply the protection mechanism of the DKMM owing to the stability effect, and kernel source code modification is required. On the other hand, the large amount of protected kernel data increases the performance cost.

Therefore, the level of selection is based on the condition of these factors from the user's kernel knowledge and environment. It indicates that the user of DKMM has to carefully determine which kernel data satisfy the conditions of the protection criteria with the kernel source code modification.

7.4 Usability

The usability of the DKMM must be considered. DKMM assumes that the `protected_kernel_data_list` is statically registered in the kernel code. The user has to manually determine which kernel data needs the protection of DKMM. The manual work includes finding the protection target of kernel data and modifying and compiling the source code. Linux kernel compiling took approximately 90 min in the evaluation environment. The source code customization relies on the user's knowledge of kernel development and deployment.

Although the dynamic registration of protected kernel data and control is important for the usability of DKMM, it might allow illegal modification of the protected kernel data. We consider that easy management is the trade-off in the design and implementation.

7.5 Portability

For the portability of DKMM to other OSs, the kernel should enable virtual memory construction and management functions. Among the existing OSs, FreeBSD and Windows provide a virtual memory, but this virtual memory still uses a single kernel memory space. As a countermeasure against Meltdown side-channel attacks, FreeBSD and Windows introduced a feature equivalent to KPTI [17], [18]. We assume that the design of DKMM can be ported to FreeBSD and Windows, that have already implemented two kernel page tables at the kernel layer. The implementation of these OSs handles the switching between the two kernel page tables. It manages the mapping of kernel code and kernel data of kernel page tables for each user process. Therefore, DKMM utilizes a similar implementation as KPTI, consisting of the page table switching and management techniques of the kernel page table, to provide and control a dedicated kernel memory space as an additional kernel page table for other OSs.

8. Related Work

Software Side-channel Attacks

Several software-based side-channel attacks have been proposed [19]. Memory operations and the response time of specific CPU instructions have also been used to infer the protected memory area [10], [15], [20]. Moreover, attacks on various CPU and MMU caches have been devised using CPU speculative execution and software implementations such as Meltdown, Spectre, and Foreshadow [1], [2], [4], [5].

Software Based Countermeasure

To counter vulnerabilities to side-channel attacks, countermeasure technologies were introduced in the CPU, kernel, and compiler. Microcode updates were performed on the CPU [21]. In the kernel, KPTI was used against Meltdown [6], which adopted a page table separation mechanism between the user and the kernel memory space [22]. For the compiler, the mitigation code [23] was introduced against Spectre to mitigate the CPU's speculative execution causing preemption.

Against Foreshadow, a user code was introduced for PTE inversion [11]. Other countermeasures against attacks between virtual machines were introduced by initializing the CPU L1 cache [12]. In ASI, the virtualization function was separated from the kernel memory space by page tables [13]. The process-local memory allocated pages so that only a specific user process could refer to a part of the kernel's virtual memory space [24].

Further, core scheduling achieved execution avoidance scheduling on the same CPU core of the user process that was being attacked or targeted [25]. Safehidden proposed the re-randomization of the kernel data placement on the kernel memory space if the translation look-aside buffer (TLB) hit was missed [26].

Hardware Based Countermeasure

To mitigate the impact of side channels, InvisiSpec examined a temporary buffer mechanism to control whether the data area could be referenced depending on the progress of the instruction when processing speculative CPU execution instructions [27]. SafeSpec proposed a control mechanism to avoid registering data in the CPU L1 cache and TLB until the end of speculative execution [28].

Cache Handling Countermeasure

As a countermeasure against side-channel attacks, a method was proposed to create a cache area that could not be referenced by the adversary's user process. In STEALTHMEM, the reference area of the last level cache (LLC) was controlled by allocating CPU cores to a virtual machine with a VMM [29]. SecDCP enabled dynamic LLC size adjustment on a per-user process basis [30]. CATalyst used Intel cache allocation technology to allocate a secure area in the LLC and allocated pages to virtual machines [31]. DAWG proposed a mechanism to link CPU cores and cache data to control accessibility [32].

9. Comparison

Table 4 presents a comparison between DKMM and previous studies related to hardware CPU cache protection features. **Table 5** presents a comparison between DKMM, ASI, and a naive method, which flushes the CPU L1 cache at the context switch of the kernel, related to software CPU cache protection features.

By reducing the possibility of sharing the CPU cache with other user processes, we mitigated the side-channel attacks on the

Table 4 Comparison of the hardware CPU cache protection features (✓ is supported).

Feature	SecDCP [30]	DAWG [32]	InvisiSpec [27]	SafeSpec [28]	DKMM
Hardware protection	✓	✓	✓	✓	
Software protection					✓
Protection granularity	CPU cache		CPU instruction		User process

Table 5 Comparison of the software CPU cache protection features (✓ is supported; △ is partially supported).

Feature	Naive method [33]	ASI [13]	DKMM
Protection configuration	△		✓
Portability		△	✓
Protection granularity	Context switch	VM	User process

CPU L1 cache by Foreshadow. DKMM can be applied to a wide range of environments because it does not require any hardware countermeasures.

CPU cache partitioning technology used hardware functions to mitigate side-channel attacks. These functions forcibly allocated the CPU cache to be used for each user process, subjecting it to side-channel attacks [30], [32]. Moreover, the CPU cache reference availability operation was based on the instruction execution order in the CPU speculative execution control. It mitigated the side-channel attack by ensuring that the inferred data were not included in the CPU cache if a CPU cache reference occurred during the side-channel attacks [27], [28]. CPU cache controlling was an important approach as a countermeasure against side-channel attacks. DKMM could be combined with existing hardware research methods.

The naive method, a side-channel countermeasure in the kernel, requires the initialization of the CPU L1 cache at each user process and kernel task switching [33]. From the viewpoint of protection configuration comparison, the naive method forcefully induces the CPU L1 cache to flush and incurs performance overhead for user processes. The naive method issues CPU hardware command (e.g., `MSR_IA32_FLUSH_CMD`) for kernel task switching at the kernel layer. It does not support the user handling of protection configuration for the user process management from the administrator. We believe that in the design of DKMM is possible to implement customizable protection of kernel memory space isolation for the user process. Although the design of DKMM issues CPU hardware command (e.g., `MSR_IA32_FLUSH_CMD`) at the switching between the user and kernel mode, the administrator can manage the protection of DKMM for the user process using `cgroups` configuration, and then apply the CPU L1 cache flush timing to the specific user process.

Moreover, the ASI separated only the kernel code and kernel data of a virtualization function (e.g., Linux KVM) from the kernel memory space [13]. The implementation of ASI focuses the Linux KVM on the x86 architecture and does not consider the portability of the ASI design and implementation for another virtualization mechanism of Linux and the CPU architecture. From the viewpoint of portability comparison, DKMM was designed to control the allocation of the kernel memory space per user process. It achieves that in a simpler design and implementation to migrate into the other OS kernel for the kernel feature protection against a Foreshadow side-channel attack.

10. Conclusion

As a countermeasure against side-channel attacks on CPU caches, the separation of the kernel memory space by KPTI against Meltdown is necessary. Furthermore, the separation of the kernel memory space of the virtualization function by ASI against Foreshadow is also necessary. However, the kernel memory space was still shared among user processes. Arbitrary kernel data could be referenced by side-channel attacks targeting the CPU cache using Foreshadow. In the kernel layer, mitigation against further side-channel attacks is important when dealing with sensitive kernel data related to user processes.

In this paper, we proposed DKMM that enabled the introduction of an additional kernel memory space for each user process as a countermeasure against Foreshadow side-channel attack. It protects the CPU cache of non-attacking user processes of the kernel memory space from a speculative execution of the CPU to access kernel data. It also prevents the kernel memory space other than the adversary's user process in the CPU cache.

We implemented DKMM in Linux and realized the allocation control of kernel data related to the container function in the kernel to a dedicated kernel memory space. To verify the effectiveness of DKMM, we confirmed that the proposed security mechanism prevented an adversary's user process from referring to the kernel data of a user process that used the container function via the CPU cache by a Foreshadow side-channel attack. The performance evaluation of the proposed security mechanism demonstrated that the maximum load per system call was 7.864 μs, the overhead to a web client program averaged between 0.55% and 0.77%, and the benchmark cost was 1.06% for the kernel.

In the future, for side-channel attacks countermeasures, researchers should evaluate the effect of countermeasures on various caches on other side-channels and the effectiveness of the performance overhead.

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

References

[1] Lipp, M., Schwarz, M., Gruss, D, Prescher, T., Haas, W., Fogh, A., Horn, J., Mangard, S., Kocher, P., Genkin, D., Yarom, Y. and Hamburg, M.,: Meltdown: Reading Kernel Memory from User Space, *Proc. 27th USENIX Security Symposium*, pp.973–990 (online), DOI: 10.5555/3277203.3277276 (2018).

[2] Kocher, P., Horn, J., Fogh, A., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M. and Yaram, Y.: Spectre Attacks: Exploiting Speculative Execution, *Proc. 2019 IEEE Symposium on Security and Privacy*, pp.1–19 (online), DOI: 10.1145/3399742 (2019).

[3] Foreshadow: Breaking the Virtual Memory Abstraction with Transient Out-of-Order Execution (online), available from ⟨https://foreshadowattack.eu/⟩ (accessed 2022-03-08).

[4] Bulck, V.J., Minkin, M., Veisse, O., Genkin, D., Kasikci, B., Piessens, F., Silberstein, M., Wenisch, F.T., Yarom, Y. and Strackx, R.: FORESHADOW: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution, *Proc. 27th USENIX Security Symposium*, pp.991–1008 (online), DOI: 10.5555/3277203.3277277 (2018).

[5] Weisse, O., Bulck, V.J., Minkin, M., Genkin, D., Kasikci, B., Piessens, F., Silberstein, M., Strackx, R., Wenisch, F.T. and Yaram, Y.: Foreshadow-NG: Breaking the Virtual Memory Abstraction with TransientOut-of-Order Execution (online), available from ⟨https://foreshadowattack.eu/⟩ (accessed 2020-08-05).

[6] 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).
[7] CVE-2018-3615: MITRE (online), available from ⟨https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-3615⟩ (accessed 2020-07-31).
[8] CVE-2018-3620: MITRE (online), available from ⟨https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-3620⟩ (accessed 2020-07-31).
[9] CVE-2018-3646: MITRE (online), available from ⟨https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-3646⟩ (accessed 2020-07-31).
[10] Schwarz, M.: Software-based Side-Channel Attacks and Defenses in Restricted Environments, *PhD Thesis, Graz University of Technology* (2019).
[11] Corbet, J.: Meltdown strikes back: The L1 terminal fault vulnerability, LWN.net (online), available from ⟨https://lwn.net/Articles/762570/⟩ (accessed 2020-08-06).
[12] The Linux kernel user's and administrator's guide: L1TF - L1 Terminal Fault (online), available from ⟨https://www.kernel.org/doc/html/latest/admin-guide/hw-vuln/l1tf.html⟩ (accessed 2020-08-06).
[13] Chartre, A.: Kernel address space isolation, LWN.net (online), available from ⟨https://lwn.net/Articles/813393/⟩ (accessed 2020-06-12).
[14] Larabel, M.: The Linux Kernel Enters 2020 At 27.8 Million Lines In Git But With Less Developers For 2019 (online), available from ⟨https://www.phoronix.com/scan.php?page=news_item&px=Linux-Git-Stats-EOY2019⟩ (accessed 2020-08-05).
[15] Yarom, Y. and Falkner, K.: FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack, *Proc. 23rd USENIX Security Symposium*, pp.719–732 (online), DOI: 10.5555/2671225.2671271 (2014).
[16] gregvish: L1TF (Foreshadow) VM guest to host memory read PoC, Github (online), available from ⟨https://github.com/gregvish/l1tf-poc⟩ (accessed 2020-07-31).
[17] Tetlow, G.: Response to Meltdown and Spectre, FreeBSD Foundation (online), available from ⟨https://lists.freebsd.org/pipermail/freebsd-security/2018-January/009719.html⟩ (accessed 2020-05-21).
[18] Ionescu, A.: Windows 17035 Kernel ASLR/VA Isolation In Practice (like Linux KAISER) (online), available from ⟨https://twitter.com/aionescu/status/930412525111296000⟩ (accessed 2020-08-06).
[19] Canella, C., Bulck, V, J., Schwarz, M., Lipp, M., Berg, V, Benjamin, Ortnet, P., Piessens, F., Evtyushkin, D. and Gruss, D.: A Systematic Evaluation of Transient Execution Attacks and Defenses, *Proc. 28th USENIX Security Symposium*, pp.249–266 (online), DOI: 10.5555/3361338.3361356 (2019).
[20] Jang, Y., Lee, S. and Kim, T.: Breaking Kernel Address Space Layout Randomization with Intel TSX, *Proc. 2016 ACM SIGSAC Conference on Computer and Communications Security*, pp.380–392 (online), DOI: 10.1145/2976749.2978321 (2016).
[21] Addressing Hardware Vulnerabilities, Intel Corporation (online), available from ⟨https://www.intel.com/content/www/us/en/architecture-and-technology/facts-about-side-channel-analysis-and-intel-products.html⟩ (accessed 2020-08-06).
[22] Hanse, D.: KAISER: unmap most of the kernel from userspace page tables, LWN.net (online), available from ⟨https://lwn.net/Articles/738997/⟩ (accessed 2020-08-06).
[23] LLVM.: Introduce the "retpoline" x86 mitigation technique (online), available from ⟨https://reviews.llvm.org/D41723⟩ (accessed 2020-08-06).
[24] Hillenbrand, M.: Process-local memory allocations for hiding KVM secrets, LWN.net (online), available from ⟨https://lwn.net/Articles/791069/⟩ (accessed 2019-08-08).
[25] Zijlstra, P.: Core scheduling, LWN.net (online), available from ⟨https://lwn.net/Articles/780703/⟩ (accessed 2020-06-12).
[26] Wang, Z., Wu, C., Zhang, Y., Tang, B., Yew P.-C., Xie, M., Lai, Y., Kang, Y., Cheng, Y. and Shi, Z.: Safehidden: An efficient and secure information hiding technique using re-randomization, *Proc. 28th USENIX Security Symposium*, pp.1239–1256 (online), DOI: 10.5555/3361338.3361424 (2019).
[27] Yan, M., Choi, J., Skarlatos, D., Morrison, A., Fletcher, W.C. and Torrellas, J.: Invisispec: Making speculative execution invisible in the cache hierarchy, *Proc. 51st Annual IEEE/ACM International Symposium on Microarchitecture*, pp.428–441 (online), DOI: 10.1109/MICRO.2018.00042 (2018).
[28] Khasawneh, N.K., Koruyeh, M.E., Song, C., Evtyushkin, D., Ponomarev, D. and Abu-Ghazaleh, N.: SafeSpec: Banishing the Spectre of a Meltdown with Leakage-Free Speculation, *Proc. 56th ACM/IEEE Design Automation Conference*, pp.1–6 (online), DOI: 10.1145/3316781.3317903 (2019).
[29] Kim, T., Peinado, M. and Mainar-Ruiz, G.: STEALTHMEM: System-Level Protection Against Cache-Based Side Channel Attacks in the Cloud, *Proc. 21st USENIX Security Symposium*, pp.189–204 (online), DOI: 10.5555/2362793.2362804 (2012).
[30] Wang, Y., Ferraiuolo, A., Zhang, D., Myers, C.A. and Suh, E.G.: SecDCP: Secure dynamic cache partitioning for efficient timing channel protection, *Proc. 53rd ACM/EDAC/IEEE Design Automation Conference*, pp.1–6 (online), DOI: 10.1145/2897937.2898086 (2016).
[31] Liu, F., Ge, Q., Yaram, Y., Mckeen, F., Rozas, C., Heiser, G. and Lee, B.R.: CATalyst: Defeating last-level cache side channel attacks in cloud computing, *Proc. IEEE International Symposium on High Performance Computer Architecture*, pp.406–418 (online), DOI: 10.1109/HPCA.2016.7446082 (2016).
[32] Kiriansky, V., Lebedev, L., Amarasinghe, S., Devadas, S. and Emer, J.: DAWG: A Defense Against Cache Timing Attacks in Speculative Execution Processors, *Proc. 51st Annual IEEE/ACM International Symposium on Microarchitecture*, pp.974–987 (online), DOI: 10.1109/MICRO.2018.00083 (2018).
[33] Singh, B.: [RFC PATCH] arch/x86: Optionally flush L1D on context switch (online), available from ⟨https://lore.kernel.org/lkml/20200313220415.856-1-sblbir@amazon.com/⟩ (accessed 2022-03-07).

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.