WaitWatcher & WaitGuard: Detecting Flush-Based Cache Side-Channels through Spurious Wakeups

Lukas Lamster, Fabian Rauscher, Martin Unterguggenberger, and Stefan Mangard

Institute of Information Security (ISEC), Graz University of Technology
{firstname.lastname}@tugraz.at

Abstract. Flush+Reload and Flush+Flush attacks target CPU caches and allow malicious actors to leak confidential data across different CPU cores. Typically, detection mechanisms against such attacks leverage hardware performance counters to observe architectural and microarchitectural events. However, recent research has shown that state-of-the-art security monitors can effectively be bypassed by camouflaged Flush+Reload attacks. Thus, flush-based cache side-channel attacks are still a significant threat to system security.

In this work, we present *WaitGuard*, a novel detection technique with a >99.9 % detection rate based on the userspace monitor and wait instructions. Our framework automatically profiles internal CPU interactions of userspace monitor/waits with other unprivileged instructions. We use WaitWatcher to analyze 7 different server and desktop-class x86 CPUs from Intel and AMD. In our analysis, we uncover 5 spurious wakeup triggers and 18 user-mode instructions that completely bypass the wakeup mechanisms. Based on our analysis, we develop *WaitGuard*, a novel detection mechanism that repurposes the recently introduced userspace monitor and wait instructions to detect flush-based cache side-channel attacks on modern x86 hardware. We implement WaitGuard as a drop-in security monitor that reliably detects Flush+Reload and Flush+Flush attacks with a detection rate of >99.9 %, even when introducing heavy system noise. Moreover, we find that WaitGuard also detects the previously invisible camouflaged Flush+Reload attacks. Finally, we demonstrate the real-world applicability of WaitGuard by showing its effectiveness in detecting Flush+Reload attacks on the OpenSSL AES T-table implementation.

Keywords: Side Channels · Cache Attacks · Userspace monitor/wait.

This preprint has not undergone any post-submission improvements or corrections. The Version of Record will be published in ESORICS 2025.

1 Introduction

Computing systems require the strong isolation of shared system resources. For instance, modern cloud computing systems co-locate mutually distrusted tenants with code execution privileges on the same physical machine. While computer systems offer strong architectural isolation for memory resources, *i.e.*, through process isolation, side-channel attacks still allow a malicious actor to leak secret data. Concretely, side-channel attacks facilitate shared system resources (*e.g.*, the cache) to extract secret information from software running on shared hardware by observing measurable side effects caused by the victim's execution.

Cache side-channel attacks take advantage of the shared cache between attacker and victim on modern CPUs. Flush+Reload [34] and Flush+Flush [13], in particular, flush cache locations that contain shared data or code, *e.g.*, shared libraries. Subsequently, by timing a memory access or a second flush operation after executing the victim code, the attacker can learn whether the victim accessed the flushed location to infer secret information, such as cryptographic key material [13, 34]. As modern CPUs utilize caches that are shared or can be manipulated across cores, an attacker is not constrained to attacking victims on the same core, thus significantly increasing the attack surface through potential cross-core data leakage [13, 34].

Common detection techniques for flush-based side-channel attacks rely on hardware performance counters [3, 18], which measure low-level system events such as cache evictions or TLB misses [3]. While these detection mechanisms were assumed to be sufficient to detect Flush+Reload and Flush+Flush attacks, recent research by Kosasih et al. [18] demonstrated that it is possible to circumvent performance counter-based detection completely. They present a camouflaged Flush+Reload attack that bypasses state-of-the-art detection mechanisms, successfully leaking secret information from a victim.

In this work, we present *WaitGuard*, a novel approach for detecting flush-based side-channel attacks by repurposing the userspace monitor and wait instructions on commodity x86 Intel and AMD machines. *WaitGuard* is capable of detecting Flush+Reload and Flush+Flush attacks with a detection rate of >99.9 %. Furthermore, we introduce *WaitWatcher*, a framework for systematically analyzing the behavior of userspace monitor/wait instructions.

With WaitWatcher, we provide a framework that automatically analyzes the recently introduced userspace monitor and wait instructions, originally intended to reduce the CPU energy consumption of busy waits. We perform a comprehensive study, analyzing 7 different server and desktop-class x86 CPUs from Intel and AMD using WaitWatcher. Our study uncovers five different spurious wakeup triggers and 18 instructions that completely bypass the wakeup mechanisms.

With WaitGuard, we present a drop-in security monitor for detecting flush-based cache side-channel attacks by repurposing the unprivileged userspace monitor and wait instructions. While these instructions are intended for detecting writes to monitored addresses, we find that flushing the monitored addresses also causes the wait to abort. Combining this address monitoring with an access latency measurement after each wakeup allows us to detect flush-based at-

tacks with an accuracy of >99.9 %, even under heavy system load. Furthermore, WaitGuard is capable of detecting camouflaged Flush+Reload attacks, which can bypass traditional performance counter-based detection mechanisms [18].

We introduce two variants of WaitGuard that allow for flexible use of our detection mechanism. First, we propose a self-contained variant in which a process monitors its own memory using a dedicated monitor thread. This variant can be implemented without involving the host operating system (OS) and allows the process to determine whether it wants to continue execution in the case of a potential attack. Second, we detail a delegated variant, which hands the monitoring task to the operating system. The OS uses a dedicated thread to protect the data of the victim process.

We implement the self-contained variant of WaitGuard and evaluate its effectiveness. Using three different Flush+Reload attacks, we find that WaitGuard can reliably detect side-channel attacks with a detection rate of >99.9 %. Furthermore, we find that the rate of false positives is vanishingly low, thus underlining the feasibility of our approach.

Contributions. In this work, we make the following key contributions:

- **WaitWatcher.** We present WaitWatcher, an open-source[1] framework that automatically analyzes the behavior of the recently introduced x86 userspace monitor and wait instructions.
- **Insights on Monitor/Wait Instructions.** We provide a comprehensive study with overall 7 Intel and AMD CPUs with different microarchitectures, showcasing undocumented CPU behavior, such as spurious wakeup triggers and unprivileged instructions that can bypass the wakeup mechanisms.
- **WaitGuard.** We present a novel mechanism that reliably detects cache-based side-channel attacks by repurposing the userspace monitor/wait instructions available on commodity x86 hardware.
- **Proof-of-Concept and Evaluation.** We evaluate WaitGuard, showcasing the detection of flush-based side-channel attacks with a high accuracy of >99.9 %, including camouflaged Flush+Reload attacks.

Outline. The remainder of this work is organized as follows. Section 2 provides the background of this work and discusses related work. Section 3 introduces the WaitWatcher framework and presents the analysis results for 7 Intel and AMD machines. In Section 4 we introduce WaitGuard, a novel drop-in security monitor that detects flush-based cache side-channel attacks. Section 5 discusses the implementation and evaluation of WaitGuard. Section 6 concludes this work.

2 Background and Related Work

This section provides the required background on cache side-channels, flush-based side-channel attacks, and proposed detection mechanisms. Furthermore, we discuss the recently introduced userspace monitor and wait instructions that provide the foundation of our approach.

[1] The WaitWatcher Framework

2.1 Cache Side-Channels

The cache hierarchy of modern CPUs consists of multiple *levels*, which are categorized depending on their proximity to the CPU cores. The Level 1 (L1) and Level 2 (L2) caches are usually small and deeply integrated in the core to allow for minimal access latencies. While the L1 and L2 are not shared across physical cores, with hyperthreading they can be shared across logical cores which share the same execution engine. The last-level cache (LLC) is furthest away from the cores, provides more storage space, and is typically shared across multiple physical cores. Cache side-channels exploit timing variations that depend on whether data is cached or not, *e.g.*, a load access that results in a cache hit is significantly faster than a cache miss that results in a main memory access (*i.e.*, the DRAM) [34]. Through measuring timing differences, an attacker can determine whether a value was recently accessed (*i.e.*, cached) or not.

2.2 Flush-based Side-Channel Attacks

Flush-based cache side-channel attacks are based on flushing an address from the cache, waiting for victim activity, and measuring whether the victim accessed the address. This attack relies on the availability of shared memory between the victim and the attacker to flush the victim address. While these attacks are typically performed on shared libraries, they are also possible on arbitrary memory and cross-VM due to page deduplication [27, 31]. Depending on how the attacker determines whether the victim address was accessed, we distinguish between Flush+Reload and Flush+Flush attacks [13, 34].

Flush+Reload. In 2014, Yuval et al. [34] introduced a cache side-channel attack dubbed Flush+Reload. They use the `clflush` instruction, which evicts victim data from all cache levels, to build a cross-core attack. The basic principle of Flush+Reload is to evict a shared address using `clflush`, wait for or trigger victim execution, and measure the access latency for the evicted address. If the access is fast (cache hit) the victim accessed the memory location. If the access is slow (cache miss) the victim did not accessed the memory location.

In their proof-of-concept attack, they use Flush+Reload to leak the RSA private key of GnuPG [1]. The targeted program executes different code paths depending on the secret key bits. Thus, the execution behavior measured with Flush+Reload allows the attacker to reconstruct the key bits.

Flush+Flush. Gruss et al. [13] refined Flush+Reload by timing the `clflush` instruction instead of a separate memory access and dubbed this attack Flush+Flush. Flush+Flush is based on latency variations of the flush instruction that depend on whether the flushed address is present in the cache. `clflush` has to perform significantly more work when a cache line is present as it has to evict it from all cache levels. Thus, measuring the latency of `clflush` allows the attacker to determine the cache state of the victim address. The attack consists of a loop that executes `clflush` and measures the latency. Gruss et al. [13] also

implement covert channels using Flush+Reload and Flush+Flush. Their evaluation showcases that Flush+Flush achieves high transmission rates while staying largely undetected. Furthermore, Flush+Flush does not have a blind spot, unlike Flush+Reload, which can miss victim accesses if they occur between the attackers memory access and the flush which restores the cache state [29].

2.3 Side-Channel Attack Detection

Due to the threat of side-channel attacks, researchers proposed a multitude of possible solutions. One promising approach is to *detect* side-channel attacks using hardware performance counters (HPC) [4–6, 9, 10, 19, 21–23, 28, 33, 36]. HPCs monitor certain architectural and micro-architectural events occurring on a system. Many defenses use the values reported by HPCs to assess the current state of the system. Due to the distinct behavior of side-channel attacks, they either cause detectable anomalies in the performance counter behavior or follow certain signatures [3].

While HPCs were generally thought to allow for the accurate detection of cache side-channel attacks, issues arise with this approach. Recently, Kosasih et al. [18] introduced a camouflaged Flush+Reload attack that bypasses all considered detection classifiers. Furthermore, the set of available performance counters is hardware-specific, varies between CPU vendors, and also varies between CPU generations and families. It is, thus, challenging to provide a *generic* detection mechanisms that does not require tedious adaption for each specific hardware platform. Also, granting userspace programs access to low-level performance counters can facilitate further side-channel attacks, by introducing another potential attack surface [8, 12, 32].

2.4 Userspace Wait Instructions

Recent AMD and Intel processor generations introduced new instructions for reducing the CPU power consumption while waiting for certain events, such as write accesses. Both vendors provide dedicated userspace monitoring and waiting instructions [7, 16] that eliminate the need for busy waits by replacing them with efficient alternatives. For Intel CPUs, the new instructions are dubbed `umonitor` and `umwait`, while AMD denotes them as `monitorx` and `waitx`. They allow userspace code to set up a monitor for a certain memory granule and subsequently call the corresponding wait instruction to hint the CPU to enter a low-power state. Write accesses to the monitored address wake up the waiting thread. However, other events, such as non-maskable interrupts or a wait time exceeding an operating-system-defined limit may cause spurious wakeups. In the case of Intel CPUs, a flag indicates whether a wakeup was due to the operating system timeout or any other source.

Recently, Zhang et al. [35] demonstrated that not only *architectural* write accesses to the monitored addresses act as wakeup triggers. They find that speculative write accesses also cause the waiting thread to exit its low-power state. Thus, they illustrate that undocumented, implementation-specific sources

can also act as a wakeup trigger for the novel wait instructions. In their work, Zhang et al. [35] use these spurious wakeups to improve the leakage achieved with speculative execution attacks and to perform website fingerprinting attacks.

Terminology. In the remainder of this work, we use the term `monitor` and `wait` to refer to the userspace monitoring instructions of both Intel and AMD. In cases where it is important to distinguish between vendor-specific behavior, we use the corresponding instruction names, *i.e.*, `umonitor`/`umwait` for Intel and `monitorx`/`waitx` for AMD.

3 WaitWatcher Framework

Previous research explored specific parts of the new `wait` instructions. Despite this, there is no openly available framework for analyzing how `monitor` and `wait` instructions are influenced by other instructions on different CPUs. Furthermore, there is no comprehensive study on wakeup triggers and the behavior of the `monitor`/`wait` instructions across different CPU architectures. While the corresponding documentations describe modification of the monitored memory location as a wakeup trigger [7,16], we will show that this is not always the case. Additionally, there might exist further undocumented wakeup triggers. As the x86 instruction set architecture consists of thousands of instructions, a manual analysis even on a single platform would require an extensive amount of time. Thus, performing a manual analysis on multiple platforms and microarchitectures is infeasible. Therefore, an automated approach for both Intel and AMD systems is needed. To solve this issue, we develop the WaitWatcher framework. Our WaitWatcher framework implements an automated analysis for finding interactions between `monitor/wait` instructions and all other instructions available on the system under test. We base our implementation on the open-source Minefield framework developed by Kogler et al. [17]. Their framework is designed to test instructions for their susceptability to undervolting fault attacks. However, we find that Minefields' instruction gathering and sample generation is a suitable starting point for WaitWatcher.

3.1 Framework Design and Analysis Approach

The analysis performed by our framework can be divided into three distinct steps. We denote these steps as *Instruction Gathering*, *Sample Generation*, and *Interaction Analysis*. After executing these three steps, the framework produces a list of instructions and how they interact with the `monitor`/`wait` instructions.

Instruction Gathering. First, we gather a list of all instructions that are available on the target ISA, in this case x86. We obtain this data by using publicly available sources provided by Abel et al. [2]. The data is provided as an XML file that contains detailed information about the available instructions as well as their source and destination operands. Furthermore, each instruction entry in the list

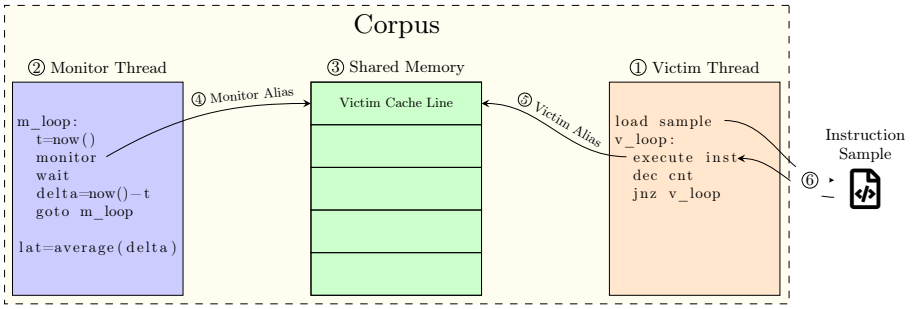

Fig. 1: An overview of the *Instruction Analysis* step of our framework. The monitor thread monitors a victim cache line and measures the time spent waiting in a loop. The victim thread executes the instruction sample on the victim cache line. The average waiting duration determines whether an instruction affects the wakeup behavior.

contains information on the required privilege level and if the instruction is part of an ISA extension. We consider this list as the ground truth for all further steps of the analysis.

Sample Generation. Next, we generate a distinct dynamically loadable file for each of the available instructions. During this step, we filter out instructions that require a higher privilege level than CPL 3 (*i.e.*, userspace mode). This is due to the fact that we are only interested in analyzing the interaction of userspace instructions and `wait` instructions. Based on the source and destination operands, our framework instruments the instructions with a prologue and epilogue. Both of them are required to set up registers and memory locations such that they are contextually meaningful for the analyzed instruction. This is especially vital for instructions that interpret register values as addresses, as a faulty setup will cause crashes. Due to the operand-dependent instrumentation, we consider operations with different operand types as distinct instructions. Thus, a `mov` from register to memory and a `mov` from register to register are compiled into two separate files. Note that this step also compiles instructions that may not be supported by the system under test. Reducing the set of compiled instructions to potentially executable instructions is a possible optimization for speeding up the sample generation step. However, the sample generation is only executed once and, even on our slowest system, takes only minutes of execution time due to the parallel nature of our implementation. During this step, the framework excludes instructions that are not supported by the currently used compiler version. Instructions that are affected by this exclusion belong to currently unimplemented extensions. Furthermore, we exclude instructions that perform control-flow transfers as we only focus on data-oriented instructions.

Interaction Analysis. In the third step, we perform the actual analysis of the interaction between instructions and the `monitor`/`wait` instructions. For that, the

previously compiled and instrumented samples are dynamically linked by a test program denoted as the *corpus*. The corpus handles the logic to detect an interaction between the currently analyzed instruction sample and the `monitor`/`wait` instructions. Figure 1 illustrates how the corpus and the samples interact to detect potential wakeup triggers. The corpus creates ① a *victim thread* and ② a *monitor thread*. It then sets up a shared memory region ③ and two aliases (④,⑤) to the region. As the instructions we want to analyze are compiled to a shared object, they need to be loaded ⑥ by the victim thread. After the initial setup phase, the monitoring thread repeatedly calls the `monitor` and `wait` instructions and measures the time between the invocation of the `wait` instruction and the subsequent wakeup. When invoking the `monitor` instruction, the monitor thread uses ④ the monitor alias for its target address. Meanwhile, the victim thread executes the currently loaded sample in a tight loop. The samples are instrumented such that they use ⑤ the victim alias as their destination operand. The victim thread halts once a configured number of iterations is reached. Once the victim thread is finished, the monitor thread averages the latency values measured during the monitoring phase.

Our framework uses the average wait duration to distinguish between the samples that interact with `wait` (*i.e.*, cause wakeups) and those that do not cause wakeups. As the latency is significantly lower on wakeups, ambiguities in the form of false positives and false negatives are highly unlikely. Combining the average latency with a check that tests whether the monitored location was actually modified after invoking the sample loop allows us to detect two types of atypical behavior. A high average wakeup latency combined with a memory modification indicates that the analyzed instruction does not act as a wakeup trigger and bypasses the wakeup logic. Contrarily, a low average wakeup latency and an unmodified victim memory imply that the analyzed instruction is a wakeup trigger without actually writing to memory. We denote these instructions as *spurious* triggers. Overall, we classify instructions into four categories based on the averaged wait times and the state of the data after sample invocation. Besides the atypical cases listed above, the other two categories contain the instructions that behave according to their expected wakeup behavior.

At the end of the interaction analysis, our framework generates a report for all instruction samples that were compiled during sample generation. For each sample, we log the exact operation and operands, the averaged wakeup latency, and the category in which the operation was classified. Furthermore, the report contains information on which instructions are unimplemented on the current system and which samples encountered runtime errors. We execute the analysis in two configurations where the monitor and victim thread are either sibling threads or pinned to unrelated cores.

Transient Execution. While the main focus of WaitWatcher lies on regular instruction execution, we also implement the analysis of transiently executed instructions. When using the transient execution mode, the instruction sample is modified such that the analyzed instruction is never executed architecturally. Thus, when measuring positive interaction with the analyzed instruction, we

Table 1: The analysis results of our framework for architectural execution. Depending on the microarchitecture, multiple spurious wakeup triggers are identified. We also find that some instructions are able to modify data without causing a wakeup. Results reported for AMD systems assume sibling threads.

CPU Generation	Analyzed Instructions	Data Mod.		No Data Mod.	
		👁	🙈	👁	🙈
🖥 Zen 2	2996	385	14	**4**	2593
🖥 Zen 3	3005	399	0	**4**	2602
🗄 Zen 4c	12 880	655	6	**4**	12 215
🖥 Alder Lake	3044	389	12	**3**	2640
🖥 Arrow Lake	3124	418	12	**0**	2692
🗄 Sapphire Rapids	15 039	647	18/0†	**5**	14 369
🗄 Emerald Rapids	15 040	647	18/0†	**5**	14 370

👁 Wakeup Trigger 🙈 No Wakeup Trigger
🗄 Server CPU 🖥 Desktop CPU † Arbitrary Thread/Sibling Thread

can conclude that transient effects also act as wakeup triggers. Note that the transient execution version of WaitWatcher may require manual optimizations depending on the analyzed system due to differences in the underlying hardware.

3.2 Analysis Results

We find that our analysis approach and instruction instrumentation work reliably for most of the instructions implemented on our tested systems. A small subset of all instructions triggers runtime errors. This is usually the case for instructions where the operands are contextually bound to a range that is not documented in the ground truth used for instrumenting the instructions. As our framework logs crashing instructions, they can be manually analyzed if required.

We execute our framework on multiple microarchitectures from both Intel and AMD. In our analysis, we target an overall of seven systems containing both server-grade and desktop CPUs. Table 1 lists the quantitative results of our automated analysis. As in the sample generation step, we distinguish between invocations of the same operation using different source and destination types when counting instructions. The main focus of our analysis lies on instructions that either modify data without causing a wakeup or, conversely, do not modify data but act as wakeup triggers. We find that all considered systems except for Arrow Lake CPUs have at least three spurious wakeup triggers. Furthermore, on some of the analyzed systems, we find instructions that can completely bypass the monitoring logic and modify memory without causing wakeups. A detailed description of the results of our analysis is given below.

Spurious Wakeup Triggers. Our framework allows us to reproduce the findings regarding undocumented wakeup triggers provided by previous work [35].

However, we also observe previously unreported additional triggers. Specifically, we find that `clflush` and `clflushopt` also cause wakeups on Intel Sapphire Rapids and Emerald Rapids. Previous research only analyzed Intel desktop CPUs and reported that `clflush` only causes wakeups for AMDs `mwaitx` instructions [35]. Furthermore, we find that on Emerald Rapids and Sapphire Rapids, both `prefetch` and `prefetchwt1` cause wakeups in addition to the previously reported `prefetchw` instruction. Thus, for Emerald Rapids and Sapphire Rapids we find an overall of five spurious wakeup triggers. Our measurements indicate that on Intel server-grade CPUs, spurious triggers reliably cause a wakeup *independently* of the core on which the monitor and victim thread are running. Furthermore, we find that Intel's desktop CPUs do not experience the same spurious wakeup triggers as their larger counterparts. On Alder Lake, we find that `prefetch`, `prefetchw`, and `prefetchwt1` cause spurious wakeups, while on Arrow Lake, no spurious triggers are detected.

On AMD systems, `clflushopt`, `clflush`, `clwb`, and `prefetchw` cause spurious wakeups. For AMD machines, we find no difference between server-grade and desktop CPUs.

Undetected Data Modification. Besides spurious wakeup triggers, we also identify instructions that bypass the monitoring functionality. We find that `movnt*` and `vmovnt*` instructions allow for data modification without triggering the wakeup of the `wait` instructions on all considered Intel machines. We observe this behavior for the regular and `evex` encoded instructions. However, we also find that both `vmovnt*` and `movnt*` instructions *do* trigger wakeups when using the `ZMM` register or running the victim and monitor threads on the same core, *i.e.*, as sibling threads. On Zen 4c, we find that `vmovnt*` instructions encoded using the `evex` prefix modify data without triggering a wakeup of `mwaitx`.

Non-Conformant Behavior. Surprisingly, we find that on all analyzed AMD systems it is necessary to run the monitoring thread and the victim thread as sibling threads. When pinning one of the two threads to a different core, no instruction will reliably trigger a wakeup. This directly contradicts the description of the `mwaitx` instruction given in the Architecture Programmer's Manual, which states that *a store from another processor* shall cause a wakeup of the instruction [7].

Contrarily, on the analyzed Intel machines, we find that the wakeup behavior of *most* instructions does not depend on the cores on which the monitor and victim thread are executed. Only the non-temporal move instructions discussed above experience different behavior when executed on non-sibling threads.

Wakeups on Transient Execution. We find that transiently executed instructions also cause spurious wakeups, thus confirming the results of Zhang et al. [35]. Interestingly, we find that not only write accesses trigger wakeups. Instructions such as `prefetchw`, which do not actually perform a write, may also act as transient spurious wakeup triggers. However, we find that transient invocations of `clflush` do *not* trigger spurious wakeups. Note that while some data-modifying instruction instructions cause wakeups when executed transiently, some fail to do

so. We assume that this behavior depends on the latency of the instruction and the size of the speculation window. Instructions that finish within the speculation window cause wakeups while those that do not finish do not trigger wakeups.

4 Detecting Flush-based Side-Channel Attacks

In this section, we show how the insights gained in our analysis allow us to implement a performance-counter agnostic detection method for flush-based side-channel attacks on Intel CPUs.

First, we discuss the general idea of our new side-channel detection mechanism and introduce WaitGuard. WaitGuard is the first side-channel detection using userspace `wait` instructions to detect Flush+Flush and Flush+Reload attacks. We elaborate on the monitoring approach and show how WaitGuard can detect malicious flush operations on monitored data on Intel CPUs. We then propose two variants of WaitGuard and elaborate on their respective use cases. Theoretically, the presented approach also applies to AMD systems given that the behavior of `mwaitx` would reflect the description given in the architecture programmer's manual.

Attacker Model. For our detection approach, we assume a Flush+Reload or Flush+Flush attacker that aims to extract secret information from a victim process. The attacker can perform `clflush` operations on cache lines that the attacker process shares with the victim process. Without loss of generality, we assume that victim cache lines contain read-only data. We assume that the attacker has full knowledge of the victim process. Furthermore, we assume that the attacker process and the victim process are completely synchronized. Thus, the attacker knows precisely which victim operations are currently being executed. We do not consider attacks that use a contention-based approach to evict victim cache lines. Attacks like Prime+Probe or Evict+Reload [14, 26] were relatively straightforward to mount across cores on older machines using fully inclusive L3 caches. However, newer CPU generations, such as the ones supporting userspace wait instructions, utilize L3 caches that are not fully inclusive. Evictions from the shared L3 do not cause evictions from the private L1 and L2 caches of the victim, making the eviction step of both Prime+Probe and Evict+Reload no longer possible across cores. Additionally, accesses that can be served by the L1 or L2 do not result in the cache line being written into the L3. Therefore, the probing step of Prime+Probe on the L3 can not detect memory accesses that can be served from these private caches [30].

4.1 Design of WaitGuard

Our analysis results show that `clflush` acts as a wakeup trigger for both Intel and AMD CPUs. Hence, we propose to use the `monitor`/`wait` instructions as a detection mechanism for flush-based side-channel attacks. Given that we can monitor a cache line using `monitor` and `wait`, a `clflush` instruction that targets

this cache line will cause a wakeup without a write access. As `clflush` constitutes the core operation of Flush+Reload and Flush+Flush attacks, it should thus be possible to detect such attacks by observing the behavior of `wait` when monitoring a currently targeted cache line. Based on this general idea, we develop WaitGuard. WaitGuard is the first detection mechanism for flush-based side-channels that leverages the novel `monitor` and `wait` instructions. While doing so, our approach does not require access to any hardware performance counters besides the time stamp counter (TSC).

Monitoring Logic. WaitGuard implements a *monitoring logic* which aims to detect all flushes to a monitored cache line. The core of our monitoring logic is a tight loop that calls `monitor` followed by a `wait` instruction. We measure the time passed between executing the `wait` instruction and the subsequent wakeup. This wait time is the first indicator on whether the cache line is being targeted by a flush-based attack. The monitoring loop is, in this aspect, equivalent to the loop used in our Interaction Analysis. However, we cannot purely rely on this metric to detect flush-based attacks. There are multiple sources that can wake up a waiting thread. Additionally to a write access to the monitored cache line, interrupts and an OS-defined timeout can cause spurious wakeups [35]. Thus, using a single measurement causes too many false positives as spurious wakeups may be interpreted as write accesses. To avoid this issue, we use an additional metric to improve the results of our detection: after the wakeup, we measure the access latency to the monitored cache line. In the case of Flush+Reload or Flush+Flush attacks, the access latency is high as the data was flushed and must be fetched again from memory. Given that `monitor` performs a 1-byte fetch operation, the monitored cache line is always cached [7,16]. Thus, accessing the cache line after a wakeup due to a timer interrupt or any other trigger will result in a fast access. We combine the access latency measurement with a rolling average over the latest wakeup latencies to improve our detection reliability. This allows us to detect anomalies due to the significant increase of flush operations during Flush+Reload and Flush+Flush attacks.

4.2 WaitGuard Variants

We propose two variants of how WaitGuard can be implemented. When elaborating on the variants, we use the term *victim* process or cache line to refer to the potential target of a side-channel attack. Figure 2 schematically depicts both variants of WaitGuard.

Drop-In Variant. First, we propose a self-contained variant of WaitGuard, which is illustrated in Figure 2 (a). This variant requires the victim process to be able to create threads. The victim creates a list of cache lines that require monitoring. This list can either be generated manually or by using a profiling approach such as the one proposed by Gruss et al. [14]. For each of these cache lines the victim spawns one thread that monitors the corresponding cache line ①. Subsequently, the monitoring thread calls `monitor` followed by a `wait` on the to-be-monitored

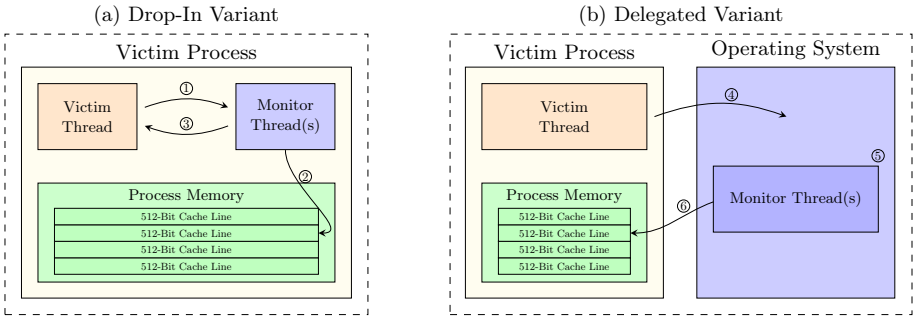

Fig. 2: Two variants of WaitGuard. The drop-in variant can be deployed by any process and does not require OS support. The delegated variant spawns a dedicated monitoring thread which is managed by the OS.

cache line ②. After waking up from the wait, the monitor thread checks how long it was waiting and the access latency when reading from the monitored cache line. Depending on the measured values, the monitor thread decides whether the cache line was flushed. This approach has the advantage that it can be implemented as a drop-in side-channel detection for userspace processes on off-the-shelf Intel CPUs. Upon detection of `clflush` operations, the monitoring thread informs the victim ③. Thus, the process can decide how to react and whether it is safe to proceed execution. Note, however, that one hardware thread can only monitor one cache line at a time. Thus, monitoring multiple cache lines must either be achieved by monitoring one cache line after the other or by spawning additional monitoring threads.

Delegated Variant. The delegated version of WaitGuard relies on the operating system, as shown in Figure 2 (b). Like in the drop-in variant, the victim process creates a list of cache lines that require monitoring. This list is passed to the operating system ④ which, in turn, creates one or multiple monitoring threads ⑤ that monitor the given victim cache lines ⑥. Contrary to the drop-in variant, the OS can take measures to isolate the victim from potential attackers (*e.g.*, by providing a non-shared mapping of the shared library under attack). However, the delegated variant requires the operating system to be aware of the side-channel detection approach. While it is possible to modify the Linux kernel, using a kernel module would allow for easier integration in existing systems.

5 WaitGuard Implementation and Evaluation

In this section, we discuss our proof-of-concept implementation and evaluate the detection rate of WaitGuard. We evaluate our implementation on a commodity Intel system and achieve a detection rate of >99.9 % for basic Flush+Reload attacks. Furthermore, we show how WaitGuard can be configured to detect stealthy attacks such as the ones proposed by Kosasih et al. [18].

5.1 Implementation

Our proof-of-concept implementation consists of the drop-in variant of WaitGuard and multiple victim processes. In our implementation, multiple cache lines are monitored by spawning a dedicated monitor thread for each cache line. Monitor threads are pinned to sibling threads. Our PoC uses the following logic to classify whether an attack is occurring or not. We implement a moving average over the wait durations to smoothen the values in the presence of disturbances due to interrupts and other spurious wakeups. After each wakeup, we additionally measure the access latency to the monitored data. Combining both values, we decide whether we were woken due to a `clflush` and flag such cases as potential attacks. If the current average wait duration is below a *critical wait duration* and the memory access latency is above a pre-defined threshold, a detection is logged. Both the critical wait duration and the access latency threshold are measured manually and are expected to vary across systems. Note that the requirement for manual measurement is a limitation of our PoC and the calibration can be automated. The PoC logs the timestamp at which the attack was detected. Our implementation is parameterizable with regard to the moving average window size and the critical wait duration.

To show the importance of combining our wait-based approach with memory access timing measurements, we also implement a variant that does *not* use `monitor` and `wait` instructions. Instead, it only measures the cache access latency to determine whether an attack occurred. We refer to this variant as the *measure-only* (MO) variant.

Victim Processes. We implement multiple victims, denoted as $\mathcal{V}_1$ to $\mathcal{V}_3$, to evaluate WaitGuard for Flush+Reload attacks under different attack scenarios. At their core, all victims implement an AES encryption using OpenSSL 3.3.3 [25] and are based on an open-source implementation of a Flush+Reload attack [24]. The cache lines targeted by the attacker hold the T-tables, which are a popular target for side-channel attacks [11,13,15,20]. We compile OpenSSL without hardware acceleration to enforce the usage of software-based T-tables. All victims perform 300 000 encryptions. Depending on the victim, the attacker performs one or multiple measurements for each encryption. We combine the attacker and the victim in the same process to ensure synchronization and a minimal time difference between flushing target cache lines, performing the OpenSSL AES encryption, and measuring whether the relevant T-table entries were accessed.

The attacker in $\mathcal{V}_1$ leaks individual key bytes by attacking one T-table entry at a time. $\mathcal{V}_1$ first performs a fixed number of encryptions where the attacker exclusively targets the first T-table cache line. Next, the attacker targets the second T-table entry. The attacker proceeds like this until all key bytes are leaked. Victim $\mathcal{V}_2$ implements a more efficient attacker that first flushes all targeted cache lines. Then, the victim executes its encryption operation. Subsequently, the attacker measures the access latency for all previously flushed cache lines. For $\mathcal{V}_3$, we implement a stealthy variant of $\mathcal{V}_2$. Each encryption iteration has a

0.2% chance of actually performing the Flush+Reload attack. In all other cases, the victim performs the encryption without being attacked.

5.2 Evaluation Setup

We use the previously described victims to gauge how well WaitGuard can detect the implemented attacks. Each victim invocation executes for a set amount of time and logs the begin and end timestamp of each Flush+Reload attack, thus generating a list of time frames in which an attack occurred. Similarly, WaitGuard logs the time stamps at which it detects an attack. We use the system-wide timestamp counter for these measurements. Once the victim finishes, we compare the detection timestamps with the time frames at which attacks were performed. We consider an attack as detected whenever the victim logs *at least one detection timestamp* during the attack time frame. All victims described above are implemented such that we introduce an artificial delay of several milliseconds after each attack run. We use this delay to create a temporal separation between attack time frames to better catch spurious wakeups and avoid a bias toward overly optimistic detection results.

We use the `cpu` stressor of `stress-ng` to evaluate our PoC under varying system loads by generating additional noise. The imposed load increases from 0% to 90% in 10% steps. During our evaluation, we do *not* isolate the cores that are executing the victim and the monitor code.

As mentioned in Section 4.1, our PoC uses a windowing approach to reduce noise. We evaluate our implementation for different window sizes to measure their impact on the detection rate. The tested window sizes range from two samples to 256 samples. We perform our evaluation on a commodity Intel Xeon Gold 6530 CPU with 32 cores and 64 threads. Our system is equipped with 512GB DDR5 DRAM and runs the Linux kernel version 6.8.0.

5.3 Detection Rate and False Positive Rate

The left column of Figure 3 illustrates the detection rates and false positives rates for a 16-sample window PoC. The detection rate is computed as the ratio between the number of actually performed attacks and the number of mounted attacks. For $\mathcal{V}_1$ and $\mathcal{V}_2$ we achieve a >99.9 % detection rate. However, we can not detected stealthy attacks like the one implemented in $\mathcal{V}_3$ due to the averaging approach. As the stealthy attack does not flush in every iteration, the computed average waiting time will only drop slightly when seeing a wakeup due to the attack. Thus, a 16-sample window is only suitable for detecting classical Flush+Reload attacks. We find that window sizes that exceed 16 samples do not offer any additional benefits.

Conversely, we find that when using a smaller window size of just two samples we can detect >99.9 % of stealthy attacks, as depicted in the right column of Figure 3. The short sleep duration that is observed when the attack is actually performed is enough to bias the average such that it falls below the detection threshold. Thus, attacks like the one implemented by $\mathcal{V}_3$ can also be detected

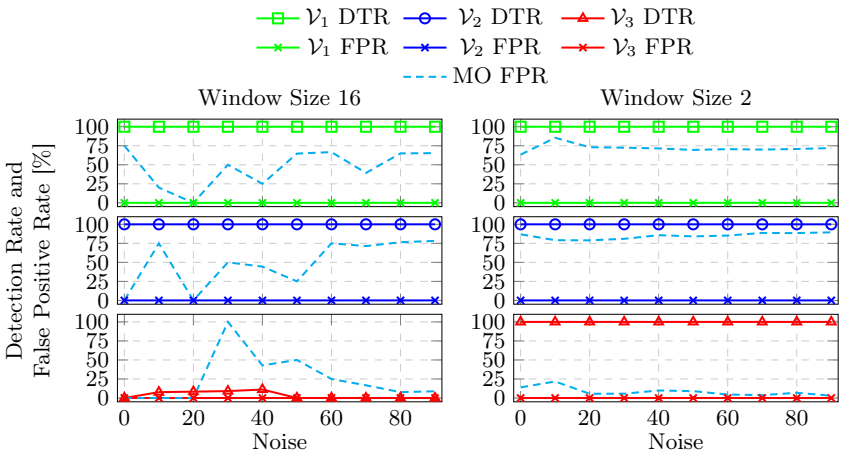

Fig. 3: The detection rate (DTR) and false positive rate (FPR) for WaitGuard using a 16-sample and a 2-sample window. While $\mathcal{V}_1$ and $\mathcal{V}_2$ are detected in both cases, $\mathcal{V}_3$ can only be detected by the 2-sample window configuration. The Measure-Only (MO) false positive rate is infeasibly high.

using WaitGuard. Furthermore, we find that decreasing the window size does not increase the false positive rate. As our PoC only flags an attack if both the wakeup latency and the subsequent memory access latency are low, false positives are rare. For all tested configurations we find that the rate of false positives is vanishingly low. The measured false positive rates are well below the 1% mark, regardless of the imposed system noise. We find that a measure-only variant (MO) is infeasible. While all attacks are flagged as detected, the amount of false positives is overwhelming, as depicted by the dashed lines in Figure 3. We observe large false positive rates, reaching up to 100%, across all noise levels. We find that the high false-positive rate is due to the monitor claiming to see a large number of attacks during the window between the attack rounds. Even *without* an active attacker, the measure-only variant logs, on average, 15.6 attacks per second. Contrarily, when using `wait` and `monitor`, we see no such behavior.

Scheduled Monitoring We furthermore investigate whether it is feasible to monitor multiple cache lines using a single thread. For that, we extend the monitor implementation such that each thread holds a list of monitored cache line addresses. All measured and computed values such as the averaged wakeup durations are unique for each monitored cache line. Thus, each monitored cache line is handled independently from all other cache lines. We implement a pseudo-random scheduling and a cache-line focused scheduling. When using the pseudo-random scheduling, the monitor thread picks a cache line at random and installs the monitor for the address at the beginning of each monitoring iteration. After measuring the wakeup time the monitor thread picks the next cache line by

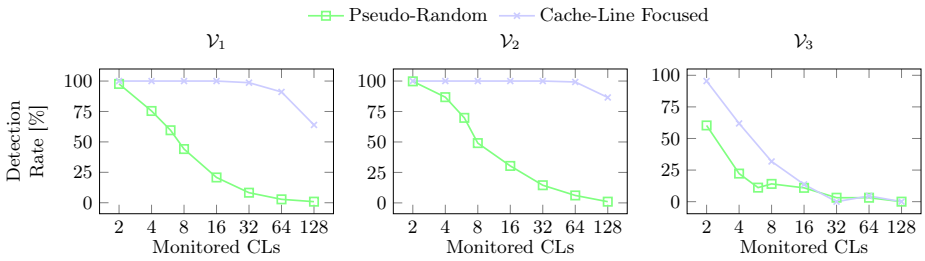

Fig. 4: The detection rates when monitoring multiple cache lines using a single thread. The detection rate drops steadily with the increasing number of monitored cache lines. When focussing on an attacked cache line, the detection rate remains high, even for multiple cache lines.

randomly selecting one from the given list. The cache-line focused scheduling initially operates equivalently. However, as soon as a cache line that is likely under attack is identified, the scheduling algorithm prioritizes this cache line.

Figure 4 illustrates the detection rates for both scheduling approaches. For the pseudo-random scheduling, the detection rate declines with an increasing number of cache lines being monitored by a single thread. The cache-line focused scheduling performs significantly better, especially for $\mathcal{V}_1$ and $\mathcal{V}_2$ victims. Note that all of the the given curves were measured using a 2-sample window size as this configuration performed best in the previous experiments.

5.4 Wakeup Distribution

Besides the detection rate, we also determine the empirical distribution of wakeup times within the attack time frames. The distributions are illustrated in Figure 5. For $\mathcal{V}_1$ and $\mathcal{V}_2$ most of the wakeups occur at the beginning of the attack. Note that $\mathcal{V}_1$ attacks the four victim cache lines sequentially. This is reflected in the distribution of $\mathcal{V}_1$, which shows an increase of wakeups around the points in time at which the attacker flushes the victim cache lines. For $\mathcal{V}_3$ we can observe that the wakeup timings are uniformly distributed. This is due to $\mathcal{V}_3$ distributing its attack phases across the whole attack window.

The distributions suggest that WaitGuard flags attacks early in the attack process or close to the `clflush` invocations. When computing the distributions for noise levels below 50% (not depicted in the figure), we observe even more pronounced peaks in the distribution. This suggests that additional system noise has a negative impact on the latency between the flush operation and the detection by WaitGuard.

6 Conclusion

In this work, we presented *WaitWatcher*, an automated analysis framework that determines how unprivileged instructions interact with the new `umonitor`/`umwait`

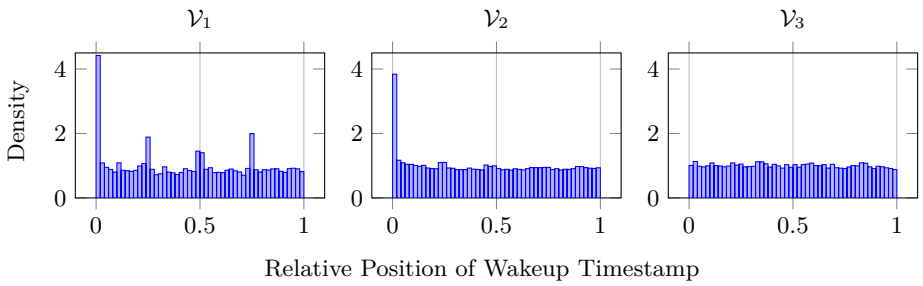

Fig. 5: The empirical distribution of wakeup timings in normalized attack windows for a 2-sample configuration.

and `monitorx`/`waitx` instructions. Contrarily to the work of Zhang et al. [35], WaitWatcher analyzes *all* executable userspace instructions with and without transient execution. We performed our analysis on 7 server and desktop-class x86 CPUs, uncovering 5 spurious wakeup triggers as well as 18 instructions that modify memory without causing wakeups. In addition, we found that, on AMD systems, wakeups are only correctly triggered when the monitoring thread and the thread performing the write access are sibling threads.

Based on the spurious wakeup triggers discovered in our analysis, we proposed *WaitGuard*, a novel mechanism that leverages the `umonitor`/`umwait` instructions on Intel x86 hardware to detect Flush+Reload and Flush+Flush attacks. Contrary to many existing detection approaches [3], WaitGuard operates without hardware performance counters and provides a drop-in solution that can be easily included in userspace software. We implemented a proof-of-concept of WaitGuard and evaluated its detection rate for different variants of Flush+Reload, including a stealthy attack. We find that WaitGuard achieves a detection rate of up to >99.9 % while reporting close to zero false positives. Furthermore, we find that WaitGuard is resistant to heavy system noise, *i.e.*, the detection accuracy stays constant. Hence, our approach is feasible for detecting flushes on vulnerable cache lines and, thus, Flush+Reload and Flush+Flush attacks. Besides a high detection rate, we find that WaitGuard can detect camouflaged attacks that are not detected by existing side-channel detection approaches [18].

Acknowledgements. We would like to thank the reviewers for their detailed feedback. Furthermore, we would like to thank David Schrammel for his support during the initial phase of this work. This research is supported by the European Research Council (ERC project FSSec 101076409) and the Austrian Research Promotion Agency (FFG) via the AWARE project (FFG grant number 891092). Additional funding was provided by generous gifts from Red Hat and Intel.

References

1. The GNU Privacy Guard. https://www.gnupg.org/, accessed: 2025-01-13
2. Abel, A., Reineke, J.: uops.info: Characterizing Latency, Throughput, and Port Usage of Instructions on Intel Microarchitectures. In: ASPLOS. pp. 673–686 (2019)
3. Akram, A., Mushtaq, M., Bhatti, M.K., Lapotre, V., Gogniat, G.: Meet the Sherlock Holmes' of Side Channel Leakage: A Survey of Cache SCA Detection Techniques. IEEE Access **8**, 70836–70860 (2020)
4. Alam, M., Bhattacharya, S., Mukhopadhyay, D., Bhattacharya, S.: Performance Counters to Rescue: A Machine Learning based safeguard against Micro-architectural Side-Channel-Attacks. IACR Cryptol. ePrint Arch. p. 564 (2017)
5. Allaf, Z., Adda, M., Gegov, A.E.: A Comparison Study on Flush+Reload and Prime+Probe Attacks on AES Using Machine Learning Approaches. In: Advances in Computational Intelligence Systems - Contributions Presented at the 17th UK Workshop on Computational Intelligence, September 6-8, 2017, Cardiff, UK. Advances in Intelligent Systems and Computing, vol. 650, pp. 203–213 (2017)
6. Allaf, Z., Adda, M., Gegov, A.E.: ConfMVM: A Hardware-Assisted Model to Confine Malicious VMs. In: 20th UKSim-AMSS International Conference on Computer Modelling and Simulation, UKSim 2018, Cambridge, United Kingdom, March 27-29, 2018. pp. 49–54 (2018)
7. AMD: AMD Architecture Programmer's Manual. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html (03 2024), 24594 rev. 3.36
8. Bhattacharya, S., Mukhopadhyay, D.: Utilizing Performance Counters for Compromising Public Key Ciphers. ACM Trans. Priv. Secur. **21**, 5:1–5:31 (2018)
9. Briongos, S., Irazoqui, G., Malagón, P., Eisenbarth, T.: CacheShield: Detecting Cache Attacks through Self-Observation. In: CODASPY. pp. 224–235 (2018)
10. Chiappetta, M., Savas, E., Yilmaz, C.: Real time detection of cache-based side-channel attacks using hardware performance counters. Appl. Soft Comput. **49**, 1162–1174 (2016)
11. Didier, G., Maurice, C.: Calibration Done Right: Noiseless Flush+Flush Attacks. In: DIMVA. LNCS, vol. 12756, pp. 278–298 (2021)
12. Gast, S., Weissteiner, H., Schröder, R.L., Gruss, D.: Counterseveillance: Performance-counter attacks on amd sev-snp. In: Network and Distributed System Security Symposium 2025: NDSS 2025 (2025)
13. Gruss, D., Maurice, C., Wagner, K., Mangard, S.: Flush+Flush: A Fast and Stealthy Cache Attack. In: DIMVA. LNCS, vol. 9721, pp. 279–299 (2016)
14. Gruss, D., Spreitzer, R., Mangard, S.: Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In: USENIX Security Symposium. pp. 897–912 (2015)
15. Gülmezoglu, B., Inci, M.S., Apecechea, G.I., Eisenbarth, T., Sunar, B.: A Faster and More Realistic Flush+Reload Attack on AES. In: COSADE. LNCS, vol. 9064, pp. 111–126 (2015)
16. Intel: Intel Software Developer's Manual. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html (12 2024)
17. Kogler, A., Gruss, D., Schwarz, M.: Minefield: A Software-only Protection for SGX Enclaves against DVFS Attacks. In: USENIX Security Symposium. pp. 4147–4164 (2022)
18. Kosasih, W., Feng, Y., Chuengsatiansup, C., Yarom, Y., Zhu, Z.: SoK: Can We Really Detect Cache Side-Channel Attacks by Monitoring Performance Counters? In:

Proceedings of the 19th ACM Asia Conference on Computer and Communications Security, ASIA CCS 2024, Singapore, July 1-5, 2024 (2024)
19. Kulah, Y., Dincer, B., Yilmaz, C., Savas, E.: SpyDetector: An approach for detecting side-channel attacks at runtime. Int. J. Inf. Sec. **18**, 393–422 (2019)
20. Lipp, M., Gruss, D., Spreitzer, R., Maurice, C., Mangard, S.: ARMageddon: Cache Attacks on Mobile Devices. In: USENIX Security Symposium. pp. 549–564 (2016)
21. Mushtaq, M., Akram, A., Bhatti, M.K., Chaudhry, M., Lapotre, V., Gogniat, G.: NIGHTs-WATCH: a cache-based side-channel intrusion detector using hardware performance counters. In: HASP. pp. 1:1–1:8 (2018)
22. Mushtaq, M., Akram, A., Bhatti, M.K., Chaudhry, M., Yousaf, M.M., Farooq, U., Lapotre, V., Gogniat, G.: Machine Learning For Security: The Case of Side-Channel Attack Detection at Run-time. In: 25th IEEE International Conference on Electronics, Circuits and Systems, ICECS 2018, Bordeaux, France, December 9-12, 2018. pp. 485–488 (2018)
23. Mushtaq, M., Bricq, J., Bhatti, M.K., Akram, A., Lapotre, V., Gogniat, G., Benoit, P.: WHISPER: A Tool for Run-Time Detection of Side-Channel Attacks. IEEE Access **8**, 83871–83900 (2020)
24. nepoche: https://github.com/nepoche/Flush-Reload. `https://github.com/nepoche/Flush-Reload` (2017), accessed: 2025-01-13
25. OpenSSL: OpenSSL 3.3.3. `https://github.com/openssl/openssl/tree/openssl-3.3.3` (2025), accessed: 2025-01-13
26. Osvik, D.A., Shamir, A., Tromer, E.: Cache Attacks and Countermeasures: The Case of AES. In: CT-RSA. LNCS, vol. 3860, pp. 1–20 (2006)
27. Philippe-Jankovic, D., Zia, T.A.: Breaking vm isolation-an in-depth look into the cross vm flush reload cache timing attack. International Journal of Computer Science and Network Security (IJCSNS) (2017)
28. Raj, A., Dharanipragada, J.: Keep the PokerFace on! Thwarting cache side channel attacks by memory bus monitoring and cache obfuscation. J. Cloud Comput. **6**, 28 (2017)
29. Rauscher, F., Fiedler, C., Kogler, A., Gruss, D.: A Systematic Evaluation of Novel and Existing Cache Side Channels. In: Network and Distributed System Security Symposium 2025: NDSS 2025 (2025)
30. Rauscher, F., Fiedler, C., Kogler, A., Gruss, D.: A Systematic Evaluation of Novel and Existing Cache Side Channels. In: NDSS (2025)
31. Suzaki, K., Iijima, K., Yagi, T., Artho, C.: Memory deduplication as a threat to the guest OS. In: EUROSEC. p. 1 (2011)
32. Uhsadel, L., Georges, A., Verbauwhede, I.: Exploiting Hardware Performance Counters. In: FDTC. pp. 59–67 (2008)
33. Wang, Z., Peng, S., Guo, X., Jiang, W.: Zero in and TimeFuzz: Detection and Mitigation of Cache Side-Channel Attacks. In: Innovative Security Solutions for Information Technology and Communications - 11th International Conference, SecITC 2018, Bucharest, Romania, November 8-9, 2018, Revised Selected Papers. LNCS, vol. 11359, pp. 410–424 (2018)
34. Yarom, Y., Falkner, K.: FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In: USENIX Security Symposium. pp. 719–732 (2014)
35. Zhang, R., Kim, T., Weber, D., Schwarz, M.: (M)WAIT for It: Bridging the Gap between Microarchitectural and Architectural Side Channels. In: USENIX Security Symposium. pp. 7267–7284 (2023)
36. Zhang, T., Zhang, Y., Lee, R.B.: CloudRadar: A Real-Time Side-Channel Attack Detection System in Clouds. In: RAID. LNCS, vol. 9854, pp. 118–140 (2016)