# Flush+Flush: A Fast and Stealthy Cache Attack

Daniel Gruss, Clémentine Maurice†, Klaus Wagner, and Stefan Mangard

Graz University of Technology, Austria

**Abstract.** Research on cache attacks has shown that CPU caches leak significant information. Proposed detection mechanisms assume that all cache attacks cause more cache hits and cache misses than benign applications and use hardware performance counters for detection.
In this article, we show that this assumption does not hold by developing a novel attack technique: the *Flush+Flush* attack. The *Flush+Flush* attack only relies on the execution time of the flush instruction, which depends on whether data is cached or not. *Flush+Flush* does not make any memory accesses, contrary to any other cache attack. Thus, it causes no cache misses at all and the number of cache hits is reduced to a minimum due to the constant cache flushes. Therefore, *Flush+Flush* attacks are stealthy, *i.e.*, the spy process cannot be detected based on cache hits and misses, or state-of-the-art detection mechanisms. The *Flush+Flush* attack runs in a higher frequency and thus is faster than any existing cache attack. With 496 KB/s in a cross-core covert channel it is 6.7 times faster than any previously published cache covert channel.

## 1 Introduction

The CPU cache is a microarchitectural element that reduces the memory access time of recently-used data. It is shared across cores in modern processors, and is thus a piece of hardware that has been extensively studied in terms of information leakage. Cache attacks include covert and cryptographic side channels, but caches have also been exploited in other types of attacks, such as bypassing kernel ASLR [14], detecting cryptographic libraries [17], or keystroke logging [10]. Hardware performance counters have been proposed recently as an OS-level detection mechanism for cache attacks and Rowhammer [5, 13, 31]. This countermeasure is based on the assumption that all cache attacks cause significantly more cache hits and cache misses than benign applications. While this assumption seems reasonable, it is unknown whether there are cache attacks that do not cause a significant number of cache hits and cache misses.

In this article, we present the *Flush+Flush* attack. *Flush+Flush* exploits the fact that the execution time of the `clflush` instruction is shorter if the data

This paper has been accepted at DIMVA 2016 (`dimva2016.mondragon.edu/en`). The final publication is available at link.springer.com (`http://link.springer.com/`).

† Part of the work was done while author was affiliated to Technicolor and Eurecom.

is not cached and higher if the data is cached. At the same time, the `clflush` instruction evicts the corresponding data from all cache levels. *Flush+Flush* exploits the same hardware and software properties as *Flush+Reload* [45]: it works on read-only shared memory, cross-core attack and in virtualized environments. In contrast to *Flush+Reload*, *Flush+Flush* does not make any memory accesses and thus does not cause any cache misses at all and only a minimal number of cache hits. This distinguishes *Flush+Flush* from any other cache attack. However, with both *Flush+Reload* and *Flush+Flush* the victim process experiences an increased number of cache misses.

We evaluate *Flush+Flush* both in terms of *performance* and *detectability* in three scenarios: a covert channel, a side-channel attack on user input, and a side-channel attack on AES with T-tables. We implement a detection mechanism that monitors cache references and cache misses of the last-level cache, similarly to state of the art [5, 13, 31]. We show that existing cache attacks as well as Rowhammer attacks can be detected using performance counters. However, we demonstrate that this countermeasure is non-effective against the *Flush+Flush* attack, as the fundamental assumption fails. The *Flush+Flush* attack is thus more stealthy than existing cache attacks, *i.e.*, a *Flush+Flush* spy process cannot be detected based on cache hits and cache misses. Thus, it cannot be detected by state-of-the-art detection mechanisms.

The *Flush+Flush* attack runs in a higher frequency and thus is faster than any existing cache attack in side-channel and covert channel scenarios. It achieves a cross-core transmission rate of 496 KB/s, which is 6.7 times faster than any previously published cache covert channel. The *Flush+Flush* attack does not trigger prefetches and thus allows to monitor multiple addresses within a 4 KB memory range in contrast to *Flush+Reload* that fails in these scenarios [10].

Our key contributions are:

- We detail a new cache attack technique that we call *Flush+Flush*. It relies only on the difference in timing of the `clflush` instruction between cached and non-cached memory accesses.
- We show that in contrast to all other attacks, *Flush+Flush* is stealthy, *i.e.*, it cannot be detected using hardware performance counters. We show that *Flush+Flush* also outperforms all existing cache attacks in terms of speed.

The remainder of this paper is organized as follows. Section 2 provides background information on CPU caches, shared memory, and cache attacks. Section 3 describes the *Flush+Flush* attack. Section 4 investigates how to leverage hardware performance counters to detect cache attacks. We compare the performance and detectability of *Flush+Flush* attacks compared to state-of-the-art attacks in three scenarios: a covert channel in Section 5, a side-channel attack on keystroke timings in Section 6, and on cryptographic algorithms in Section 7. Section 8 discusses implications and countermeasures. Section 9 discusses related work. Finally, we conclude in Section 10.

# 2 Background

## 2.1 CPU Caches

CPU caches hide the memory accesses latency to the slow physical memory by buffering frequently used data in a small and fast memory. Modern CPU architectures implement $n$-way set-associative caches, where the cache is divided into cache sets, and each cache set comprises several cache lines. A line is loaded in a set depending on its address, and each line can occupy any of the $n$ ways.

On modern Intel processors, there are three cache levels. The L3 cache, also called last-level cache, is shared between all CPU cores. The L3 cache is inclusive, *i.e.*, all data within the L1 and L2 caches is also present in the L3 cache. Due to these properties, executing code or accessing data on one core has immediate consequences even for the private caches of the other cores. This can be exploited in so called cache attacks. The last-level cache is divided into as many slices as cores, interconnected by a ring bus. Since the Sandy Bridge microarchitecture, each physical address is mapped to a slice by an undocumented so-called *complex-addressing* function, that has recently been reversed-engineered [27].

A cache replacement policy decides which cache line to replace when loading new data in a set. Typical replacement policies are least-recently used (LRU), variants of LRU and bimodal insertion policy where the CPU can switch between the two strategies to achieve optimal cache usage [33]. The unprivileged `clflush` instruction evicts a cache line from all the cache hierarchy. However, a program can also evict a cache line by accessing enough memory.

## 2.2 Shared Memory

Operating systems and hypervisors instrument shared memory to reduce the overall physical memory utilization and the TLB utilization. Shared libraries are loaded into physical memory only once and shared by all programs using them. Thus, multiple programs access the same physical pages mapped within their own virtual address space.

The operating system similarly optimizes mapping of files, forking a process, starting a process twice, or using `mmap` or `dlopen`. All cases result in a memory region shared with all other processes mapping the same file.

On personal computers, smartphones, private cloud systems and even in public clouds [1], another form of shared memory can be found, namely content-based page deduplication. The hypervisor or operating system scans the physical memory for byte-wise identical pages. Identical pages are remapped to the same physical page, while the other page is marked as free. This technique can lower the use of physical memory and TLB significantly. However, sharing memory between completely unrelated and possibly sandboxed processes, and between processes running in different virtual machines brings up security and privacy concerns.

### 2.3 Cache Attacks and Rowhammer

Cache attacks exploit timing differences caused by the lower latency of CPU caches compared to physical memory. Access-driven cache attacks are typically devised in two types: *Prime+Probe* [30, 32, 39] and *Flush+Reload* [11, 45].

In *Prime+Probe* attacks, the attacker occupies a cache set and measures whenever a victim replaces a line in that cache set. Modern processors have a physically indexed last-level cache, use complex addressing, and undocumented replacement policies. Cross-VM side-channel attacks [16, 24] and covert channels [28] that tackle these challenges have been presented in the last year. Oren et al. [29] showed that a *Prime+Probe* cache attack can be launched from within sandboxed JavaScript in a browser, allowing a remote attacker to eavesdrop on network traffic statistics or mouse movements through a website.

*Flush+Reload* is a two phase attack that works on a single cache line. First, it *flushes* a cache line using the `clflush` instruction, then it measures the time it takes to *reload* the data. Based on the time measurement, the attacker determines whether a targeted address has been reloaded by another process in the meantime. In contrast to *Prime+Probe*, *Flush+Reload* exploits the availability of shared memory and especially shared libraries between the attacker and the victim program. Applications of *Flush+Reload* have been shown to be reliable and powerful, mainly to attack cryptographic algorithms [12, 17, 18, 48].

Rowhammer is not a typical cache attack but a DRAM vulnerability that causes random bit flips by repeatedly accessing a DRAM row [20]. It however shares some similarities with caches attacks since the accesses must bypass all levels of caches to reach DRAM and trigger bit flips. Attacks exploiting this vulnerability have already been demonstrated to gain root privileges and to evade a sandbox [36]. Rowhammer causes a significant number of cache hits and cache misses, that resemble a cache attack.

## 3 The *Flush+Flush* Attack

The *Flush+Flush* attack is a faster and stealthier alternative to existing cache attacks that also has fewer side effects on the cache. In contrast to other cache attacks, it does not perform any memory accesses. For this reason it causes no cache misses and only a minimal number of cache hits. Thus, proposed detection mechanisms based on hardware performance counters fail to detect the *Flush+Flush* attack. *Flush+Flush* exploits the same hardware and software properties as *Flush+Reload*. It runs across cores and in virtualized environments if read-only shared memory with the victim process can be acquired.

Our attack builds upon the observation that the `clflush` instruction can abort early in case of a cache miss. In case of a cache hit, it has to trigger eviction on all local caches. This timing difference can be exploited in form of a cache attack, but it can also be used to derive information on cache slices and CPU cores as each core can access its own cache slice faster than others.

The attack consists of only one phase, that is executed in an endless loop. It is the execution of the `clflush` instruction on a targeted shared memory line.

The attacker measures the execution time of the `clflush` instruction. Based on the execution time, the attacker decides whether the memory line has been cached or not. As the attacker does not load the memory line into the cache, this reveals whether some other process has loaded it. At the same time, `clflush` evicts the memory line from the cache for the next loop round of the attack.

The measurement is done using the `rdtsc` instruction that provides a sub-nanosecond resolution timestamp. It also uses `mfence` instructions, as `clflush` is only ordered by `mfence`, but not by any other means.

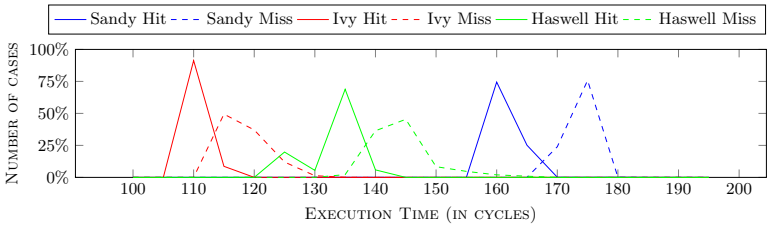


**Fig. 1.** Execution time of the `clflush` instruction on cached and uncached memory on different CPU architectures

Figure 1 shows the execution time histogram of the `clflush` instruction for cached and non-cached memory lines, run on the three setups with different recent microarchitectures: a Sandy Bridge i5-2540M, an Ivy Bridge i5-3320M and a Haswell i7-4790. The timing difference of the peaks is 12 cycles on Sandy Bridge, 9 cycles on Ivy Bridge, and 12 cycles on Haswell. If the address maps to a remote core, another penalty of 3 cycles is added to the minimum execution time for cache hits. The difference is enough to be observed by an attacker. We discuss this timing difference and its implications in Section 9.1. In either case the execution time is less than the access time for both memory cached in the last-level cache and memory accesses that are not cached. Therefore, *Flush+Flush* is significantly faster than any other last-level cache attack.

The *Flush+Flush* attack inherently has a slightly lower accuracy than the *Flush+Reload* technique in some cases, due to the lower timing difference between a hit and a miss and because of a lower access time on average. Nevertheless, the same amount of information is extracted faster using the *Flush+Flush* attack due to the significantly lower execution time. Furthermore, the reload-step of the *Flush+Reload* attack can trigger the prefetcher and thus destroy measurements by fetching data into the cache. This is the case especially when monitoring more than one address within a physical page [10]. As the *Flush+Flush* attack never performs any memory accesses, this problem does not exist and the *Flush+Flush* attack achieves an even higher accuracy here. For the same reason, the *Flush+Flush* attack causes no cache misses and only a minimal number of cache hits. Thus, recently proposed detection mechanisms using cache references and cache misses fail to detect *Flush+Flush*.

| Name | Description |
|---|---|
| BPU_RA/_RM | Branch prediction unit read accesses/misses |
| BRANCH_INSTRUCTIONS/_MISSES | Retired branch instructions/mispredictions |
| BUS_CYCLES | Bus cycles |
| CACHE_MISSES/_REFERENCES | Last-level cache misses/references |
| UNC_CBO_CACHE_LOOKUP | C-Box events incl. `clflush` (all slices) |
| CPU_CYCLES/REF_CPU_CYCLES | CPU cycles with/without scaling |
| DTLB_RA/_RM/_WA/_WM | Data TLB read/write accesses/misses |
| INSTRUCTIONS | Retired instructions |
| ITLB_RA/_RM | Instruction TLB read/write accesses |
| L1D_RA/_RM/_WA/_WM | L1 data cache read/write accesses/misses |
| L1I_RM | L1 instruction cache read misses |
| LL_RA/_WA | Last-level cache read/write accesses |

**Table 1.** List of hardware performance events we use.

## 4 Detecting Cache Attacks with Hardware Performance Counters

Cache attacks can lead to an increased number of cache hits or cache misses in the attacker process or in other processes. Thus, it may be possible to detect abnormal behavior on a system level. However, to stop or prevent an attack, it is necessary to identify the attacking process. Therefore, we consider an attack *stealthy* if the attacking spy process cannot be identified.

Hardware performance counters are special-purpose registers that are used to monitor special hardware-related events. Events that can be monitored include cache references and cache misses on the last-level cache. They are mostly used for performance analysis and fine tuning, but have been found to be suitable to detect Rowhammer and the *Flush+Reload* attack [5,13,31]. The focus of our work is to show that detection of existing attacks is straight-forward, but detection of the *Flush+Flush* attack using these performance counters is infeasible, due to the absence of cache misses and the minimal number of cache references.

We analyze the feasibility of such detection mechanisms using the Linux `perf_event_open` syscall interface that provides userspace access to a subset of all available performance counters on a per-process basis. The actual accesses to the model specific registers are performed in the kernel. The same information can be used by a system service to detect ongoing attacks. During our tests we ran the performance monitoring with system service privileges.

We analyzed all 23 hardware and cache performance events available with the Linux syscall interface on our system. Additionally, we analyzed the so called *uncore* [15] performance monitoring units and found one called C-Box that is influenced by cache hits, misses and `clflush` instructions directly. The `UNC_CBO_CACHE_LOOKUP` event of the C-Box allows monitoring a last-level cache lookups per cache slice, including by the `clflush` instruction. The C-Box monitoring units are not available through a generic interface but only through model specific registers. Table 1 lists all events we evaluated. We found that there are no other performance counters documented to monitor cache hits, misses or `clflush` instructions specifically. Furthermore, neither the hypervisor nor the

operating system can intercept the `clflush` instruction or monitor the frequency of `clflush` instructions being executed using performance counters.

The number of performance events that can be monitored simultaneously is limited by hardware. On all our test systems it is possible to monitor up to 4 events simultaneously. Thus, any detection mechanism can only use 4 performance events simultaneously.

We evaluated the 24 performance counters for the following scenarios:

1. Idle: idle system,
2. Firefox: user scrolling down a chosen Twitter feed in Firefox,
3. OpenTTD: user playing a game
4. stress -m 1: loop reading and writing in dynamically allocated 256 MB arrays,
5. stress -c 1: loop doing a CPU computation with almost no memory,
6. stress -i 1: loop calling the I/O `sync()` function,
7. *Flush+Reload*: cache attack on the GTK library to spy on keystroke events,
8. Rowhammer: Rowhammer attack.

The first 3 scenarios are casual computer usage scenarios, the next 3 cause a benign high load situation and the last 2 perform an attack. A good detection mechanism classifies as benign the scenarios 1 to 6 and as attacks 7 and 8.

We use the instruction TLB (ITLB) performance counters (`ITLB_RA`+`ITLB_WA`) to normalize the performance counters to make cache attacks easier to detect, and prevent scenarios 2 and 3 from being detected as malicious. Indeed, the main loop that is used in the *Flush+Reload* and Rowhammer attacks causes a high number of last-level cache misses while executing only a small piece of code. Executing only a small piece of code causes a low pressure on the ITLB.

Table 2 shows a comparison of performance counters for the 8 scenarios tested over 135 seconds. These tests were performed in multiple separate runs as the performance monitoring unit can only monitor 4 events simultaneously. Not all cache events are suitable for detection. The `UNC_CBO_CACHE_LOOKUP` event that counts cache slice events including `clflush` operations shows very high values in case of `stress -i`. It would thus lead to false positives. Similarly, the `INSTRUCTIONS` event used by Chiappetta et al. [5] has a significantly higher value in case of `stress -c` than in the attack scenarios and would cause false positives in the case of benign CPU intensive activities. The `REF_CPU_CYCLES` is the unscaled total number of CPU cycles consumed by the process. Divided by the TLB events, it shows how small the executed loop is. The probability of false positive matches is high, for instance in the case of `stress -c`.

Thus, 4 out of 24 events allow detecting both *Flush+Reload* and Rowhammer without causing false positives for benign applications. The rationale behind these events is as follows:

1. `CACHE_MISSES` occur after data has been flushed from the last-level cache,
2. `CACHE_REFERENCES` occur when reaccessing memory,
3. `L1D_RM` occur because flushing from last-level cache also flushes from the lower cache levels,

| Event / Test | Idle | Firefox | OTTD | stress -m | stress -c | stress -i | F+R | Rowhammer |
|---|---|---|---|---|---|---|---|---|
| BPU_RA | 4.35 | 14.73 | 67.21 | 92.28 | 6 109 276.79 | 3.23 | 127 443.28 | 23 778.66 |
| BPU_RM | 0.36 | 0.32 | 1.87 | 0.00 | 12 320.23 | 0.36 | 694.21 | 25.53 |
| BRANCH_INST. | 4.35 | 14.62 | 74.73 | 92.62 | 6 094 264.03 | 3.23 | 127 605.71 | 23 834.59 |
| BRANCH_MISS. | 0.36 | 0.31 | 2.06 | 0.00 | 12 289.93 | 0.35 | 693.97 | 25.85 |
| BUS_CYCLES | 4.41 | 1.94 | 12.39 | 52.09 | 263 816.26 | 6.20 | 30 420.54 | 98 406.44 |
| CACHE_MISSES | 0.09 | 0.15 | 2.35 | 58.53 | 0.06 | 1.92 | 693.67 | 13 766.65 |
| CACHE_REFER. | 0.40 | 0.98 | 6.84 | 61.05 | 0.31 | 2.28 | 693.92 | 13 800.01 |
| UNC_CBO_LOO. | 432.99 | 3.88 | 18.66 | 4 166.71 | 0.31 | 343 224.44 | 2 149.72 | 50 094.17 |
| CPU_CYCLES | 38.23 | 67.45 | 449.23 | 2 651.60 | 9 497 363.56 | 237.62 | 1 216 701.51 | 3 936 969.93 |
| DTLB_RA | 5.11 | 19.19 | 123.68 | 31.78 | 6 076 031.42 | 3.04 | 47 123.44 | 25 459.36 |
| DTLB_RM | 0.07 | 0.09 | 1.67 | 0.05 | 0.05 | 0.04 | 0.05 | 0.03 |
| DTLB_WA | 1.70 | 11.18 | 54.88 | 30.97 | 3 417 764.10 | 1.13 | 22 868.02 | 25 163.03 |
| DTLB_WM | 0.01 | 0.01 | 0.03 | 2.50 | 0.01 | 0.01 | 0.01 | 0.16 |
| INSTRUCTIONS | 20.24 | 66.04 | 470.89 | 428.15 | 20 224 639.96 | 11.77 | 206 014.72 | 132 896.65 |
| ITLB_RA | 0.95 | 0.97 | 0.98 | 1.00 | 0.96 | 0.97 | 0.96 | 0.97 |
| ITLB_RM | 0.05 | 0.03 | 0.02 | 0.00 | 0.04 | 0.03 | 0.04 | 0.03 |
| L1D_RA | 5.11 | 18.30 | 128.75 | 31.53 | 6 109 271.97 | 3.01 | 47 230.08 | 26 173.65 |
| L1D_RM | 0.37 | 0.82 | 8.47 | 61.63 | 0.51 | 0.62 | 695.22 | 15 630.85 |
| L1D_WA | 1.70 | 10.69 | 57.66 | 30.72 | 3 436 461.82 | 1.13 | 22 919.77 | 25 838.20 |
| L1D_WM | 0.12 | 0.19 | 1.50 | 30.57 | 0.16 | 0.44 | 0.23 | 10.01 |
| L1I_RM | 0.12 | 0.65 | 0.21 | 0.03 | 0.65 | 1.05 | 1.17 | 1.14 |
| LL_RA | 0.14 | 0.39 | 5.61 | 30.73 | 0.12 | 0.47 | 695.35 | 9 067.77 |
| LL_WA | 0.01 | 0.02 | 0.74 | 30.30 | 0.01 | 0.01 | 0.02 | 4 726.97 |
| REF_CPU_CYC. | 157.70 | 69.69 | 445.89 | 1 872.05 | 405 922.02 | 223.08 | 1 098 534.32 | 3 542 570.00 |

**Table 2.** Comparison of performance counters normalized to the number of ITLB events in different cache attacks and normal scenarios over 135 seconds in separate runs.

4. `LL_RA` are a subset of the `CACHE_REFERENCES` counter, they occur when reaccessing memory,

Two of the events are redundant: `L1D_RM` is redundant with `CACHE_MISSES`, and `LL_RA` with `CACHE_REFERENCES`. We will thus focus on the `CACHE_MISSES` and `CACHE_REFERENCES` events as proposed in previous work [5, 13, 31].

We define that a process is considered malicious if more than $k_m$ cache miss or $k_r$ cache reference per ITLB event are observed. The attack is detected if

$$\frac{C_{\texttt{CACHE\_MISSES}}}{C_{\texttt{ITLB\_RA}} + C_{\texttt{ITLB\_WA}}} \geq k_m, \quad \text{or} \quad \frac{C_{\texttt{CACHE\_REFERENCES}}}{C_{\texttt{ITLB\_RA}} + C_{\texttt{ITLB\_WA}}} \geq k_r,$$

with $C$ the value of the corresponding performance counter. The operating system can choose the frequency in which to run the detection checks.

The thresholds for the cache reference and cache hit rate are determined based on a set of benign applications and malicious applications. It is chosen to have the maximum distance to the minimum value for any malicious application and the maximum value for any benign application. In our case this is $k_m = 2.35$ and $k_r = 2.34$. Based on these thresholds, we perform a classification of processes into malicious and benign processes. We tested this detection mechanism against various cache attacks and found that it is suitable to detect different *Flush+Reload*, *Prime+Probe* and Rowhammer attacks as malicious. However, the focus of our work is not the evaluation of detection mechanisms based on performance counters, but to show that such detection mechanisms cannot reliably detect the *Flush+Flush* attack due to the absence of cache misses and a minimal number of cache references.

In the following sections, we evaluate the performance and the detectability of *Flush+Flush* compared to the state-of-the-art cache attacks *Flush+Reload* and *Prime+Probe* in three scenarios: a covert channel, a side channel on user input and a side channel on AES with T-tables.

## 5 Covert Channel Comparison

In this section, we describe a generic low-error cache covert channel framework. In a covert channel, an attacker runs two unprivileged applications on the system under attack. The processes are cooperating to communicate with each other, even though they are not allowed to by the security policy. We show how the two processes can communicate using the *Flush+Flush*, *Flush+Reload*, and *Prime+Probe* technique. We compare the performance and the detectability of the three implementations. In the remainder of the paper, all the experiments are performed on a Haswell i7-4790 CPU.

### 5.1 A Low-error Cache Covert Channel Framework

In order to perform meaningful experiments and obtain comparable and fair results, the experiments must be reproducible and tested in the same conditions. This includes the same hardware setup, and the same protocols. Indeed, we cannot compare covert channels from published work [24, 28] that have different capacities and error rates. Therefore, we build a framework to evaluate covert channels in a reproducible way. This framework is generic and can be implemented over any covert channel that allows bidirectional communication, by implementing the `send()` and `receive()` functions.

The central component of the framework is a simple transmission protocol. Data is transmitted in packets of $N$ bytes, consisting of $N - 3$ bytes payload, a 1 byte sequence number and a CRC-16 checksum over the packet. The sequence number is used to distinguish consecutive packets. The sender retransmits packets until the receiver acknowledges it. Packets are acknowledged by the receiver if the checksum is valid.

Although errors are still possible in case of a false positive CRC-16 checksum match, the probability is low. We choose the parameters such that the effective error rate is below 5%. The channel capacity measured with this protocol is comparable and reproducible. Furthermore, it is close to the effective capacity in a real-world scenario, because error-correction cannot be omitted. The number of transmitted bits is the minimum of bits sent and bits received. The transmission rate can be computed by dividing the number of transmitted bits by the runtime. The error rate is given by the number of all bit errors between the sent bits and received bits, divided by the number of transmitted bits.

### 5.2 Covert Channel Implementations

We first implemented the *Flush+Reload* covert channel. By accessing fixed memory locations in a shared library the a 1 is transmitted, whereas a 0 is transmitted

by omitting the access. The receiver performs the actual *Flush+Reload* attack to determine whether a 1 or a 0 was transmitted. The bits retrieved are then parsed as a data frame according to the transmission protocol. The sender also monitors some memory locations using *Flush+Reload* for cache hits too, to receive packet acknowledgments.

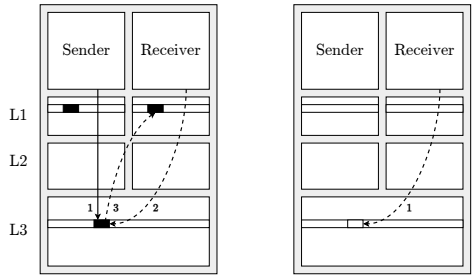


**Fig. 2.** Illustration of the *Flush+Flush* covert channel.

The second implementation is the *Flush+Flush* covert channel, illustrated by Figure 2. It uses the same sender process as the *Flush+Reload* covert channel. To transmit a 1 (Figure 2-a), the sender accesses the memory location, that is cached (step 1). This time, the receiver only flushes the shared line. As the line is present in the last-level cache by inclusiveness, it is flushed from this level (step 2). A bit also indicates that the line is present in the L1 cache, and thus must also be flushed from this level (step 3). To transmit a 0 (Figure 2-b), the sender stays idle. The receiver flushes the line (step 1). As the line is not present in the last-level cache, it means that it is also not present in the lower levels, which results in a faster execution of the `clflush` instruction. Thus only the sender process performs memory accesses, while the receiver only flushes cache lines. To send acknowledgment bytes the receiver performs memory accesses and the sender runs a *Flush+Flush* attack.

The third implementation is the *Prime+Probe* covert channel. It uses the same attack technique as Liu et al. [24], Oren et al. [29], and Maurice et al. [28]. The sender transmits a 1 bit by priming a cache set. The receiver probes the same cache set. Again the receiver determines whether a 1 or a 0 was transmitted. We make two adjustments for convenience and to focus solely on the transmission part. First, we compute a static eviction set by using the complex addressing function [27] on physical addresses. This avoids the possibility of errors introduced by timing-based eviction set computation. Second, we map the shared library into our address space to determine the physical address to attack to make an agreement on the cache sets in sender and receiver. Yet, the shared library is never accessed and unmapped even before the *Prime+Probe* attack is started. We assume that the sender and receiver have agreed on the cache sets in a preprocessing step. This is practical even for a timing-based approach.

| Technique | Packet size | Capacity in KB/s | Error rate | Sender references | Sender misses | Sender stealth | Receiver references | Receiver misses | Receiver stealth |
|---|---|---|---|---|---|---|---|---|---|
| ***Flush+Flush*** | 28 | 496 | 0.84% | 1809.26 | 96.66 | ✗ | 1.75 | 1.25 | ✓ |
| *Flush+Reload* | 28 | 298 | 0.00% | 526.14 | 56.09 | ✗ | 110.52 | 59.16 | ✗ |
| *Flush+Reload* | 5 | 132 | 0.01% | 6.19 | 3.20 | ✗ | 45.88 | 44.77 | ✗ |
| ***Flush+Flush*** | 5 | 95 | 0.56% | 425.99 | 418.27 | ✗ | 0.98 | 0.95 | ✓ |
| *Prime+Probe* | 5 | 67 | 0.36% | 48.96 | 31.81 | ✗ | 4.64 | 4.45 | ✗ |
| *Flush+Reload* | 4 | 54 | 0.00% | 0.86 | 0.84 | ✓ | 2.74 | 1.25 | ✗ |
| ***Flush+Flush*** | 4 | 52 | 1.00% | 0.06 | 0.05 | ✓ | 0.59 | 0.59 | ✓ |
| *Prime+Probe* | 4 | 34 | 0.04% | 55.57 | 32.66 | ✗ | 5.23 | 5.01 | ✗ |

**Table 3.** Comparison of capacity and detectability of the three cache covert channels with different parameters. *Flush+Flush* and *Flush+Reload* use the same sender process.

### 5.3 Performance Evaluation

Table 3 compares the capacity and the detectability of the three covert channels in different configurations. The *Flush+Flush* covert channel is the fastest of the three covert channels. With a packet size of 28 bytes the transmission rate is 496 KB/s. At the same time the effective error rate is only 0.84%. The *Flush+Reload* covert channel also achieved a good performance at a packet size of 28 bytes. The transmission rate then is 298 KB/s and the error rate $< 0.005\%$. With a packet size of 4 bytes, the performance is lower in all three cases.

A *Prime+Probe* covert channel with a 28-byte packet size is not realistic. First, to avoid triggering the hardware prefetcher we do not access more than one address per physical page. Second, for each eviction set we need 16 addresses. Thus we would require $28\text{B} \cdot 4096 \cdot 16 = 14\,\text{GB}$ of memory only for the eviction sets. For *Prime+Probe* we achieved the best results with a packet size of 5 bytes. With this configuration the transmission rate is 68 KB/s at an error rate of 0.14%, compared to 132 KB/s using *Flush+Reload* and 95 KB/s using *Flush+Flush*.

The *Flush+Flush* transmission rate of 496 KB/s is significantly higher than any other state-of-the-art cache covert channels. It is 6.7 times as fast as the fastest cache covert channel to date [24] at a comparable error rate. Our covert channel based on *Flush+Reload* is also faster than previously published cache covert channels, but still much slower than the *Flush+Flush* covert channel. Compared to our *Prime+Probe* covert channel, *Flush+Flush* is 7.3 times faster.

### 5.4 Detectability

Table 3 shows the evaluation of the detectability for packet sizes that yielded the highest performance in one of the cases. *Flush+Reload* and *Flush+Flush* use the same sender process, the reference and miss count is mainly influenced by the number of retransmissions and executed program logic. *Flush+Reload* is detected in all cases either because of its sender or its receiver, although its sender process with a 4-byte packet size stays below the detection threshold. The *Prime+Probe* attack is always well above the detection threshold and therefore always detected as malicious. All *Flush+Flush* receiver processes are classified

| Technique | Cache references | Cache misses | Stealthy |
|---|---|---|---|
| *Flush+Reload* | 5.140 | 5.138 | ✗ |
| ***Flush+Flush*** | 0.002 | 0.000 | ✓ |

**Table 4.** Comparison of performance counters normalized to the number of ITLB events for cache attacks on user input.

as benign. However, only the sender process used for the *Flush+Flush* and the *Flush+Reload* covert channels with a 4-byte packet size is classified as benign.

The receiver process performs most of the actual cache attack. If it is sufficient to keep the receiver process stealthy, *Flush+Flush* clearly outperforms all other cache attacks. If the sender has to be stealthy as well, the sender process used by *Flush+Flush* and *Flush+Reload* performs better than the *Prime+Probe* sender process. However, due to the high number of cache hits it is difficult to keep the sender process below the detection threshold. An adversary could choose to reduce the transmission rate in order to be stealthier in either case.

## 6 Side-Channel Attack on User Input

Another cache attack that has been demonstrated recently using *Flush+Reload*, is eavesdropping on keystroke timings. We attack an address in the GTK library invoked when processing keystrokes. The attack is implemented as a program that constantly flushes the address, and derives when a keystroke occurred, based on memory access times or the execution time of the `clflush` instruction.

### 6.1 Performance Evaluation

We compare the three attacks *Flush+Flush*, *Flush+Reload*, and *Prime+Probe*, based on their performance in this side-channel attack scenario. During each test we simulate a user typing a 1000-character text into an editor. Each test takes 135 seconds. As expected, *Flush+Reload* has a very high accuracy of 96.1%. This allows direct logging of keystroke timings. *Flush+Flush* performs notably well, with 74.7% correctly detected keystrokes. However, this makes a practical attack much harder than with *Flush+Reload*. The attack with *Prime+Probe* yielded no meaningful results at all due to the high noise level. In case of *Flush+Reload* and *Flush+Flush* the accuracy can be increased significantly by attacking 3 addresses that are used during keystroke processing simultaneously. The decision whether a keystroke was observed is then based on these 3 addresses increasing the accuracy significantly. Using this technique reduces the error rate in case of *Flush+Reload* close to 100% and above 92% in case of *Flush+Flush*.

### 6.2 Detectability

To evaluate the detectability we again monitored the cache references and cache misses events, and compared the three cache attacks with each other and with an

idle system. Table 4 shows that *Flush+Reload* generates a high number of cache references, whereas *Flush+Flush* causes a negligible number of cache references. We omitted *Prime+Probe* in this table as it was not sufficiently accurate to perform the attack.

*Flush+Reload* yields the highest accuracy in this side-channel attack, but it is easily detected. The accuracy of *Flush+Flush* can easily be increased to more than 92% and it still is far from being detected. Thus, *Flush+Flush* is a viable and stealthy alternative to the *Flush+Reload* attack as it is not classified as malicious based on the cache references or cache misses performance counters.

## 7 Side-Channel Attack on AES with T-Tables

To round up our comparison with other cache attacks, we compare *Flush+Flush*, *Flush+Reload*, and *Prime+Probe* in a high frequency side-channel attack scenario. Finding new cache attacks is out of scope of our work. Instead, we try to perform a fair comparison between the different attack techniques by implementing a well known cache attack using the three techniques on a vulnerable implementation of a cryptographic algorithm. We attack the OpenSSL T-Table-based AES implementation that is known to be susceptible to cache attacks [2, 30]. This AES implementation is disabled by default for security reasons, but still exists for the purpose of comparing new and existing side-channel attacks.

The AES algorithm uses the T-tables to compute the ciphertext based on the secret key $k$ and the plaintext $p$. During the first round, table accesses are made to entries $T_j[p_i \oplus k_i]$ with $i \equiv j \mod 4$ and $0 \leq i < 16$. Using a cache attack it is possible to derive values for $p_i \oplus k_i$ and thus, possible key-byte values $k_i$ in case $p_i$ is known.

### 7.1 Attack Implementation Using Flush+Flush

The implementation of the chosen-plaintext attack side-channel attacks for the three attack techniques is very similar. The attacker triggers an encryption, choosing $p_i$ while all $p_j$ with $i \neq j$ are random. One cache line holds 16 T-Table entries. The cache attack is now performed on the first line of each T-Table. The attacker repeats the encryptions with new random plaintext bytes $p_j$ until only one $p_i$ remains to always cause a cache hit. The attacker learns that $p_i \oplus k_i \equiv_{\lceil 4 \rceil} 0$ and thus $k_i \equiv_{\lceil 4 \rceil} p_i$. After performing the attack for all 16 key bytes, the attacker has derived 64 bits of the secret key $k$. As we only want to compare the three attack techniques, we do not extend this attack to a full key recovery attack.

### 7.2 Performance Evaluation

Figure 3 shows a comparison of cache templates generated with *Flush+Reload*, *Flush+Flush*, and *Prime+Probe* using 1 000 000 encryptions to create a visible pattern in all three cases. Similar templates can be found in previous work [10, 30, 37]. Table 5 shows how many encryptions are necessary to determine the

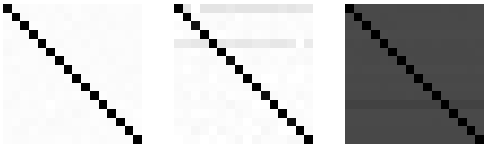

**Fig. 3.** Comparison of Cache Templates (address range of the first T-table) generated using *Flush+Reload* (left), *Flush+Flush* (middle), and *Prime+Probe* (right). In all cases $k_0 =$ `0x00`.

| Technique | Number of encryptions |
|---|---|
| *Flush+Reload* | 250 |
| ***Flush+Flush*** | 350 |
| *Prime+Probe* | 4 800 |

**Table 5.** Number of encryptions to determine the upper 4 bits of a key byte.

upper 4 bits correctly. We performed encryptions until the correct guess for the upper 4 bits of key byte $k_0$ had a 5% margin over all other key candidates. *Flush+Flush* requires around 1.4 times as many encryptions as *Flush+Reload*, but 13.7 times less than *Prime+Probe* to achieve the same accuracy.

*Flush+Flush* is the only attack that does not trigger the prefetcher. Thus, we can monitor multiple adjacent cache sets. By doing this we double the number of cache references, but increase the accuracy of the measurements so that 275 encryptions are sufficient to identify the correct key byte with a 5% margin. That is only 1.1 times as many encryptions as *Flush+Reload* and 17.5 times less than *Prime+Probe*. Thus, *Flush+Flush* on multiple addresses is faster at deriving the same information as *Flush+Reload*.

### 7.3 Detectability

Table 6 shows a comparison of the performance counters for the three attacks over 256 million encryptions. The *Flush+Flush* attack took only 163 seconds whereas *Flush+Reload* took 215 seconds and *Prime+Probe* 234 seconds for the identical attack. On a system level, it is possible to notice ongoing cache attacks on AES in all three cases due to the high number of cache misses caused by the AES encryption process. However, to stop or prevent the attack, it is necessary to detect the spy process. *Prime+Probe* exceeds the detection threshold by a factor of 468 and *Flush+Reload* exceeds the threshold by a factor of 1070. To stay below the detection threshold, slowing down the attack by at least the same factor would be necessary. In contrast, *Flush+Flush* is not detected based on our classifier and does not have to be slowed down to be stealthy.

| Technique | Cache references | Cache misses | Execution time in s | References (norm.) | Misses (norm.) | Stealthy |
|---|---|---|---|---|---|---|
| *Flush+Reload* | $1\,024 \cdot 10^6$ | 19 284 602 | 215 | 2 513.43 | 47.33 | ✗ |
| *Prime+Probe* | $4\,222 \cdot 10^6$ | 294 897 508 | 234 | 1 099.63 | 76.81 | ✗ |
| ***Flush+Flush*** | $768 \cdot 10^6$ | 1 741 | 163 | 1.40 | 0.00 | ✓ |

**Table 6.** Comparison of the performance counters when performing 256 million encryptions with different cache attacks and without an attack.

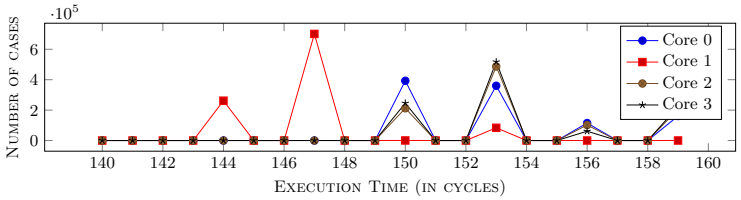


**Fig. 4.** Excerpt of the `clflush` histogram for an address in slice 1 on different cores. The lower execution time on core 1 shows that this address maps to slice 1.

## 8 Discussion

### 8.1 Using `clflush` to Detect Cores and Cache Slices

The *Flush+Flush* attack can be used to determine on which CPU core a process is running or to which cache slice an address maps. Indeed, a `clflush` on a remote cache slice takes longer than a `clflush` on a local cache slice, as shown in Figure 4. This is due to the ring bus architecture connecting remote slices. Knowing the physical address of a memory access on a local slice, we can then use the complex addressing function [27] to determine on which core the process runs. However, this would require high privileges. Yet, it is possible to determine to which slice an address maps without knowing the physical address by performing a timing attack. This can be done by an unprivileged process, as pinning a thread to a CPU core requires no privileges.

This can be exploited to detect colocation on the same CPU, CPU core or hyperthreading core in restricted environments even if the `cpuid` instructions is virtualized. It is more difficult to determine which CPU core a thread runs on based on memory access timings because of the influence of lower level caches. Such an attack has also not been demonstrated yet. The information on the executing CPU core can be used to enhance cache attacks and other attacks such as the Rowhammer attack [9, 20]. Running `clflush` on a local slice lowers the execution time of each Rowhammer loop round by a few cycles. The probability of bit flips increases as the execution time lowers, thus we can leverage the information whether an address maps to a local slice to improve this attack.

A similar timing difference also occurs upon memory accesses that are served from the local or a remote slice respectively. The reason again is the direct connection to the local cache slice while remote cache slices are connected via a ring

bus. However, as memory accesses will also be cached in lower level caches, it is more difficult to observe the timing difference without `clflush`. The `clflush` instruction directly manipulates the last-level cache, thus lower level caches cannot hide the timing difference.

While the operating system can restrict access on information such as the CPU core the process is running on and the physical address mapping to make efficient cache attacks harder, it cannot restrict access to the `clflush` instruction. Hence, the effect of such countermeasures is lower than expected.

### 8.2 Countermeasures

We suggest modifying the `clflush` instruction to counter the wide range of attacks that it can be used for. The difference in the execution time of `clflush` is 3 cycles depending on the cache slice and less than 12 cycles depending on whether it is a cache miss. In practice the `clflush` instruction is used only in rare situations and not in a high frequency. Thus, a hypothetical performance advantage cannot justify introducing these exploitable timing differences. We propose making `clflush` a constant-time instruction. This would prevent the *Flush+Flush* attack completely, as well as information leakage on cache slices and CPU cores.

*Flush+Flush* is the only cache attack that does not perform any memory accesses and thus causes no cache misses and only a minimal number of cache references. One theoretical way to detect our attack would be to monitor each load, e.g., by timing, and to stop when detecting too many misses. However, this solution is currently not practical, as a software-based solution that monitors each load would cause a significant performance degradation. A similar hardware-based solution called *informing loads* has been proposed by Kong et al. [21], however it needs a change in the instruction set. Without hardware modifications it would be possible to enable the `rdtsc` instruction only in privileged mode as can be done using seccomp on Linux [25] since 2008. Fogh [7] proposed to simulate the `rdtsc` in an interrupt handler, degrading the accuracy of measurements far enough to make cache attacks significantly harder.

*Flush+Reload* and *Flush+Flush* both require shared memory. If shared memory is not available, an attacker would have to resort to a technique that even works without shared memory such as *Prime+Probe*. Furthermore, making the `clflush` instruction privileged would prevent *Flush+Reload* and *Flush+Flush* as well. However, this would require changes in hardware and could not be implemented in commodity systems.

## 9 Related work

### 9.1 Detecting and Preventing Cache Attacks

Zhang et al. [47] proposed HomeAlone, a system-level solution that uses a *Prime+Probe* covert channel to *detect* the presence of a foe co-resident virtual

machine. The system monitors random cache sets so that friendly virtual machines can continue to operate if they change their workload, and that foe virtual machines are either detected or forced to be silent. Cache Template Attacks [10] can be used to detect attacks on shared libraries and binaries as a user. However, such a permanent scan increases the system load and can only detect attacks in a small address range within a reasonable response time.

Herath and Fogh [13] proposed to monitor cache misses to detect *Flush+Reload* attacks and Rowhammer. The system would slow down or halt all attacker processes. With the detection mechanism we implemented, we show that this technique is feasible for previous attacks but not for the *Flush+Flush* attack. Chiappetta et al. [5] proposed to build a trace of cache references and cache misses over the number of executed instructions to detect *Flush+Reload* attacks. They then proposed three methods to analyze this trace: a correlation-based method, and two other ones based on machine learning techniques. However, a learning phase is needed to detect malicious programs that are either from a set of known malicious programs or resemble a program from this set. They are thus are less likely to detect new or unknown cache attacks or Rowhammer attacks, in contrast to our ad-hoc detection mechanism. Payer [31] proposed a system called HexPADS to use cache references, cache misses, but also other events like page faults to detect cache attacks and Rowhammer at runtime.

Cache attacks can be *prevented* at three levels: at the hardware level, at the system level, and finally, at the application level. At the hardware level, several solutions have been proposed to prevent cache attacks, either by removing cache interferences, or randomizing them. The solutions include new secure cache designs [23, 41, 42] or altering the prefetcher policy [8]. However, hardware changes are not applicable to commodity systems. At the system level, page coloring provides cache isolation in software [19,34]. Zhang et al. [49] proposed a more relaxed isolation like repeated cache cleansing. These solutions cause performance issues, as they prevent optimal use of the cache. Application-level countermeasures seek to find the source of information leakage and patch it [4]. However, application-level countermeasures are bounded and cannot prevent cache attacks such as covert channels and Rowhammer. In contrast with prevention solutions that incur a loss of performance, using performance counters does not prevent attacks but rather detect them without overhead.

### 9.2 Usage of Hardware Performance Counters in Security

Hardware performance counters are made for performance monitoring, but security researchers found other applications. In defensive cases, performance counters allow detection of malware [6], integrity checking of programs [26], control flow integrity [44], and binary analysis [43]. In offensive scenarios, it has been used for side-channel attacks against AES [40] and RSA [3]. Performance counters have also been used by Maurice et al. [27] to reverse engineer the complex addressing function of the last-level cache of modern Intel CPUs.

### 9.3 Cache Covert Channels

Cache covert channels are a well-known problem, and have been studied relatively to the recent evolutions in microarchitecture. The two main types of access-driven attacks can be used to derive a covert channel. Covert channels using *Prime+Probe* have already been demonstrated in [24, 28]. *Flush+Reload* has been used for side-channels attacks [45], thus a covert channel can be derived easily. However, to the best of our knowledge, there was no study of the performance of such a covert channel.

In addition to building a covert channel with our new attack *Flush+Flush*, we re-implemented *Prime+Probe* and implemented *Flush+Reload*.[1] We thus provide an evaluation and a fair comparison between these different covert channels, in the same hardware setup and with the same protocol.

### 9.4 Side-Channel Attacks on User Inputs

Section 6 describes a side channel to eavesdrop on keystrokes. If an attacker has root access to a system, there are simple ways to implement a keylogger. Without root access, software-based side-channel attacks have already proven to be a reliable way to eavesdrop on user input. Attacks exploit the execution time [38], peaks in CPU and cache activity graphs [35], or system services [46]. Zhang et al. [46] showed that it is possible to derive key sequences from inter-keystroke timings obtained via `procfs`. Oren et al. [29] demonstrated that cache attacks in sandboxed JavaScript inside a browser can derive user activities, such as mouse movements. Gruss et al. [10] showed that auto-generated *Flush+Reload* attacks can be used to measure keystroke timings as well as identifying keys.

## 10 Conclusion

In this paper we presented *Flush+Flush*, a novel cache attack that, unlike any other, performs no memory accesses. Instead, it relies only on the execution time of the flush instruction to determine whether data is cached. *Flush+Flush* does not trigger prefetches and thus is applicable in more situations than other attacks. The *Flush+Flush* attack is faster than any existing cache attack. It achieves a transmission rate of 496 KB/s in a covert channel scenario, which is 6.7 times faster than any previous cache covert channel. As it performs no memory accesses, the attack causes no cache misses at all. For this reason, detection mechanisms based on performance counters to monitor cache activity fail, as their underlying assumption is incorrect.

While the *Flush+Flush* attack is significantly harder to detect than existing cache attacks, it can be prevented with small hardware modifications. Making the `clflush` instruction constant-time has no measurable impact on today's

[1] After public disclosure of the *Flush+Flush* attack on November 14, 2015, *Flush+Flush* has also been demonstrated on ARM-based mobile devices [22].

software and does not introduce any interface changes. Thus, it is an effective countermeasure that should be implemented.

Finally, the experiments led in this paper broaden the understanding of the internals of modern CPU caches. Beyond the adoption of detection mechanisms, the field of cache attacks benefits from these findings, both to discover new attacks and to be able to prevent them.

## 11 Acknowledgments

We would like to thank Mathias Payer, Anders Fogh, and our anonymous reviewers for their valuable comments and suggestions.

Supported by the EU Horizon 2020 programme under GA No. 644052 (HECTOR), the EU FP7 programme under GA No. 610436 (MATTHEW), the Austrian Research Promotion Agency (FFG) and Styrian Business Promotion Agency (SFG) under GA No. 836628 (SeCoS), and Cryptacus COST Action IC1403.

## References

1. Barresi, A., Razavi, K., Payer, M., Gross, T.R.: CAIN: silently breaking ASLR in the cloud. In: WOOT'15 (2015)
2. Bernstein, D.J.: Cache-timing attacks on AES. Tech. rep., Department of Mathematics, Statistics, and Computer Science, University of Illinois at Chicago (2005)
3. Bhattacharya, S., Mukhopadhyay, D.: Who watches the watchmen?: Utilizing Performance Monitors for Compromising keys of RSA on Intel Platforms. Cryptology ePrint Archive, Report 2015/621 (2015)
4. Brickell, E., Graunke, G., Neve, M., Seifert, J.P.: Software mitigations to hedge AES against cache-based software side channel vulnerabilities. Cryptology ePrint Archive, Report 2006/052 (2006)
5. Chiappetta, M., Savas, E., Yilmaz, C.: Real time detection of cache-based side-channel attacks using hardware performance counters. Cryptology ePrint Archive, Report 2015/1034 (2015)
6. Demme, J., Maycock, M., Schmitz, J., Tang, A., Waksman, A., Sethumadhavan, S., Stolfo, S.: On the feasibility of online malware detection with performance counters. ACM SIGARCH Computer Architecture News 41(3), 559–570 (2013)
7. Fogh, A.: Cache side channel attacks. `http://dreamsofastone.blogspot.co.at/2015/09/cache-side-channel-attacks.html` (2015)
8. Fuchs, A., Lee, R.B.: Disruptive Prefetching: Impact on Side-Channel Attacks and Cache Designs. In: Proceedings of the 8th ACM International Systems and Storage Conference (SYSTOR'15) (2015)
9. Gruss, D., Maurice, C., Mangard, S.: Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript. In: DIMVA'16 (2016)
10. Gruss, D., Spreitzer, R., Mangard, S.: Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In: USENIX Security Symposium (2015)
11. Gullasch, D., Bangerter, E., Krenn, S.: Cache Games – Bringing Access-Based Cache Attacks on AES to Practice. In: S&P'11 (2011)

12. Gülmezoğlu, B., Inci, M.S., Eisenbarth, T., Sunar, B.: A Faster and More Realistic Flush+Reload Attack on AES. In: Constructive Side-Channel Analysis and Secure Design (COSADE) (2015)
13. Herath, N., Fogh, A.: These are Not Your Grand Daddy's CPU Performance Counters - CPU Hardware Performance Counters for Security. Black Hat 2015 Briefings (Aug 2015), https://www.blackhat.com/docs/us-15/materials/us-15-Herath-These-Are-Not-Your-Grand-Daddys-CPU-Performance-Counters-CPU-Hardware-Performance-Counters-For-Security.pdf
14. Hund, R., Willems, C., Holz, T.: Practical Timing Side Channel Attacks against Kernel Space ASLR. In: 2013 IEEE Symposium on Security and Privacy. pp. 191–205 (2013)
15. Intel: Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3 (3A, 3B & 3C): System Programming Guide 253665 (2014)
16. Irazoqui, G., Eisenbarth, T., Sunar, B.: S$A: A Shared Cache Attack that Works Across Cores and Defies VM Sandboxing – and its Application to AES. In: S&P'15 (2015)
17. Irazoqui, G., Inci, M.S., Eisenbarth, T., Sunar, B.: Know thy neighbor: Crypto library detection in cloud. Proceedings on Privacy Enhancing Technologies 1(1), 25–40 (2015)
18. Irazoqui, G., Inci, M.S., Eisenbarth, T., Sunar, B.: Lucky 13 strikes back. In: AsiaCCS'15 (2015)
19. Kim, T., Peinado, M., Mainar-Ruiz, G.: StealthMem: system-level protection against cache-based side channel attacks in the cloud. In: Proceedings of the 21st USENIX Security Symposium (2012)
20. Kim, Y., Daly, R., Kim, J., Fallin, C., Lee, J.H., Lee, D., Wilkerson, C., Lai, K., Mutlu, O.: Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In: Proceeding of the 41st annual International Symposium on Computer Architecuture (ISCA'14) (2014)
21. Kong, J., Acıiçmez, O., Seifert, J.P., Zhou, H.: Hardware-software integrated approaches to defend against software cache-based side channel attacks. In: Proceedings of the 15th International Symposium on High Performance Computer Architecture (HPCA'09). pp. 393–404 (2009)
22. Lipp, M., Gruss, D., Spreitzer, R., Mangard, S.: Armageddon: Last-level cache attacks on mobile devices. CoRR abs/1511.04897 (2015)
23. Liu, F., Lee, R.B.: Random Fill Cache Architecture. In: IEEE/ACM International Symposium on Microarchitecture (MICRO'14). pp. 203–215 (2014)
24. Liu, F., Yarom, Y., Ge, Q., Heiser, G., Lee, R.B.: Last-Level Cache Side-Channel Attacks are Practical. In: S&P'15 (2015)
25. lwn.net: 2.6.26-rc1 short-form changelog. https://lwn.net/Articles/280913/ (May 2008)
26. Malone, C., Zahran, M., Karri, R.: Are hardware performance counters a cost effective way for integrity checking of programs. In: Proceedings of the sixth ACM workshop on Scalable trusted computing (2011)
27. Maurice, C., Le Scouarnec, N., Neumann, C., Heen, O., Francillon, A.: Reverse Engineering Intel Complex Addressing Using Performance Counters. In: RAID (2015)
28. Maurice, C., Neumann, C., Heen, O., Francillon, A.: C5: Cross-Cores Cache Covert Channel. In: DIMVA'15 (2015)
29. Oren, Y., Kemerlis, V.P., Sethumadhavan, S., Keromytis, A.D.: The Spy in the Sandbox – Practical Cache Attacks in Javascript. arXiv: 1502.07373v2 (2015)

30. Osvik, D.A., Shamir, A., Tromer, E.: Cache Attacks and Countermeasures: the Case of AES. In: CT-RSA 2006 (2006)
31. Payer, M.: HexPADS: a platform to detect "stealth" attacks. In: ESSoS'16 (2016)
32. Percival, C.: Cache missing for fun and profit. In: Proceedings of BSDCan (2005)
33. Qureshi, M.K., Jaleel, A., Patt, Y.N., Steely, S.C., Emer, J.: Adaptive insertion policies for high performance caching. ACM SIGARCH Computer Architecture News 35(2), 381 (Jun 2007)
34. Raj, H., Nathuji, R., Singh, A., England, P.: Resource Management for Isolation Enhanced Cloud Services. In: Proceedings of the 1st ACM Cloud Computing Security Workshop (CCSW'09). pp. 77–84 (2009)
35. Ristenpart, T., Tromer, E., Shacham, H., Savage, S.: Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-Party Compute Clouds. In: CCS'09 (2009)
36. Seaborn, M., Dullien, T.: Exploiting the DRAM rowhammer bug to gain kernel privileges. In: Black Hat (2015)
37. Spreitzer, R., Plos, T.: Cache-Access Pattern Attack on Disaligned AES T-Tables. In: Constructive Side-Channel Analysis and Secure Design (COSADE). pp. 200–214 (2013)
38. Tannous, A., Trostle, J.T., Hassan, M., McLaughlin, S.E., Jaeger, T.: New Side Channels Targeted at Passwords. In: ACSAC. pp. 45–54 (2008)
39. Tromer, E., Osvik, D.A., Shamir, A.: Efficient Cache Attacks on AES, and Countermeasures. Journal of Cryptology 23(1), 37–71 (Jul 2010)
40. Uhsadel, L., Georges, A., Verbauwhede, I.: Exploiting hardware performance counters. In: 5th Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC'08). (2008)
41. Wang, Z., Lee, R.B.: New cache designs for thwarting software cache-based side channel attacks. ACM SIGARCH Computer Architecture News 35(2), 494 (Jun 2007)
42. Wang, Z., Lee, R.B.: A Novel Cache Architecture with Enhanced Performance and Security. In: IEEE/ACM International Symposium on Microarchitecture (MICRO'08). pp. 83–93 (2008)
43. Willems, C., Hund, R., Fobian, A., Felsch, D., Holz, T., Vasudevan, A.: Down to the bare metal: Using processor features for binary analysis. In: ACSAC'12 (2012)
44. Xia, Y., Liu, Y., Chen, H., Zang, B.: CFIMon: Detecting violation of control flow integrity using performance counters. In: DSN'12 (2012)
45. Yarom, Y., Falkner, K.: Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. In: USENIX Security Symposium (2014)
46. Zhang, K., Wang, X.: Peeping Tom in the Neighborhood: Keystroke Eavesdropping on Multi-User Systems. In: USENIX Security Symposium (2009)
47. Zhang, Y., Juels, A., Oprea, A., Reiter, M.K.: HomeAlone: Co-residency Detection in the Cloud via Side-Channel Analysis. In: S&P'11 (2011)
48. Zhang, Y., Juels, A., Reiter, M.K., Ristenpart, T.: Cross-Tenant Side-Channel Attacks in PaaS Clouds. In: CCS'14 (2014)
49. Zhang, Y., Reiter, M.: Düppel: retrofitting commodity operating systems to mitigate cache side channels in the cloud. In: CCS'13 (2013)