FlushBlocker: Lightweight mitigating mechanism for CPU cache flush instruction based attacks

Shuhei Enomoto
Tokyo University of Agriculture and Technology
Japan
enomoto@asg.cs.tuat.ac.jp

Hiroki Kuzuno
Intelligent Systems Laboratory, SECOM Co, Ltd.
Japan
kuzuno@s.okayama-u.ac.jp

***Abstract*—CPU cache flush instruction based attacks (cache instruction attacks) such as FLUSH+RELOAD can function in many environments. Meltdown and Spectre adopt FLUSH+RELOAD with cache instructions to access secret data. Additionally, Rowhammer employs cache instructions to modify data in physical memory. An adversary can read and write arbitrary data using these attacks. The deployment of corresponding hardware to combat these attacks is difficult for users, and existing software-based countermeasures incur high overheads, or cannot be applied to a variety of machines.**

In this study, we propose a novel mitigation mechanism for cache instruction attacks called FlushBlocker, which employs an effective approach that focuses on restricting cache flush instructions. We implemented FlushBlocker on the latest Linux kernel to conduct experiments. The experimental results indicate that FlushBlocker prevents existing cache instruction attacks and runtime overhead is negligible.

***Index Terms*—side-channel attack, operating system, security**

1. Introduction

Hardware devices have been targeted by CPU cache flush instruction based attacks (in short, cache instruction attacks). An attacker issues instructions that control CPU cache sectors (e.g., clflush in x86) to read or write arbitrary data. Meltdown [1] and Spectre [2] adopted FLUSH+RELOAD [3], [4] with transient execution of CPU to access protected information on an operating system (OS) kernel or a user process (e.g., x86 or ARM architecture). Additionally, Rowhammer [5] used cache instructions during the attack phase to flip bits in a physical memory (e.g., DDR3 memory) to implement an actual attack (e.g., privilege escalation) [6].

Protection against Meltdown and Spectre with FLUSH+RELOAD by disabling last-level cache (LLC) and prohibiting of the out-of-order and speculative features require performance degradation. Moreover, the memory with an error detection feature (e.g., ECC memory) is effective in mitigating 1-bit flip based Rowhammer attack. However, ECC memory can not protect against more than 2-bit flips at a time [7].

The current software based countermeasures have two challenges unaddressed. First, to prevent Meltdown / Spectre, Kernel page table isolation (KPTI) [8], [9] contributes approximately 22% of the overhead [10], and Retpoline [11] requires 66% overhead [10]. Additionally overhead reducing approaches require hardware support. Hardware based taint tracking Spectre mitigation [12] and modifying the source code in the hypervisor's layer to mitigate Meltdown [13] and Rowhammer [7] is difficult to be applied by cloud users.

In this paper, we propose FlushBlocker, a novel mitigation approach against cache instruction attacks such as FLUSH+RELOAD based Meltdown / Spectre and Rowhammer. FlushBlocker achieves two objectives. First, it focuses on preventing cache instruction attacks with a low overhead. It prohibits the cache flush instructions from accessing the user process by scanning the instructions. Second, FlushBlocker supports various environments and minimizes the need for hardware features. It is deployed as a kernel component with countermeasures against defeating methods. Moreover, we consider several defeating methods that forcibly circumvent FlushBlocker to utilize kernel capabilities such as user process creation or memory mapping. We meticulously produced countermeasures for each defeating method, without side effects at the kernel layer.

The contributions of this study are summarized as follows:

1) We present FlushBlocker's design for mitigating cache instruction attacks without hardware support.
2) Implementing FlushBlocker provides an additional kernel mechanism that corresponds to defeating methods. In addition, FlushBlocker transparently scans and traps the cache flush instructions for the user process.
3) We evaluated the implemented FlushBlocker in the latest Linux kernel where it successfully prevented Meltdown's, Spectre's, and Rowhammer's proofs-of-concept (PoC) attacks and defeated cache instruction attack. In addition, we measured FlushBlocker ' s performance using benchmarks, and demonstrate that the performance overhead is only 0.013%–8.60% that of native kernel.

2. Background

2.1. FLUSH+RELOAD Attack

Meltdown [1] and Spectre [3] employ FLUSH+RELOAD [3], [4] with transient execution

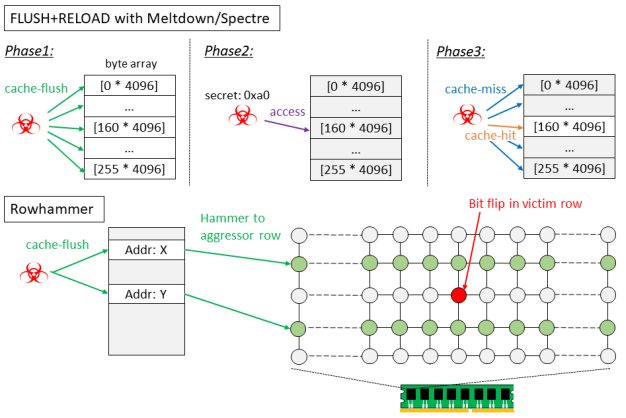

Figure 1. FLUSH+RELOAD and Rowhammer attack overview

CPU vulnerabilities to deliver a side-channel attack. These attacks leak secret data from the victim's kernel or user process to the attacker's user process in LLC.

Figure 1 shows the three phases of a FLUSH+RELOAD attack for estimating one byte of the secret data. First, an attacker creates a malicious user process that contains the FLUSH+RELOAD and Meltdown / Spectre payload. The user process allocates a one-byte array from *array[0*4096]* to *array[255*4096]* and executes cache flush instructions to evict all the cached elements from the CPU cache. Next, the user process obtains the one-byte secret data leaked by Meltdown / Spectre and it accesses *array[secret*4096]* to store the secret data in the CPU cache. Finally, the user process calculates the access time for each index of the array. Since *array[secret*4096]* is the shortest access time, the user process determines the secret data.

Although Meltdown and Spectre can combine with other CPU cache based attack methods (e.g., PRIME+PROBE [14]), we analyzed actual Meltdown/Spectre based malwares[1] indicating that ten out of the total twelve samples contained FLUSH+RELOAD.

2.2. Rowhammer Attack

The Rowhammer [5] is a one of the fault injection attacks in DRAM. An attacker repeatedly and with a high frequency accesses memory location called memory rows, to flip the data bits stored there. This allows write access to OS kernel data for privilege escalation [6].

Figure 1 shows an overview of Rowhammer. First, an attacker identifies a target memory row (*victim row*) that relates to the target user's or kernel's virtual address. In addition, it also identifies the virtual addresses related to the memory rows around the *victim row* (*aggressor rows*). Next, they access the virtual addresses of the *aggressor rows* with the cache flush instruction.

We analyzed the samples of Rowhammer's PoC attack published as open-sourced projects, which indicate that seven of the total ten samples contained cache flush instructions.

2.3. Threat Model

We postulate that the threat model of the adversary's environment is as follows:

[1]We downloaded samples from hybrid-analysis.com

Permission: An adversary has normal user privileges. For example, an adversary can access file system, networking and control processes as a non-root user. However, they cannot insert malicious kernel modules into the OS kernel.

Attack-Method: An adversary attacks the system with cache instruction attacks such as FLUSH+RELOAD based Meltdown / Spectre and Rowhammer. Through such attacks, an adversary attempts to obtain read and write access to arbitrary data.

3. Proposed Approach

3.1. Requirements of FlushBlocker

We propose FlushBlocker with a low overhead for kernel processing to meet the requirements of mitigating the cache instruction attacks.

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

To satisfy requirement 1, FlushBlocker targets the kernel component design to reduce the negative effects of mitigating the processes for kernel processing. FlushBlocker ensures that user applications incur running costs in the native environment.

To satisfy requirement 2, FlushBlocker does not rely on hardware features to cover multiple architectures and kernel designs for easy porting of the mitigation process.

3.2. Approach of FlushBlocker

FlushBlocker focuses on cache instruction attacks as it needs specific cache flush instructions in the attacking phase and this helps in restricting the cache flush instructions to mitigate the attacks. When using FlushBlocker, the malicious user process cannot obtain secret data from the CPU cache because it is difficult to identify the byte of data that matches the CPU cache during the cache access calculation. In addition, the malicious user process cannot repeatedly access memory row with cache flush instructions. Therefore, Meltdown and Spectre with FLUSH+RELOAD attacks and Rowhammer attack processes cannot easily execute the cache instruction attacks.

4. Design and Implementation

4.1. Monitoring of User Process

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

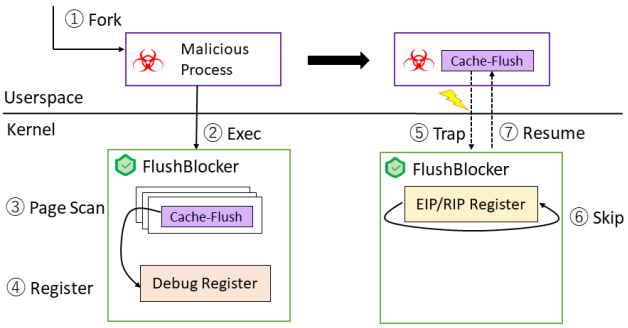

Figure 2. FlushBlocker design overview

2) The kernel loads the malicious user process to memory through the program execution sequence.
3) FlushBlocker scans all the 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 when the pages contain such cache flush instructions, and subsequently, identifies the malicious user process as a monitoring target.
5) The malicious user process initiates a cache instruction attack.
6) FlushBlocker traps the execution of the cache flush instructions at the kernel. When a trap occurs, FlushBlocker determines if 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 the execution position of the program after the cache flush instructions.

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

4.2. 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 regarding treatment of the trapped instructions to the root user. In the current design, root user can set a number for tolerance of cache flush instructions per one second of one process. Based on the setting, FlushBlocker determines whether or not to skip the instructions.

4.3. Defeating Methods of FlushBlocker

Several defeating methods 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.

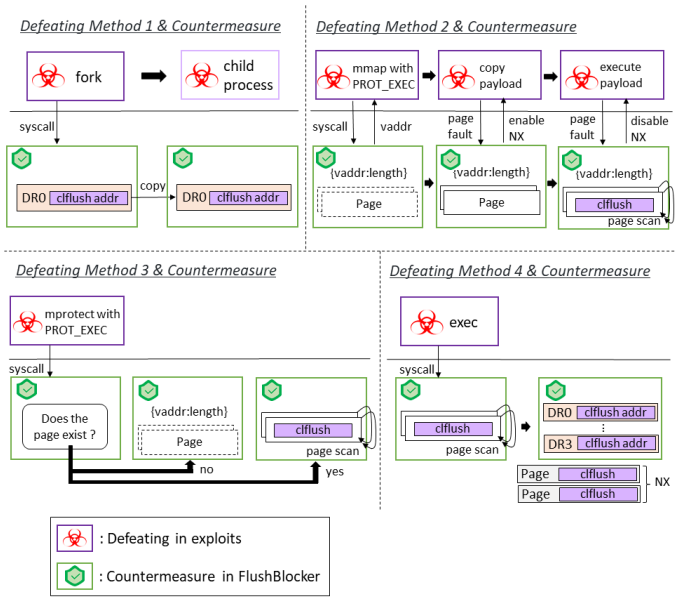

Figure 3. Countermeasures against defeating FlushBlocker

Defeating Method 2 (DM2): When 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.

4.4. Countermeasures for the Defeating Methods

FlushBlocker provides countermeasures for DMs (Figure 3).

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 at the `exec` system call timing. FlushBlocker adopts the following countermeasures to traps the cache flush instruction of an additional page:

1) A malicious user process creates an additional page with an execution flag through the memory allocation sequence.
2) FlushBlocker stores the starting and ending virtual addresses of the page during the memory allocation sequence.
3) A malicious user process copies the program code payload containing the cache flush instruction.
4) FlushBlocker triggers the page writing and demands paging of page faults at the kernel to deter-

mine if 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 at 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 a page. Moreover, FlushBlocker 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.

4.5. Implementation

We implemented FlushBlocker on a Linux kernel with x86_64 architecture.

4.5.1. Page Scanning. Page scanning of the user process at the `exec` system call invocation. FlushBlocker reads every code page of the user process and searches 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, our prototype targets the two instructions.

4.5.2. 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.

4.5.3. Trapping of Cache Flush Instructions. FlushBlocker provides a configurable policy related to a number for tolerance of cache flush instructions. In the current implementation, FlushBlocker uses the procfs interface (`/proc/flushblocker`) to obtain the setting. To avoid access of the setting by a malicious user, only the root user has read and write access to the file. When the hardware debug interruption occurs, FlushBlocker identifies if the user process is being monitored by using a monitoring flag variable. In case of exceeding tolerance, FlushBlocker skips the instructions to increase the `IP` register for the malicious user process.

4.5.4. 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 instruction 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 instruction of the 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 the 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 debug register entries using the least recently used algorithm when a page fault has occurred and exchanges addresses on the hardware debug registers with NX bit handling.

5. Evaluation

The objectives of evaluating FlushBlocker are as follows:

1) **Security capability of FlushBlocker:** We evaluated whether the FlushBlocker kernel can prevent the Meltdown, Spectre, and Rowhammer PoC code execution. Additionally, we assessed whether the FlushBlocker kernel can prevent DMs 1 through 4 that attempt to issue and execute clflush instructions.
2) **Performance measurements:** We measured the performance overhead for the kernel processing

TABLE 1. PREVENTION RESULT OF FLUSHBLOCKER FOR CACHE INSTRUCTION ATTACKS (✓ SUCCESS; – FAILURE)

Attack	Description	Vanilla kernel	FlushBlocker
Meltdown	PoC [15] w/o KPTI	–	✓
Spectre	PoC [16] w/o Retpoline	–	✓
Rowhammer	PoC [17]	–	✓

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	–	✓

time using the benchmark software on the FlushBlocker kernel.

3) **Analysis of benign applications:** We statically analyzed the binary file of benign applications that contain cache flush instructions to determine the effects of restricting cache flush instructions.

FlushBlocker was evaluated on a Linux kernel 5.7.15. The evaluation environment was executed on physical machine 1 equipped with an Intel (R) Core (TM) i9-10900T (1.90 GHz, x86_64) processor with 32 GB memory, and physical machine 2 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; FlushBlocker's implementation required 664 lines in the Linux kernel.

5.1. Security Capability of FlushBlocker

First, we compared the security capabilities by using actual Meltdown, Spectre, and Rowhammer PoC codes [15]–[17] on a vanilla kernel without KPTI [8], [9] and Retpoline [11] and a kernel with FlushBlocker. In the experiment of Rowhammer, the PoC code attempted to double-sided Rowhammer for 10 hours. Table 1 lists the security capability results. We ensured that FlushBlocker successfully prevented leaks using Meltdown and Spectre. Additionally, FlushBlocker had no bit flips, whereas vanilla kernel had 2-bit flips.

Next, we evaluated FlushBlocker without countermeasures and FlushBlocker that adopted countermeasures for the four DMs. The DMs attempted to execute the clflush instruction on the kernel with 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 issuing clflush instructions.

5.2. Performance Measurements

For performance measurement, we used original micro-benchmark program and real-world server applications to compare the vanilla kernel and the kernel with FlushBlocker. The vanilla kernel is compiled with default config enabled security features such as KPTI.

First, We measured the FlushBlocker overhead using the original benchmark program. The original benchmark program uses a fork and exec library call that creates a new user process, executing a `pwd` command. Figure 4 indicates that FlushBlocker requires an overhead of 8.60% for the original benchmark program.

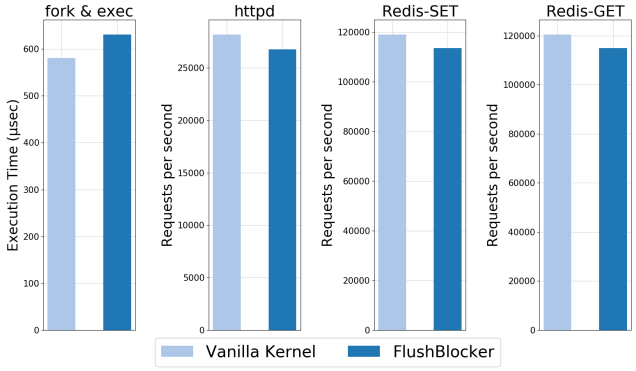

Figure 4. Results of the benchmark software

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

Next, we measured the real-world application's performance using Apache httpd 2.4.46 with ApacheBench 2.3 and Redis 6.2.3 with Redis-benchmark 6.2.3. Both benchmarks create 100 clients and send 10,000 requests to calculate the requests per second with a network speed of 1 Gbps. Figure 4 shows the results of the applications. FlushBlocker experiences a 0.013% slowdown for httpd and 0.045% - 0.046% slowdown for Redis.

5.3. Analysis of Benign Applications

We analyzed benign applications to determine whether clflush and clflushopt instructions are included in the ELF binary applications on the default installation setting of major Linux distributions to avoid the effects of FlushBlocker. Table 3 presents the results of the analysis. The results show 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 doesn't call clflush because the instruction pattern is found in data section.

6. Discussion

Evaluation Consideration: The evaluation results indicate that FlushBlocker can prevent actual attacks from Meltdown, Spectre, and Rowhammer PoC code. In addition, FlushBlocker counters DM1 through DM4 that attempt to prevent the scanning and monitoring of cache flush instructions. The performance evaluation results indicate that FlushBlocker has the kernel processing performance overhead from 0.013% to 8.60%.

FlushBlocker achieves the light performance effect that requires page scanning during the user process creation time and its function processing is only invoked from the debug register entries. The results of ApacheBench and Redis-benchmark show that the server applications are low overhead than micro-benchmark such as simple fork-exec. The reason is that the server applications have fewer

syscalls leading to FlushBlocker during the benchmark time. Additionally, FlushBlocker employs a page fault mechanism to counter DM2 - DM4, we will evaluate the overhead of the actual page fault for cache flush trapping.

Implementation Consideration: The implementation of FlushBlocker drops the execution privilege of pages to trap cache flush instructions across multiple pages. If the user process stores more than five cache flush instructions per page, FlushBlocker stops the running user process on the x86 architecture. FlushBlocker then handles these pages, which contain over five cache flush instructions in the future.

Portability Consideration: The implementation of FlushBlocker proceeds to the kernel of another OS by adopting a page management mechanism on the x86_64 architecture. The Windows kernel provides the trapping function for user process creation at the kernel-mode driver API [2] and the referring and setting functions [3] handle the debug register of the user application thread.

7. Related Work

Side-Channel Attacks: Meltdown adopts a FLUSH+RELOAD attack [3], which targets CPU L3 caches to access the secret data of user processes and kernels. The FLUSH+RELOAD attack calculates the access time of information after flushing the entire cache. A short access time indicates that information has leaked from the cache. The FLUSH+FLUSH [18] 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 [13] 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 [12] 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.

Comparison with Countermeasures: We compared the features of FlushBlocker with other mechanisms. Although KPTI [8] and EPTI [13] prevent Meltdown from the userspace to the kernel, KPTI has a 22% overhead cost for prevention [10] and EPTI requires code modification of the hypervisor. FlushBlocker directly inspects the behavior of malicious user processes using cache flush instructions without page table isolation with low overhead and high portability. To address other cache flush instruction attacks (e.g., PRIME+PROBE [14]), FlushBlocker must adopt instruction filtering customized for each cache controlling phase.

8. Conclusion

Cache instruction attacks are widely adopted by Meltdown, Spectre, and Rowhammer attacks. As software based countermeasures against cache instruction attacks incur performance overheads and another countermeasures with CPU virtualization require hardware support, these approaches cannot be deployed in many environments, such as embedded or mobile hardware.

Our novel design for cache instruction attack mitigation, FlushBlocker, enables CPU cache flush instructions to be trapped and monitored to prevent FLUSH+RELOAD and Rowhammer attacks. The design of FlushBlocker focuses on the kernel component that transparently prohibits the CPU cache flush instruction for the user process from being executed. Hence, adversaries cannot easily leak the secret data from CPU caches because the FLUSH+RELOAD and Rowhammer attacks have failed.

An evaluation of FlushBlocker confirms that it can prevent the PoC code of Meltdown, Spectre, and Rowhammer from being executed. Moreover, FlushBlocker counters the four FlushBlocker DMs, and achieves a low-performance overhead of 0.013%–8.60% for kernel processing.

[2] PsSetCreateProcessNotifyRoutine

[3] PsGetContextThread and PsSetContextThread

References

[1] Moritz Lipp, *et al.* Meltdown: Reading kernel memory from user space. In Security '18, pages 973–990. USENIX, 2018.

[2] Paul Kocher, *et al.* Spectre attacks: Exploiting speculative execution. In S&P '19. IEEE, 2019.

[3] Yuval Yarom, *et al.* Flush+reload: A high resolution, low noise, l3 cache side-channel attack. In Security '14, pages 719–732. USENIX, 2014.

[4] Zhang Xiaokuan, *et al.* Return-oriented flush-reload side channels on arm and their implications for android devices. In CCS '16, page 858–870. ACM, 2016.

[5] Google Project Zero. Exploiting the DRAM rowhammer bug to gain kernel privileges. https://googleprojectzero.blogspot.com/2015/03/exploiting-dram-rowhammer-bug-to-gain.html, 2015.

[6] Daniel Gruss, *et al.* Another flip in the wall of rowhammer defenses. In S&P '18, pages 245–261. IEEE, 2018.

[7] Radhesh Krishnan Konoth, *et al.* Zebram: Comprehensive and compatible software protection against rowhammer attacks. In OSDI '18, pages 697–710. USENIX, 2018.

[8] Kernel.org. Page table isolation (pti). https://www.kernel.org/doc/html/latest/x86/pti.html.

[9] Daniel Gruss, *et al.* KASLR is dead: Long live KASLR. In ESSoS '17, pages 161–176. Springer, 2017.

[10] Xiang (Jenny) Ren, *et al.* An analysis of performance evolution of linux's core operations. In SOSP '19, pages 554–569. ACM, 2019.

[11] Paul Turner. Retpoline: A Branch Target Injection Mitigation. https://support.google.com/faqs/answer/7625886, 2018.

[12] Michael Schwarz, *et al.* Context: A generic approach for mitigating spectre. In NDSS '20. The Internet Society, 2020.

[13] Zhichao Hua, *et al.* EPTI: Efficient defence against meltdown attack for unpatched vms. In ATC '18, pages 255–266. USENIX, 2018.

[14] Mehmet Sinan Inci, *et al.* Cache attacks enable bulk key recovery on the cloud. In CHES '16, pages 368–388. Springer, 2016.

[15] paboldin. meltdown-exploit. https://github.com/paboldin/meltdown-exploit, 2018.

[16] Eugnis. spectre-attack. https://github.com/Eugnis/spectre-attack, 2018.

[17] google. rowhammer-test. https://github.com/google/rowhammer-test, 2014.

[18] Daniel Gruss, *et al.* Flush+flush: A fast and stealthy cache attack. In DIMVA '16. Springer, 2016.