Efficient Protection Mechanism for CPU Cache Flush Instruction Based Attacks

Enomoto, Shuhei
Kuzuno, Hiroki
Yamada, Hiroshi

(Citation)
IEICE Transactions on Information and Systems,E105.D(11):1890-1899

(Issue Date)
2022-11-01

(Resource Type)
journal article

(Version)
Version of Record

(URL)
https://hdl.handle.net/20.500.14094/0100483236

IEICE TRANS. INF. & SYST., VOL.E105–D, NO.11 NOVEMBER 2022

PAPER Special Section on Next-generation Security Applications and Practice

Efficient Protection Mechanism for CPU Cache Flush Instruction Based Attacks

Shuhei ENOMOTO[†a)], ***Nonmember***, **Hiroki KUZUNO**[††b)], ***and*** **Hiroshi YAMADA**[†c)], ***Members***

SUMMARY CPU flush instruction-based cache side-channel attacks (cache instruction attacks) target a wide range of machines. For instance, Meltdown / Spectre combined with FLUSH+RELOAD gain read access to arbitrary data in operating system kernel and user processes, which work on cloud virtual machines, laptops, desktops, and mobile devices. Additionally, fault injection attacks use a CPU cache. For instance, Rowhammer, is a cache instruction attack that attempts to obtain write access to arbitrary data in physical memory, and affects machines that have DDR3. To protect against existing cache instruction attacks, various existing mechanisms have been proposed to modify hardware and software aspects; however, when latest cache instruction attacks are disclosed, these mechanisms cannot prevent these. Moreover, additional countermeasure requires long time for the designing and developing process. This paper proposes a novel mechanism termed FlushBlocker to protect against all types of cache instruction attacks and mitigate against cache instruction attacks employ latest side-channel vulnerability until the releasing of additional countermeasures. FlushBlocker employs an approach that restricts the issuing of cache flush instructions and the attacks that lead to failure by limiting control of the CPU cache. To demonstrate the effectiveness of this study, FlushBlocker was implemented in the latest Linux kernel, and its security and performance were evaluated. Results show that FlushBlocker successfully prevents existing cache instruction attacks (e.g., Meltdown, Spectre, and Rowhammer), the performance overhead was zero, and it was transparent in real-world applications.

key words: *side-channel attack, operating system, security*

1. Introduction

Several side-channel attacks related to CPU cache and cache flush instructions (cache instruction attacks) have been reported over the past decade [1]–[6]. These attacks are a serious problem as they affect a wide range of machines. For example, FLUSH+RELOAD [1], [2] is a cache instruction attack that leaks credentials from the victim's user process to an adversary's user process in a page sharing environment. Researchers have reported that FLUSH+RELOAD works on Intel x86 and ARM CPUs. In x86 based FLUSH+RELOAD, the attack attempts to store and extract credentials with last-level cache (LLC) by misemploying cache flush instructions such as clflush.

Meltdown [4] and Spectre [5] have gained attention as serious side-channel attacks in x86 CPUs. These attacks fall under the category of cache instruction attacks as they combine FLUSH+RELOAD in the attack phase. Meltdown and Spectre attempt to obtain read access to arbitrary data in physical memory; they can escape memory protection by operating system (OS) kernels and leak credentials from kernel and user processes (e.g., kernel pointer).

Moreover, fault injection attacks also include cache instruction attacks, for example, Rowhammer [7]. Rowhammer tries to flip bits by gaining write to access to the target memory row of DRAM with a high frequency. Therefore, Rowhammer needs to flush the CPU cache with specific instructions such as clflush in x86. With Rowhammer, an adversary can write arbitrary data in physical memory, which leads to actual attacks (e.g., privilege escalation) [6].

To prevent cache instruction attacks, several security mechanisms have been proposed involving hardware and software modifications; an example of this is the introduction of a fixed CPU, by Intel (e.g., Cascade Lake microarchitecture [8]) to prevent Meltdown. To protect against Rowhammer, the introduction of DDR4 memory has a mitigation effect. In software based protection, kernel page table isolation (KPTI) [9] is effective for protecting against Meltdown and Retpoline [10] has a mitigation outcome for Spectre. Although these security mechanisms have good effect for mitigating to existing cache based side-channels such as Meltdown, Spectre and Rowhammer, these may not be able to prevent cache instruction attacks when the latest side-channel vulnerability is disclosed. To cover the latest side-channel vulnerability, researchers and developers need to design and implement an additional countermeasure mechanism that requires long time deployment process. For example, Cascade Lake was needed 454 days from announcement Meltdown to releasing the architecture, and KAISER [11] that is the original design of KPTI was required 46 days until merging to mainline Linux kernel [12], [13].

This study proposes a novel software based approach named FlushBlocker that provides prevention of all types of cache instruction attacks, and mitigation of these with latest side-channel vulnerability while additional countermeasure is developed. FlushBlocker achieves low overhead and high deployability, by primarily restricting the cache flush instructions. The implementation of FlushBlocker employs at the start of the user process. It scans the executable pages and registers the instructions into the hardware debug registers; and in case of trapped instructions, it skips them. Therefore, FlushBlocker only requires an additional minor

Manuscript received February 10, 2022.
Manuscript revised May 13, 2022.
Manuscript publicized July 19, 2022.

[†]The authors are with Tokyo University of Agriculture and Technology, Koganei-shi, 184–8588 Japan.
[††]The author is with Kobe University, Kobe-shi, 657–8501 Japan.
a) E-mail: enomoto@asg.cs.tuat.ac.jp
b) E-mail: kuzuno@port.kobe-u.ac.jp
c) E-mail: hiroshiy@cc.tuat.ac.jp

DOI: 10.1587/transele.2022NGP0008

cost for creating a user process phase, and an administrator can easily install FlushBlocker in various environments such as cloud virtual machines, desktops, and laptops.

FlushBlocker must address two challenges to fulfill its purpose. First, it must prevent attacks that bypass the security mechanism that it has established. To protect against bypassing attacks, this paper introduces four defeating methods by an attacker and designs countermeasures. FlushBlocker scans the pages while creating a new executable memory map (e.g., mmap syscall) or changing the existing memory map (e.g., mprotect syscall). As memory mapping operations related to executable locations occur only during the initializing phase of the user process, performance overhead by installing FlushBlocker is still low for non-malicious user processes. Second, it needs to minimize the impact on the semantics of non-malicious applications. To minimize the impact of semantics, FlushBlocker provides two policies to determine whether to skip instructions based on two policies. The administrator (e.g., the root user) can configure these policies for reduction of performance overhead and adjusting the effect of malicious and non-malicious applications.

In short, the contributions of this paper are as follows:

1. The proposed mechanism called FlushBlocker, which is a novel software based mitigation mechanism, for the existing and undisclosed cache instruction attacks works with low overhead and can be deployed to machines in various environments. Additionally, FlushBlocker is designed as an in-kernel component and is implemented in the latest Linux kernel. To prevent the cache instruction attacks, it scans executable pages in every user process to transparently restrict cache flush instructions.
2. To prevent against defeating of FlushBlocker's mechanism by an attacker, FlushBlocker resists bypassing scanning and trapping. Additionally, FlushBlocker provides two user-configurable policies to minimize the impact on semantics of programs.
3. To demonstrate, FlushBlocker when evaluated in security and performance experiments, the security evaluation showed that FlushBlocker successfully prevented the Meltdown, Spectre, and Rowhammer Proof-of-Concept (PoC). The performance evaluation indicated 31.73% overhead on UnixBench and no overhead on SPEC CPU 2017 benchmark suite and real-world server applications.

Finally, this study is an extension of the previous study with further details and evaluations [14]. The differences are identified and listed below. First, this latest study investigates the limitations of FlushBlocker through design and implementation and in comparison to previous works. Second, the consideration of the feasibility evaluation was carried out using benchmark software. Additionally, this study elucidates the background and establishes a threat model for FlushBlocker. Hence, the results reinforce FlushBlocker as a viable countermeasure against cache flush instruction attacks.

2. Background

2.1 FLUSH+RELOAD Attack

FLUSH+RELOAD [1], [2] is a side-channel attack that tracks the execution of a victim from an adversary. The attack works on an environment in which pages are shared between the adversary's user process and victim's user processes in a single OS, or the adversary's virtual machine and the victim's virtual machine in a single hypervisor.

Figure 1 (A) shows the attack of FLUSH+RELOAD. With this assumption, an adversary and a victim contain the same code contents that are shared by a page-sharing mechanism such as kernel same page merging [15]. First, the adversary evicts the code contents from all levels of the CPU cache by issuing cache flush instructions (e.g., clflush and clflushopt). Next, they wait for a certain duration to assess the time required to access the code contents. Meanwhile, they can determine if the cached code contents are executed by the victim in the waiting time. Therefore, they are available to track the victim's execution by repeating this operation. Yarom et al. showed that FLUSH+RELOAD extracts the RSA private key from GnuPG [16], and tracking the victim's execution leads to leakage of secret data.

2.2 Meltdown/Spectre Attack

Meltdown [4] and Spectre [5] are vulnerabilities in the microarchitectures of Intel x86 and ARM, where an adversary can leak arbitrary data from physical memory with side-channel attacks that use these vulnerabilities. For example, a kernel pointer that leads to a break in KASLR [17], [18],

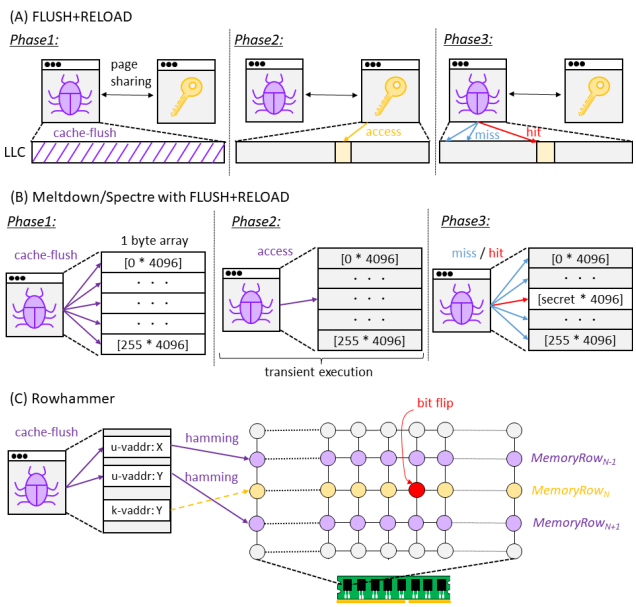

Fig. 1 FLUSH+RELOAD, Meltdown/Spectre and Rowhammer overview

and a cryptographic key, in the user process, are targeted by the attacks. In the side-channels, the attacks need to store data, leaked to the LLC, inside transient execution. Therefore, the attacks employ cache based side-channel attacks such as FLUSH+RELOAD [1], [2].

Figure 1 (B) shows the attack of the transient execution vulnerability combined with FLUSH+RELOAD for leaking one byte of secret data. First, an adversary starts a malicious user process that contains the code of the side-channels. Next, the user process creates an array of one byte from *array[0*PAGE_SIZE]* to *array[255*PAGE_SIZE]*. After creating the array, adversary issues cache flush instructions (e.g., clflush and clflushopt) to clear all the cached elements from all levels of the CPU cache. Second, the user process obtains the confidential data by transient execution vulnerability and accesses to *array[secret*PAGE_SIZE]* inside the execution. Finally, the user process measures the access time from *array[0*PAGE_SIZE]* to the *array[255*PAGE_SIZE]*. In this measurement, *array[secret*PAGE_SIZE]* is the shortest access time because only that element has been cached. Therefore, they can store the confidential data after transient execution.

Although, the transient execution vulnerable side-channels can combine with other cache based attacks (e.g., EVICT+RELOAD and PRIME+PROBE [19]) to obtain secret data at low noise, many of the real-world malware that contain side-channel code use FLUSH+RELOAD. In fact, 10 out of the total 12 wild Meltdown/Spectre based malware [20][†] contained the behavior of FLUSH+RELOAD.

2.3 Rowhammer Attack

Rowhammer [7] is a fault injection attack that targets DRAM. This attack attempts to flip bits stored in the targeted memory row by accessing memory rows around the targeted memory row with a high frequency. The flipping action gains write access to the OS kernel data structure, which leads to more serious attacks such as privilege escalation [6].

Figure 1 (C) shows the attack on Rowhammer. First, an adversary starts a malicious user process that contains the Rowhammer code. The user process determines the virtual address that stores the target data and identifies a target memory row ($MemoryRow_N$) translated from the virtual address. At that time, the user process also identifies the virtual addresses related to memory rows around $MemoryRow_N$ ($MemoryRow_{N-1}$, $MemoryRow_{N+1}$). Next, the user process accesses the virtual addresses of $MemoryRow_{N-1}$ and $MemoryRow_{N+1}$.

To access physical memory at a high frequency, Rowhammer must flush the CPU cache. Therefore, several existing Rowhammer PoC code issue CPU cache flush instructions. For example, in the samples of Rowhammer's code disclosed as open-sourced projects, seven of the total ten samples contained a behavior that flushed the CPU cache with a clflush instruction[††].

3. Threat Model

The cache instruction attack perceives the environment of the threat model of FlushBlocker that postulates the capability of the adversary and its side-channel attack methods during an attack scenario.

Permission: An adversary assumes normal user privileges and executes the shell command with the PoC code in the environment. An adversary can access the file system, networking, and control processes as a non-root user. However, they cannot insert malicious kernel modules into the OS kernel.

Attack Method: This is a known attack method of an adversary. The adversary attacks a system with cache instruction attacks, such as FLUSH+RELOAD based Meltdown / Spectre, clflush based Rowhammer and undisclosed cache side-channels using clflush. Such attacks are adversary attempts to obtain read and write access to arbitrary data.

3.1 Attack Scenario

The attack scenario is an approach used to induce information leakage. First, the adversary executes the PoC code as the user process starts the cache instruction attack. Subsequently, the adversary's user process can access any cache data of the CPU or memory data. Therefore, the adversary's user process receives a cache flush instruction attack to bypass the privilege restrictions of the OS kernel and CPU. Finally, the adversary's user process takes kernel data from the OS kernel or privilege escalation from the attack environment.

4. Proposed Approach

4.1 Requirements of FlushBlocker

This study proposes FlushBlocker with a low overhead for kernel processing to meet the following requirements of mitigating cache instruction attacks:

1. Prevent cache instruction attacks while retaining a low overhead.
2. Support a wide range of architectures and kernel designs.

To satisfy the first requirement, FlushBlocker targets the kernel component design to reduce the negative effects of mitigating the processes for kernel processing. It then ensures that user applications incur running costs in the native environment.

To satisfy the second requirement, FlushBlocker does

[†]This analysis targeted downloadable samples of malware from *hybrid-analysis.com* (accessed on January 1st, 2022).

[††]This survey targeted public repositories in *github.com* (accessed on January 1st, 2022).

not rely on hardware features to cover multiple architectures and kernel designs for easy porting of the mitigation process.

4.2 Approach of FlushBlocker

FlushBlocker restricts specific cache flush instructions during the attacking phase to prevent instruction cache attacks. When using FlushBlocker, the malicious user process cannot obtain secret data from the CPU cache, as it is difficult to identify the byte of data that matches the CPU cache during the cache access calculation. The malicious user process cannot repeatedly access the memory row using cache flush instructions. Therefore, an adversary cannot easily lead to the cache instruction attacks.

5. Design

5.1 Monitoring of User Process

FlushBlocker uses hardware breakpoint of debug registers from the CPU architecture to trap specific instructions of the user process. Additionally, FlushBlocker scans the program of the user process and then stores the position of the targeted instructions into the debug registers for dynamic trapping.

Figure 2 illustrates the handling flow process of FlushBlocker, which is detailed as follows.

1. The adversary executes a malicious user program that initiates a side-channel attack.
2. The kernel loads the malicious user process to memory through the program execution sequence.
3. FlushBlocker scans all pages of the malicious program code during the system call invocation. The scan attempts to detect the cache flush instructions on each page.
4. FlushBlocker marks the virtual addresses of the cache flush instructions if the pages contain such cache flush instructions, and subsequently, identifies the malicious user process as a monitoring target.

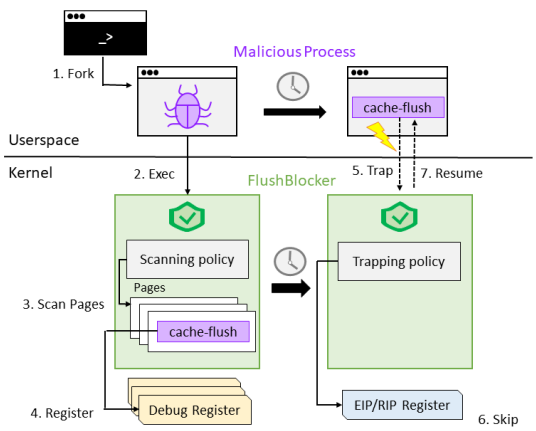

Fig. 2 FlushBlocker design overview

5. The malicious user process initiates a cache instruction attack.
6. FlushBlocker traps the execution of the cache flush instructions at the kernel and then determines whether the cache flush instructions belong to the target user process. If the target user process executes the targeted cache flush instructions, then FlushBlocker increments the program counter of the target user process.
7. The malicious user process continues to execute the program following the cache flush instructions.

FlushBlocker scans pages of the user process, making them transparent to prevent cache flush instructions for each user process.

5.2 Defeating Methods of FlushBlocker

Several defeating methods can attempt to avoid the mitigation approach of FlushBlocker by forcibly issuing cache flush instructions from malicious user processes within an existing kernel.

Defeating Method 1 (DM1): A malicious user process creates a user process or thread that issues cache flush instructions.

Defeating Method 2 (DM2): While running, a malicious user process creates an additional page that contains cache flush instructions and then issues these instructions.

Defeating Method 3 (DM3): A malicious user process creates an additional page containing cache flush instructions without an execution flag. It then enables the execution flag of the page through an `mprotect` system call.

Defeating Method 4 (DM4): A malicious user process contains more cache flush instructions than the number of hardware debug registers in a CPU's architecture.

5.3 Countermeasures for the Defeating Methods

FlushBlocker provides countermeasures for DMs (Fig. 3) are detailed below:

Countermeasure for DM1: DM1 uses an existing kernel implementation that does not set a debug register for the child user process or thread from the debug register setting of the parent user process. DM1 can issue cache instructions in the child user process or thread to avoid FlushBlocker tracking. FlushBlocker copies the debug register information to the child user process or thread from the parent user process.

Countermeasure for DM2: DM2 attempts to create an additional page with an execution flag and the cache flush instruction. It avoids FlushBlocker's page-scanning phase during the `exec` system call timing. FlushBlocker adopts the following countermeasures to trap the cache flush instructions of an additional page:

1. A malicious user process creates an additional page with an execution flag through the memory allocation sequence.

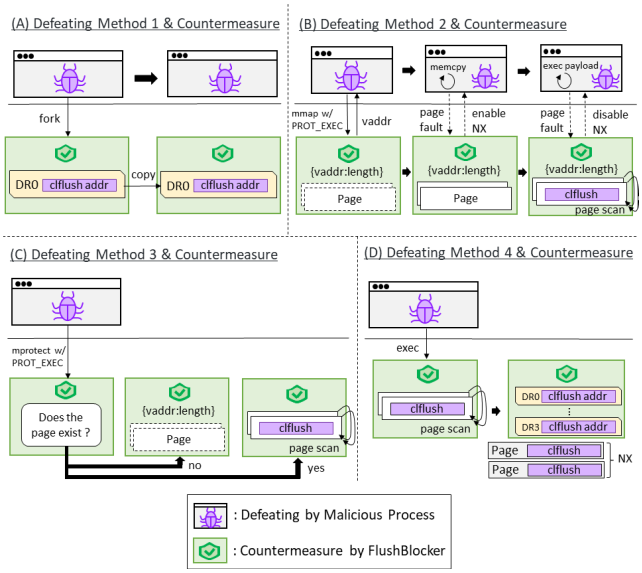

Fig. 3 Countermeasures against defeating FlushBlocker

2. FlushBlocker stores the starting and ending virtual addresses of a page during the memory allocation sequence.
3. A malicious user process copies the program code payload containing the cache flush instruction.
4. FlushBlocker triggers page writing and demands paging of page faults at the kernel to determine whether the virtual address of a page fault matches the targeted page for dropping the execution flag.
5. A malicious user process executes cache flush instructions on the additional page.
6. FlushBlocker catches a page fault with an execution flag exception, and determines the virtual address of the page fault contained in the target pages. FlushBlocker identifies previously targeted pages and enables the execution flag.

Countermeasure for DM3: DM3 adopts the execution flag to avoid FlushBlocker's page scan during the execution and additional page allocation. First, the malicious user process creates an additional page without an execution flag. Subsequently, it writes cache flush instructions to the additional page; finally, it enables the execution flag.

FlushBlocker's countermeasure against DM3 is to hook the internal process of the execution flag control to identify the cache flush instruction of the target page scanning before executing the flag available on the kernel.

Countermeasure for DM4: DM4 uses the drawback of the limitation of the hardware debug register feature. The malicious user process contains more cache flush instructions than the CPU hardware debug register. FlushBlocker cannot register all the cache flush instructions to the hardware debug register; therefore, the malicious user process issues the cache flush instruction without being trapped by FlushBlocker.

FlushBlocker counters this limitation by controlling the execution flag of those pages. Moreover, it only enables the execution flag of the page containing cache flush instructions stored in one of the hardware debug registers. After trapping the cache flush instruction, FlushBlocker exchanges the virtual address on the hardware debug register from the trapped cache flush instruction with other flush instructions. In addition, FlushBlocker enables the execution flag of pages that only store virtual addresses on the hardware debug registers.

5.4 Setting for Trusted Users

FlushBlocker's objective is to restrict the issue of cache flush instructions by a malicious user process of an adversary. Based on this, FlushBlocker provides a mechanism to allow the issue of the instructions by non-malicious user processes of trusted users (e.g., the root user). In this mechanism, the root user can set the UID of a trusted user for FlushBlocker. When the UID of a user process matches that of a trusted user, FlushBlocker does not scan the code pages of this user process.

5.5 Setting for Tolerance of Cache Flush

FlushBlocker skips cache flush instructions in the default setting when the instructions are trapped. However, skipping the instructions may compromise the semantics of the program. To minimize the impact on the semantics, FlushBlocker provides a policy for the treatment of the trapped instructions to the root user. In the current design, the root user can set a limit of tolerance of cache flush instructions per one second of one process. Based on this setting, FlushBlocker determines whether or not to skip the instructions.

6. Implementation

FlushBlocker was implemented on a Linux kernel with an x86_64 architecture.

6.1 Page Scanning

Page scanning of the user process is performed at the `exec` system call invocation. FlushBlocker reads every code page of the user process and searches for binary patterns of the cache flush instructions. In the x86_64 architecture, the user process can only issue clflush and clflushopt for cache flush of non-privileged instructions. Therefore, the prototype targets these two instructions.

6.2 Debug Register

FlushBlocker stores the virtual addresses of the cache flush instructions to debug registers. FlushBlocker requires the monitoring flag `enable_dr` variable for the `task_struct` structure and enables the monitoring flag when the user process is the monitoring target. The x86_64 architecture has eight debug registers (i.e., `DR0`–`DR7`) that require privileged

instructions to enable control. Four debug registers (i.e., DR0–DR3) are available for trapping of the virtual addresses. FlushBlocker ensures that the user process with normal privilege cannot misuse the debug registers and the ptrace system call that requires root privilege.

6.3 Countermeasures for the DMs

FlushBlocker adopts the countermeasures for DMs for Linux kernel implementation with the x86_64 architecture.

Countermeasure for DM1: To handle the continuous monitoring of malicious user process trees, FlushBlocker duplicates the debug register information to the child process from the parent process in the clone system call.

Countermeasure for DM2: To trap the cache flush instructions on an additional page while the user process runs, FlushBlocker forcibly disables the execution privilege of the VM_EXEC flag for user processes when the additional page requires the PROT_EXEC flag during the mmap system call invocation. Therefore, FlushBlocker traps the cache flush instructions of an additional page using page fault mechanisms. If an additional page contains cache flush instructions, then FlushBlocker sets the debug register of the virtual address and enables VM_EXEC.

Countermeasure for DM3: To handle the PROT_EXEC flag enabling a non-executable page that contains a program payload with the cache flush instruction, FlushBlocker hooks the mprotect system call invocation with PROT_EXEC to scan the page and check if it contains cache flush instructions. Next, it registers the additional cache flush instructions to handle trapping with the hardware debug register.

Countermeasure for DM4: FlushBlocker manages hardware debug registers to track more than five cache flush instructions on the x86_64 architecture. (e.g., x86_64 has four debug registers for trapping). FlushBlocker forcibly sets a non-executable (NX) bit for the remaining pages that are not registered to the hardware debug registers (i.e., DR0–DR3). Additionally, FlushBlocker sorts the debug register entries using the least recently used algorithm when a page fault occurs and exchanges addresses on the hardware debug registers with NX bit handling.

6.4 Scanning and Trapping Policies

FlushBlocker provides configurable policies called scanning policy and trapping policy to handle behavior when the cache flush instructions are scanned, and trapped.

Scanning Policy: an administrator of the OS can set the UID (e.g., 0 in root user) to the policy. When a user matched to the UID creates a process, FlushBlocker avoids scanning pages of the process. With the policy, user processes created by trusted users can avoid scanning and issue cache flush instructions.

Trapping Policy: an administrator can set a number for the tolerance of cache flush instructions to the policy. When hardware debug interruptions occur, FlushBlocker identifies if the user process is being monitored through a monitoring flag variable. Based on the trapping policy, with the case of exceeding tolerance, FlushBlocker skips the instructions to increase the IP register for the malicious user process.

In the current implementation, FlushBlocker uses the procfs interface to obtain the setting. To avoid access to the setting by a malicious user, only the root user has read and write access to the file.

7. Evaluation

7.1 Security Capability Experiment

The validation of the security capability of FlushBlocker by the prevention of cache instruction attacks, as follows:

1. **Prevention of cache instruction attacks:** The evaluation of the security capability of FlushBlocker was based on whether the FlushBlocker kernel can prevent the Meltdown, Spectre, and Rowhammer PoC code execution.
2. **Prevention of FlushBlocker DMs:** The evaluation of the FlushBlocker kernel on whether it can prevent DM1 through DM4 that attempt to issue and execute clflush instructions.

7.1.1 Prevention of Cache Instruction Attacks

The evaluation of practical security capability is based on the prevention of cache instruction attacks from Meltdown, Spectre, and Rowhammer PoC codes [21]–[23]. The comparison is between the vanilla kernel without KPTI [9], [11] and Retpoline [10] and a kernel with FlushBlocker. Additionally, in the experiment of Rowhammer, the PoC code is attempted on double-sided Rowhammer for 10 hours.

Table 1 lists the security capability results. The kernel with FlushBlocker successfully prevented leaks using Meltdown and Spectre. Additionally, FlushBlocker had no bit flips, whereas the vanilla kernel had 2-bit flips.

7.1.2 Prevention of FlushBlocker DMs

FlushBlocker was evaluated both with and without countermeasures for the four DMs. The DMs attempted to execute the clflush instruction on the kernel using FlushBlocker.

Table 2 shows that FlushBlocker without countermeasures (w/o CM) could not prevent cache flush instruction execution from DMs. Conversely, FlushBlocker with countermeasures (w/ CM) detected and stopped all the DMs from

Table 1 Prevention result of FlushBlocker for cache instruction attacks (✓ Success; – Failure)

Attack	Description	Vanilla kernel	FlushBlocker
Meltdown	PoC [21] w/o KPTI	–	✓
Spectre	PoC [22] w/o Retpoline	–	✓
Rowhammer	PoC [23]	–	✓

Table 2 Prevention result of FlushBlocker for defeating methods (✓ clflush not available; – clflush available)

Attack	Description	w/o CM	w/ CM
DM1	clflush through child user process	–	✓
DM2	clflush on the mmap's new page	–	✓
DM3	clflush through the mprotect	–	✓
DM4	five clflush instructions	–	✓

issuing clflush instructions.

7.2 Measurement of Performance and Effect of FlushBlocker

The objectives of evaluating FlushBlocker performance were to measure the overhead cost with benchmark software and analyze benign applications for their effect on the FlushBlocker kernel, as follows:

1. **Performance measurements:** Measuring the performance of FlushBlocker implementation, in order to determine the performance overhead for the kernel processing time using the benchmark software on the FlushBlocker kernel.
2. **Analysis of benign applications:** Analyzing the binary files of benign applications that contain cache flush instructions in order to determine whether restricting cache flush instructions have any impact.

7.2.1 Performance Measurements

For the measurement of the performance, a comparison between a vanilla kernel and the FlushBlocker kernel was conducted. These kernels are compiled with default configuration enabled security features such as KPTI.

The benchmark programs, UnixBench [24], SPEC CPU 2017 benchmark suite [25] and real-world server applications, were used to determine the average performance overhead. The real-world server application's performance was evaluated using Apache httpd 2.4.46 [26] with ApacheBench 2.3 [27] and Redis 6.2.3 [28] with Redis-benchmark 6.2.3 [29]. Both benchmarks create 10,000 requests with different client threads to calculate the requests per second in a network speed of 1 Gbps.

UnixBench: Fig. 4 shows the result of UnixBench. In UnixBench, the overall performance score (*System Benchmarks Index Score*) of FlushBlocker was approximately 31.73% less than vanilla kernel's score. When viewed as each benchmark result, the decrease of performance was more in *Execl Throughput* (557.21%), *Process Creation* (25.96%), *Shell Scripts 1 concurrent* (87.38%) and *Shell Scripts 8 concurrent* (69.10%).

SPEC CPU 2017: Fig. 5 presents that the result of the SPEC CPU 2017 benchmark suite that is executed in intrate mode. In the SPEC CPU benchmark programs, FlushBlocker does not induce performance overhead as compared to the execution of the vanilla kernel.

Server Applications: Fig. 6 and Fig. 7 show the results of the server application's performance. The results for each request per second with different client threads employ the median of five times measurements which is equal to five

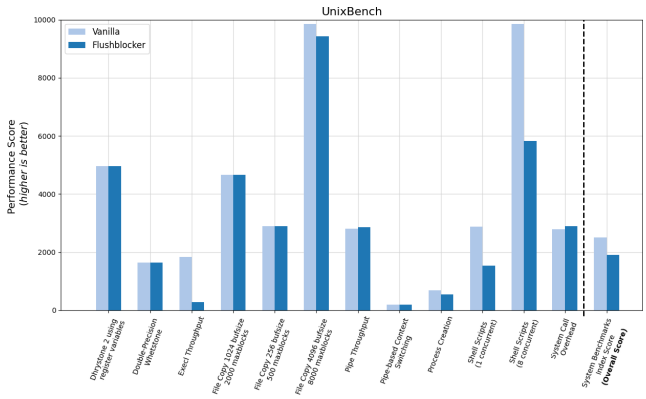

Fig. 4 Performance score of the UnixBench

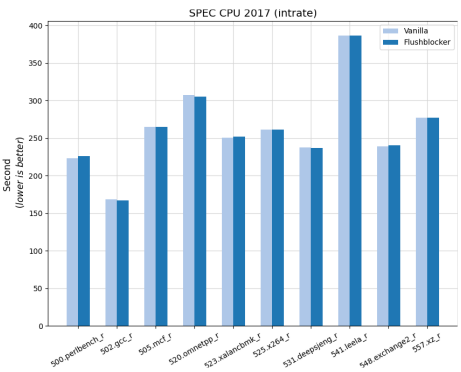

Fig. 5 Execution time of SPEC CPU 2017 benchmark (intrate mode)

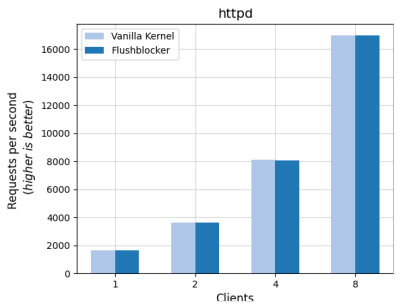

Fig. 6 Requests per second of httpd with ApacheBench

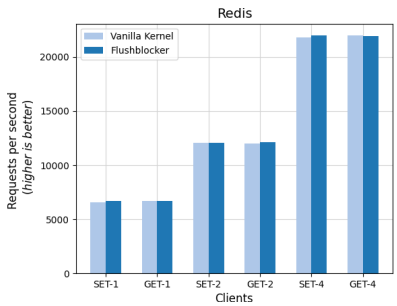

Fig. 7 Requests per second of Redis with Redis-benchmark

Table 3 Number of clflush instructions' benign applications

Environment	Total apps	clflush apps	clflushopt apps
Ubuntu Linux 20.04.1 LTS	1,484	1	0
Debian GNU/Linux 10	1,462	1	0
CentOS Linux release 7.9.2009	1,003	0	0
Linux Mint 19.3	2,182	0	0

times of the original. In addition, in the case of the server applications, a performance difference between FlushBlocker and the vanilla kernel was not seen.

7.2.2 Analysis of Benign Applications

The statical analysis of benign applications was conducted to determine whether clflush and clflushopt instructions are included in the ELF binary applications. It unveils the effects of FlushBlocker on the default installation setting of major Linux distributions.

Table 3 presents the results of the statical analysis, which indicate that one ELF binary gnome-control-center contains one clflush instruction pattern on Ubuntu Linux 20.04.1 LTS and Debian GNU/Linux 10. However, the gnome-control-center does not call clflush as the instruction pattern is found in the data section.

7.3 Evaluation Environment

FlushBlocker was evaluated using Linux kernel 5.7.15. The evaluation environment was executed on a physical machine 1, which was equipped with an Intel (R) Core (TM) i9-10900T (1.90 GHz, x86_64) processor with 32 GB memory, and physical machine 2, which was equipped with an Intel (R) Core (TM) i7-4800MQ (2.70 GHz, x86_64) processor with 16 GB memory for the client. The Linux distribution used was Ubuntu 20.04.1 LTS, which required 5 source files and 868 lines for the Linux kernel 5.7.15.

8. Discussion

8.1 Evaluation Consideration

FlushBlocker can successfully intercept the execution of cache flush instructions for the user process, it can prevent actual attacks from Meltdown, Spectre, and Rowhammer PoC codes that try to issue cache flush instructions. In addition, FlushBlocker counters DM1 through DM4, which attempts to subvert the scanning and monitoring of cache flush instructions, therefore, it can protect the kernel from cache instruction attacks.

The performance evaluation results indicate that FlushBlocker implementation requires an additional kernel processing time. The reason is that FlushBlocker needs page scanning during new code page creation time such as process creation. For that reason, in the performance measurement of UnixBench, benchmarks related to the create new user process had high overheads. Therefore, the workloads that require to repeat creating user processes that have a property of short-lived (e.g., Fuzzing) may be affected in performance. To evade this, FlushBlocker provides a policy for avoiding page scanning of user processes created by a trusted user. On the other hand, long-lived applications such as SPEC CPU 2017 benchmark suite and server applications were zero overhead.

8.2 Limitation

8.2.1 Design Limitation

The primary design limitation of FlushBlocker is the filtering of cache flush instructions from the user process. Although it potentially affects benign application behavior, FlushBlocker only causes the performance overhead due to cache hit-miss because cache flush instructions do not directly occur in data modification and control flow.

Additionally, it is necessary to prevent a FLUSH+RELOAD attack by establishing a complex cache control without cache flush instructions, which can support the filtering instructions pattern at the scanning of pages during the user process execution, when this attack is published as a PoC code.

8.2.2 Implementation Limitation

The implementation limitations of FlushBlocker must be considered. Trapping relies on the number of hardware debug registers (e.g., five debug registers on the x86 architecture) at the registration of virtual addresses for cache flush instructions. The FlushBlocker implementation drops the execution privilege of pages to trap cache flush instructions across multiple pages.

FlushBlocker handles a page fault, when the user process attempts to execute the cache flush instruction on the non-executable page. Then, FlushBlocker registers the virtual address of the page fault into the hardware debug register to avoid hardware limitation.

However, the worst case of user process stores more than five cache flush instructions per page. This is a suspicious behavior for a usual application because benign applications do not contain cache flush instructions from the evaluation result (Sect. 7.2.2).

8.3 Portability Consideration

The implementation of FlushBlocker applicability proceeds to the kernel of another OS kernels by adopting a page management mechanism on the x86_64 architecture.

In this study, FlushBlocker portability was carefully examined in other OS kernels, with a trapping capability of the user process. The Windows kernel provides the trapping function for user process creation at the kernel-mode driver API[†], and the referring and setting functions[††] handle the

[†]PsSetCreateProcessNotifyRoutine
[††]PsGetContextThread and PsSetContextThread

debug register of the user application thread.

9. Related Work

Side-Channel Attacks: Meltdown adopts a FLUSH+RELOAD attack [1], which targets the CPU L3 caches to access the secret data of the user processes and kernels. The FLUSH+RELOAD attack calculates the access time of the information after flushing the entire cache. A short access time indicates that the information has leaked from the cache. The FLUSH+FLUSH [3] attack is another flush based side-channel method that guesses secret data by observing the execution time difference of clflush after caching a target element.

Side-Channel Attack Countermeasures: KPTI [9] adopts a page table isolation method that prevents side-channel attacks from the user mode to the kernel mode. KPTI incurs a page table switching cost to update the CR3 register. EPTI [30] reduces the overhead of the KPTI by separating the page table isolation mechanism with an Intel extended page table (EPT) for a guest OS that does not require KPTI. ConTExT [31] is a mechanism for mitigating Spectre-style attacks. In ConTExT, a page storing secret data is marked as a "non-transient page", and it is tracked by hardware taint tracing for registers containing the secret data for transient execution instructions.

9.1 Comparison with Existing Protection Mechanisms

The objective is to determine which characteristics fulfills the side-channel countermeasures requirements. Table 4 shows a comparison of features supported by FlushBlocker with those of existing protection mechanisms [8]–[11], [30]–[32]. Although existing hardware and software based protection mechanisms can prevent existing cache side-channel attacks, FlushBlocker cannot prevent all of those attacks. On the other hand, FlushBlocker can mitigate the effect of cache instruction attacks that employ latest side-channels vulnerability in the future, existing mechanisms cannot mitigate the attacks.

When a latest side-channel attack is disclosed, researchers and developers also have to design and develop an additional countermeasure for the attack. Since developing a countermeasure takes time, FlushBlocker can mitigate an attack until releasing the countermeasure and merging it to OS kernels, compilers and hardware.

Table 4 Comparison of side-channel countermeasures features (✓ is supported; △ is partially supported).

Feature	Existing Hardware Protections	Existing Software Protections	Flush-Blocker
Existing Cache Attacks	✓	✓	△
Immediately Mitigation			✓
Undisclosed Cache Attacks			△

10. Conclusion

The latest OS kernel must be able to prevent cache instruction attacks (e.g., Meltdown, Spectre, and Rowhammer). Software based countermeasures that adopt KPTI, Retpoline, and previous research approaches require performance overhead. Other countermeasures with CPU virtualization require hardware support. These restrict deployment in many environments (e.g., cloud, embedded or mobile environment).

In this paper, a novel security design that prevents cache instruction attacks at the kernel layer, FlushBlocker is proposed. To prevent cache instruction attacks, FlushBlocker traps and monitors the cache flush instructions of the user process from being executed. The design of FlushBlocker transparently prohibits the cache flush instructions at the kernel component to reduce the performance overhead. Although, the adversaries attempt to execute cache instruction attacks, secret data from CPU caches cannot be easily captured as the CPU cache instructions fail.

The evaluation result of the Linux kernel with FlushBlocker deals with preventing the PoC codes of Meltdown, Spectre, and Rowhammer from being executed. FlushBlocker counters the four DMs and achieves no overhead for real-world applications. Additionally, the static analytic result of benign applications, in which an application contains the data of cache flush instruction, shows no side effects.

Acknowledgements

Hiroki's contribution contained in the paper is done when he belonged to SECOM Co., Ltd.

References

[1] Y. Yarom and K. Falkner, "FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack," Proc. 23rd USENIX Security Symposium (Security '14), San Diego, California, USA, pp.719–732, Aug. 2014.

[2] X. Zhang, Y. Xiao, and Y. Zhang, "Return-Oriented Flush-Reload Side Channels on ARM and Their Implications for Android Devices," Proc. 23rd ACM Conference on Computer and Communications Security (CCS '16), Vienna, Austria, pp.858–870, Oct. 2016.

[3] D. Gruss, C. Maurice, K. Wagner, and S. Mangard, "Flush+Flush: A Fast and Stealthy Cache Attack," Proc. 13th Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA '16), Donostia-San, Sebastián, Spain, vol.9721, pp.279–299, July 2016.

[4] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading Kernel Memory from User Space," Proc. 27th USENIX Security Symposium (Security '18), Baltimore, Maryland, USA, pp.973–990, Aug. 2018.

[5] K. Paul, H. Jann, F. Anders, G. Daniel, G. Daniel, H. Werner, H. Mike, L. Moritz, M. Stefan, P. Thomas, S. Michael, and Y. Yuval, "Spectre Attacks: Exploiting Speculative Execution," Proc. 40th IEEE Symposium on Security and Privacy (S&P '19), San Francisco, California, USA, pp.1–19, May 2019.

[6] D. Gruss, M. Lipp, M. Schwarz, D. Genkin, J. Juffinger, S. O'Connell, W. Schoechl, and Y. Yarom, "Another Flip in the Wall

of Rowhammer Defenses," Proc. 39th IEEE Symposium on Security and Privacy (S&P '18), San Francisco, California, USA, pp.245–261, May 2018.
[7] Google Project Zero, "Exploiting the DRAM rowhammer bug to gain kernel privileges," Google, https://googleprojectzero.blogspot.com/2015/03/exploiting-dram- rowhammer-bug-to-gain.html, accessed Feb. 02. 2022.
[8] Intel "Cascade Lake: Overview," Intel.com, https://www.intel.com/content/www/us/en/products/platforms/details/cascade-lake.html, accessed Feb. 02. 2022.
[9] The kernel development community, "Page Table Isolation (PTI)," Kernel.org, https://www.kernel.org/doc/html/latest/x86/pti.html, accessed Feb. 02. 2022.
[10] P. Turner, "Retpoline: A Branch Target Injection Mitigation," Google.com, https://support.google.com/faqs/answer/7625886, accessed Feb. 02. 2022.
[11] D. Gruss, M. Lipp, M. Schwarz, R. Fellner, C. Maurice, and S. Mangard, "KASLR is Dead: Long Live KASLR," Proc. 9th International Symposium on Engineering Secure Software and Systems (ESSoS '17), Bonn, Germany, vol.10379, pp.161–176, Feb. 2017.
[12] J. Corbet, "KAISER: hiding the kernel from user space," LWN.net, https://lwn.net/Articles/738975/, accessed Feb. 02. 2022.
[13] J. Corbet, "Kernel page-table isolation merged," LWN.net, https://lwn.net/Articles/742404/, accessed Feb. 02. 2022.
[14] S. Enomoto and H. Kuzuno, "FlushBlocker: Lightweight mitigating mechanism for CPU cache flush instruction based attacks," Proc. 6th European Symposium on Security and Privacy Workshops (EuroS&PW '21), Vienna, Austria, pp.74–79, Sept. 2021.
[15] The kernel development community, "Kernel Samepage Merging," Kernel.org, https://www.kernel.org/doc/html/latest/admin-guide/mm/ ksm.html, accessed Feb. 02. 2022.
[16] The GnuPG Project, "The GNU Privacy Guard," GnuPG.org, https://gnupg.org/, accessed Feb. 02. 2022.
[17] J. Corbet, "Kernel address randomization," LWN.net, https://lwn.net/Articles/444503/, accessed Feb. 02. 2022.
[18] C. Canella, M. Schwarz, M. Haubenwallner, M. Schwarzl, and D. Gruss, "KASLR: Break It, Fix It, Repeat," Proc. 15th ACM Asia Conference on Computer and Communications Security (ASIA CCS '20), Taipei, Taiwan, pp.481–493, Nov. 2020.
[19] M.S. Inci, B. Gulmezoglu, G. Irazoqui, T. Eisenbarth, and B. Sunar, "Cache Attacks Enable Bulk Key Recovery on the Cloud," Proc. 18th International Conference on Cryptographic Hardware and Embedded Systems (CHES '16), Santa Barbara, California, USA, vol.9813, pp.368–388, Aug. 2016.
[20] Fortinet, "Meltdown/Spectre Update," Fortinet.com, https://www.fortinet.com/blog/threat-research/the-exponential- growth-of-detected-malware-targeted-at-meltdown-and-spectre
[21] paboldin, "meltdown-exploit," Github.com, https://github.com/paboldin/meltdown-exploit, accessed Feb. 02. 2022.
[22] Eugnis, "spectre-attack," Github.com, https://github.com/Eugnis/spectre-attack, accessed Feb. 02. 2022.
[23] google, "rowhammer-test," Github.com, https://github.com/google/rowhammer-test, accessed Feb. 02. 2022.
[24] kdlucas, "byte-unixbench," Github.com, https://github.com/kdlucas/byte-unixbench, accessed Feb. 02. 2022.
[25] Standard Performance Evaluation Corporation, "SPEC CPU® 2017," SPEC.org, https://www.spec.org/cpu2017/, accessed Feb. 02. 2022.
[26] The Apache Software Foundation, "The Apache HTTP Server Project," Apache.org, http://httpd.apache.org/, accessed Feb. 02. 2022.
[27] The Apache Software Foundation, "ab - Apache HTTP server benchmarking tool," Apache.org, https://httpd.apache.org/docs/2.4/programs/ab.html
[28] Redis Labs, "Redis," Redis.com, https://redis.io/, accessed Feb. 02. 2022.
[29] Redis Labs, "How fast is Redis?," Redis.com, https://redis.io/topics/benchmarks, accessed Feb. 02. 2022.
[30] Z. Hua, D. Du, Y. Xia, H. Chen, and B. Zang, "EPTI: Efficient Defence against Meltdown Attack for Unpatched VMs," Proc. 2018 USENIX Annual Technical Conference (ATC '18), Boston, Massachusetts, USA, pp.255–266, July 2018.
[31] M. Schwarz, M. Lipp, C. Canella, R. Schilling, F. Kargl, and D. Gruss, "ConTExT: A Generic Approach for Mitigating Spectre," Proc. 27th Annual Network and Distributed System Security Symposium (NDSS '20), San Diego, California, USA, Feb. 2020.
[32] R.K. Konoth, M. Oliverio, A. Tatar, D. Andriesse, H. Bos, C. Giuffrida, and K. Razavi, "ZebRAM: Comprehensive and Compatible Software Protection Against Rowhammer Attacks," Proc. 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI '18), Carlsbad, California, USA, pp.697–710, Sept. 2018.

Shuhei Enomoto received his B.E. and M.E. degrees from Tokyo University of Agriculture and Technology in 2020 and 2022. His research interests include operating systems, virtualization, and software security.

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 cyber security specifically on networks and operating systems. He is a member of IEICE and IPSJ.

Hiroshi Yamada received his B.E. and M.E. degrees from the University of Electro-communications in 2004 and 2006. He received his Ph.D. degree from Keio University in 2009. He is currently an associate professor at Tokyo University of Agriculture and Technology. His research interests include operating systems, virtualization, and cloud computing. He is a member of IEEE/CS, ACM, and USENIX.