Flush+Flush: A Stealthier Last-Level Cache Attack

Daniel Gruss
Graz University of Technology, Austria
daniel.gruss@iaik.tugraz.at

Clémentine Maurice
Technicolor, Rennes, France
Eurecom, Sophia-Antipolis, France
clementine@cmaurice.fr

Klaus Wagner
Graz University of Technology, Austria
k.wagner@student.tugraz.at

Abstract—**Research on cache attacks has shown that CPU caches leak significant information. Recent attacks either use the *Flush+Reload* technique on read-only shared memory or the *Prime+Probe* technique without shared memory, to derive encryption keys or eavesdrop on user input. Efficient counter-measures against these powerful attacks that do not cause a loss of performance are a challenge. In this paper, we use hardware performance counters as a means to detect access-based cache attacks. Indeed, existing attacks cause numerous cache references and cache misses and can subsequently be detected. We propose a new criteria that uses these events for ad-hoc detection.**

These findings motivate the development of a novel attack technique: the *Flush+Flush* attack. The *Flush+Flush* attack only relies on the execution time of the flush instruction, that depends on whether the data is cached or not. Like *Flush+Reload*, it monitors when a process loads read-only shared memory into the CPU cache. However, *Flush+Flush* does not have a reload step, thus causing no cache misses compared to typical *Flush+Reload* and *Prime+Probe* attacks. We show that the significantly lower impact on the hardware performance counters therefore evades detection mechanisms. The *Flush+Flush* attack has a perfor-mance close to state-of-the-art side channels in existing cache attack scenarios, while reducing cache misses significantly below the border of detectability. Our *Flush+Flush* covert channel achieves a transmission rate of 496 KB/s which is 6.7 times faster than any previously published cache covert channel. To the best of our knowledge, this is the first work discussing the stealthiness of cache attacks both from the attacker and the defender perspective.

I. 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 hard-ware 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 [16], detecting cryptographic libraries [22], or keystroke logging [12]. The most recent attacks leverage the multiple cores in CPUs, as well as the inclusiveness of the last-level cache to mount cross-cores attacks, that also work in virtualized environments. These two features are core features of the performance of the current processors, and also the core causes of the interferences that lead to cache attacks. Efficient countermeasures that do not cause a loss of performance are thus a challenge.

Rowhammer is a DRAM vulnerability that causes random bit flips by repeatedly accessing a DRAM row [27]. Attacks exploiting this vulnerability have already been demonstrated to gain root privileges and to evade a sandbox [45], showing the severity of faulting single bits for security. Hardware coun-termeasures have been presented and some of them are now implemented in DDR4 modules, but these countermeasures are hard and slow to deploy, as they either rely on new hardware or on BIOS updates that users are not likely to do.

Hardware performance counters have been proposed re-cently as an OS-level detection mechanism for cache attacks and Rowhammer [7], [15]. Indeed, existing cache attacks cause numerous cache references and cache misses, that can be monitored via special events. As an OS-level detection mechanism, the key idea is that it detects attacks in order to later stop them without causing a loss of performance to the whole system. The evaluation of this mechanism has however not been published yet.

In this paper, we evaluate the use of hardware performance counters as a means of detecting cache attacks. We show that monitoring performance counters, especially cache ref-erences and cache misses of the last-level cache, provides an efficient way to detect of the existing attacks *Flush+Reload* and *Prime+Probe*. Additionally, we propose a new criteria for detection, that uses cache references, cache misses, and instruction TLB performance counters.

We subsequently present the *Flush+Flush* attack that seeks to evade detection. *Flush+Flush* exploits the fact that the execution time of the clflush instruction is shorter if the data 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. Thus, as in the *Flush+Reload* attack, an attacker monitors when another process loads read-only shared memory into the CPU cache. Like *Flush+Reload*, the attack is a cross-core attack and can even be applied in virtualized environments across virtual machine borders. In contrast to existing cache attacks, *Flush+Flush* does not trigger the prefetcher. Thus, it is possible to measure cache misses in some situations where existing cache attacks fail. As there is no reload-step, the *Flush+Flush* attack cannot be detected using known mechanisms, as it only causes no additional cache misses in the attacker process, but only in the benign victim process. Thus, the attack renders proposed detection mechanisms non-effective. We found that in the case of low-frequency events like keystrokes, the *Flush+Flush* attack can be barely distinguished from a process doing nothing. Thus, in contrast to other attacks it is completely stealthy.

Our key contributions are:

- We evaluate the use of a wide range of hardware per-formance counters and propose new detection criteria that detects existing covert and side channels as well as

Rowhammer.

- 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. It provides an improvement over *Flush+Reload* in terms of stealthiness.
- We evaluate the performance of *Flush+Flush* against already known state-of-the-art attacks. We build a covert channel that exceeds state-of-the-art performance, build side-channel attacks such as eavesdropping on user input, and a first-round attack on the OpenSSL T-table-based AES implementation. We show that although existing attacks are more accurate, *Flush+Flush* attacks are more stealthy.

Outline: The remainder of this paper is organized as follows. Section II provides background information on CPU caches, shared memory, and cache attacks. Section III investigates how to leverage hardware performance counters to detect cache attacks. Section IV describes the *Flush+Flush* attack. 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 V, a side-channel attack on keystroke timings in Section VI and on cryptographic algorithms in Section VII. Section VIII discusses required modifications to detection mechanisms and countermeasures to stop our attack. Section IX discusses related work. Section X describes future work. Finally, we conclude in Section XI.

II. BACKGROUND

A. 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. In this level, a part of the physical address is used as a set index. Thus, a physical address is always mapped to the same cache set. The L3 cache is also inclusive of the lower cache levels, which means that all data within the L1 and L2 caches is also present in the L3 cache. To guarantee this property, all data evicted from the L3 must also be evicted from L1 and L2. Due to these two properties of the last-level cache, executing code or accessing data on one core has immediate consequences even for the private caches of the other cores. This is exploited in cache attacks described in Section II-C.

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 a so-called *complex-addressing* function. This function distributes the traffic evenly among the slices and reduces congestion. It is undocumented, but has been reversed-engineered [17], [35], [56]. All address bits are used to determine the slice, excluding the lowest bits that determine the offset in a line. Contrary to slices, sets are directly addressed.

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 [41]. The unprivileged `clflush` instruction evicts a cache line from all the cache hierarchy. However, any program can evict a cache line by accessing a set of addresses (at least as large as the number of ways) in a way to defeat the replacement policy.

B. Shared Memory

Operating systems and hypervisors instrument shared memory to reduce the overall physical memory utilization. Shared libraries, which are typically used by several programs, 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 performs similar optimizations whenever the same file is mapped into memory more than once. This is the case when forking a process, when starting a process twice, when using `mmap` or `dlopen`. All cases result in a memory region shared with all other processes mapping the same file.

On personal computers, smartphones and private cloud systems, another form of shared memory can be found, namely content-based page deduplication. The hypervisor or operating system scans the physical memory for bytewise identical pages. Identical pages are remapped to the same physical page, while the other page is marked as free. This technique can lower the physical memory utilization of as system significantly. However, memory might be shared between completely unrelated and possibly sandboxed processes, and even between processes running in different virtual machines.

C. Cache Attacks and Rowhammer

Cache attacks exploit timing differences caused by the lower latency of CPU caches compared to physical memory. The possibility of exploiting these timing differences was first discovered by Kocher [28] and Kelsey et al. [25]. Practical attacks have focused on side channels on cryptographic algorithms and covert channels. Access-driven cache attacks can be devised in two types: *Prime+Probe* [39], [40], [48] and *Flush+Reload* [13], [55].

In *Prime+Probe* attacks, the attacker fills the cache, then waits for the victim to evict some cache sets. The attacker reads data again and determines which sets were evicted. The time taken by the accesses to the cache set is proportional to the number of cache ways that have been occupied by other processes. The challenge for this type of attack is the granularity, *i.e.*, the ability to target a specific set without any shared memory. Indeed, modern processors have a physically indexed last-level cache, use complex addressing, and undocumented replacement policies. Cross-VM side-channel attacks [19], [32] and covert channels [36] that tackle these challenges have been presented in the last year. Oren et al. [38] showed that a *Prime+Probe* cache attack can be launched from within sandboxed JavaScript in a browser, allowing a remote attacker can eavesdrop on network traffic statistics or mouse movements through a website.

Flush+Reload attacks work on a single cache line granularity. It works by frequently flushing a cache line using the `clflush` instruction. By measuring the time it takes to reload the data, the attacker determines whether a targeted address has been reloaded by another process in the meantime. *Flush+Reload* exploits the availability of shared memory and especially shared libraries between the attacker and the victim program. Gruss et al. [12] have shown that a variant of *Flush+Reload* without the `clflush` instruction is possible without a significant loss in accuracy. Applications of *Flush+Reload* have been shown to be reliable and powerful, mainly to attack cryptographic algorithms [14], [22], [23], [59].

Recent cross-core *Prime+Probe* and *Flush+Reload* attacks exploit two properties of modern CPU caches and the operating system. First, the last-level cache is shared among all cores and thus several processes work simultaneously on the same cache. Second, the last-level cache is inclusive to the lower cache levels. Thus an attacker can evict data not only from the last-level cache but also from the local lower levels of the other CPU cores.

Rowhammer is not a typical cache attack but a DRAM vulnerability that causes random bit flips by repeatedly accessing a DRAM row [27]. 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 [45]. The original attack used the `clflush` instruction to flush data from the cache, but it has been showed that it is possible to trigger bit flips without this instruction, by performing cache eviction through memory access patterns [11]. Both techniques cause a significant number of accesses to the cache, that resemble to a cache attack.

III. DETECTING CACHE ATTACKS WITH HARDWARE PERFORMANCE COUNTERS

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 recently proposed to detect Rowhammer and the *Flush+Reload* attack [7], [15].

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 [1]. The actual accesses to the model specific registers are performed in the kernel. The `perf_event_open` syscall interface can be used without root privileges by any process to monitor its own influence on the performance counters or the influence of child processes. A system service could run on a higher privilege level and thus use performance counters without restrictions. During our tests we ran the performance monitoring on root privileges to avoid any restrictions. Some performance events allow monitoring software-based events, kernel events or interrupts. We analyzed all 23 performance events available on our system of type `PERF_TYPE_HARDWARE` for generic hardware events and `PERF_TYPE_HW_CACHE` for specific cache events. We

TABLE I.   LIST OF HARDWARE PERFORMANCE EVENTS WE USE.

Name	Description
BPU_RA	Branch prediction unit read accesses
BPU_RM	Branch prediction unit read misses
BRANCH_INSTRUCTIONS	Retired branch instructions
BRANCH_MISSES	Brach mispredictions
BUS_CYCLES	Bus cycles
CACHE_MISSES	Last-level cache misses
CACHE_REFERENCES	Last-level cache accesses
UNC_CBO_CACHE_LOOKUP	System-wide C-Box last-level events including `clflush` (total over all slices)
CPU_CYCLES	CPU cycles
DTLB_RA	Data TLB read accesses
DTLB_RM	Data TLB read misses
DTLB_WA	Data TLB write accesses
DTLB_WM	Data TLB read misses
INSTRUCTIONS	Retired instructions
ITLB_RA	Instruction TLB read accesses
ITLB_RM	Instruction TLB write accesses
L1D_RA	L1 data cache read accesses
L1D_RM	L1 data cache read misses
L1D_WA	L1 data cache write accesses
L1D_WM	L1 data cache write misses
L1I_RM	L1 instruction cache read misses
LL_RA	Last-level cache read accesses
LL_WA	Last-level cache write accesses
REF_CPU_CYCLES	CPU cycles without scaling

observe that apart from a few hardware performance counters, most do not allow distinguishing between cache attack processes and benign processes. We additionally analyzed the uncore performance monitoring units called C-Box, with one C-Box per cache slice. They allow monitoring an event called `UNC_CBO_CACHE_LOOKUP`, that counts lookups to a 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. A list of all events we use in our evaluation can be found in Table I.

We evaluated the 24 performance counters for the following scenarios:

1) Idle: an idle system,
2) Firefox: a normal and benign activity with a user scrolling down the Twitter search feed for the hashtag `#rowhammer` in Firefox,
3) OpenTTD: a user playing a game
4) stress -m 1: a benign but memory intensive activity by executing `stress -m 1`,
5) stress -c 1: a benign but CPU intensive activity by executing `stress -c 1`,
6) stress -i 1: a benign but I/O intensive activity by executing `stress -i 1`,
7) *Flush+Reload*: a *Flush+Reload* side-channel attack on the GTK library to spy on keystroke events,
8) Rowhammer: a Rowhammer attack.

A good detection mechanism classifies as benign the scenarios 1 to 6 and as attacks the scenarios 7 and 8.

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 only a low pressure on the instruction TLB. Benign software will rather cause a high pressure on the instruction TLB as well. Therefore, we use the instruction TLB performance counters a normalization factor for the other performance counters.

Test	sleep 135	Firefox	OpenTTD	stress -m 1	stress -c 1	stress -i 1	Flush+Reload	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_INSTRUCTIONS	4.35	14.62	74.73	92.62	6 094 264.03	3.23	127 605.71	23 834.59
BRANCH_MISSES	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_REFERENCES	0.40	0.98	6.84	61.05	0.31	2.28	693.92	13 800.01
UNC_CBO_CACHE_LOOKUP	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_CYCLES	157.70	69.69	445.89	1 872.05	405 922.02	223.08	1 098 534.32	3 542 570.00

Table II shows a comparison of performance counters for the 8 different scenarios normalized to the number of instruction TLB events. Not all cache events are suitable for detection. Indeed, 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 in previous work by Chiappetta et al. [7] 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. It is higher in case of a high CPU core utilization and a low number of instruction TLB and L1 misses. It has a high count in the case of cache attacks, but also for the `stress -c` tool. Due to the possibility of false positives, we will not consider `REF_CPU_CYCLES` in our evaluation.

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,
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 only on the `CACHE_MISSES` and `CACHE_REFERENCES` events.

We define that a process is considered as malicious if more than 1 cache miss or 1 cache reference per instruction TLB event per second is observed. That is, the attack is detected if the rate

$$\frac{C_{\texttt{CACHE\_MISSES}}}{C_{Instruction\ TLB\ event}} \geq 1,$$

with C the value of the corresponding performance counter, or the rate

$$\frac{C_{\texttt{CACHE\_REFERENCES}}}{C_{Instruction\ TLB\ event}} \geq 1.$$

The threshold for the cache reference and cache hit rate of 1 per second is more than double the highest value of any benign process we tested and only a fifth of the lowest value we measured for *Flush+Reload* in both cases. 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.

IV. THE *Flush+Flush* ATTACK

In this section, we present an attack we called *Flush+Flush* that is a more stealthy alternative to existing cache attacks, and that defeats detection with hardware performance counters. The *Flush+Flush* attack is a variant of the *Flush+Reload* attack. It is applicable in multi-core and virtualized environments if read-only shared memory with the victim process can be acquired.

Our attack builds upon the observation that the `clflush` instruction leaks information on the state of the cache. Indeed, 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. Furthermore, if the eviction is on a remote core it has a higher minimum execution time in case of a cache hit. Thus, an attacker can derive whether a memory access is served from the CPU cache and a process can derive information on which core it runs on.

Listing 1 shows an implementation of the *Flush+Flush* attack. The attack consists of only one phase, that is executed

```
1  while (1)
2  {
3    mfence();
4    size_t time = rdtsc();
5    mfence();
6    clflush(target_addr);
7    mfence();
8    size_t delta = rdtsc() - time;
9    mfence();
10   // report cache hit/miss
11   report(delta);
12   size_t count = YIELD_COUNT;
13   while (count--) sched_yield();
14 }
```

Listing 1.   *Flush+Flush* implementation in C.

TABLE III.   EXPERIMENTAL SETUPS.

CPU	Microarchitecture	Cores	LLC associativity
i5-2540M	Sandy Bridge	2	12
i5-3320M	Ivy Bridge	2	12
i7-4790	Haswell	4	16

Fig. 1.   Comparison of memory access and `clflush` instruction on cached and uncached memory on different CPU architectures

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. At the end of an attack round, the program optionally yields `YIELD_COUNT` times in order to lower the system utilization and waits for the second process to perform some memory accesses.

The measurement is done using the `rdtsc` instruction that provides a sub-nanosecond resolution timestamp. Modern processors support out-of-order execution, which does not guarantee that the instructions are executed in the order in which they are written. It is thus essential to surround `rdtsc` with `mfence` instructions for the measurement, as `clflush` is only ordered by `mfence`, but not by any other means.

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 described in Table III. 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 access time for cache hits. The difference is enough to be observed by an attacker. We discuss this timing difference and its implications in Section IX-A.

The *Flush+Flush* attack inherently has a lower accuracy than the *Flush+Reload* technique due to the lower timing difference between a hit and a miss. New cache attacks on implementations of cryptographic algorithms will thus yield a better performance with *Flush+Reload*. On the other hand, the *Flush+Flush* attack also has some clear advantages compared to the *Flush+Reload* technique. First, the reload-step of the *Flush+Reload* attack can trigger the prefetcher and thus destroy measurements by fetching data into the cache. This
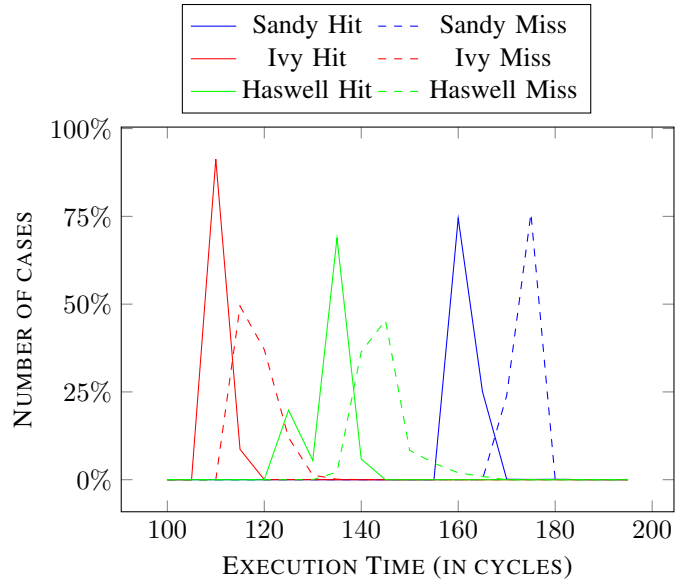
is the case especially when monitoring more than one address within a physical page [12]. Second, the `clflush` instruction typically takes between 100 and 200 cycles. The reload-step of the *Flush+Reload* attack adds at least 250 cycles when causing a cache miss. Thus, one round of the *Flush+Flush* attack is significantly faster than one round of the *Flush+Reload* attack. Third, recently proposed detection mechanisms measure cache references and cache misses. The detection mechanism we described in Section III also uses the `CACHE_REFERENCES` and `CACHE_MISSES` performance counters. However, the *Flush+Flush* attack does not influence these performance counters significantly.

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.

V.   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. The cache covert channel is established on an address in a shared library that is used by both programs. We show how the two processes can communicate through this read-only shared memory by means of a cache covert channel and how it can be implemented using the *Flush+Flush*, *Flush+Reload*, and *Prime+Probe* technique. Finally, we compare the performance and the detectability of the three implementations. In the remainder of the paper, all the experiments are performed using the Haswell CPU described in Table III.

Fig. 2. Format of a data packet in our covert channel framework.
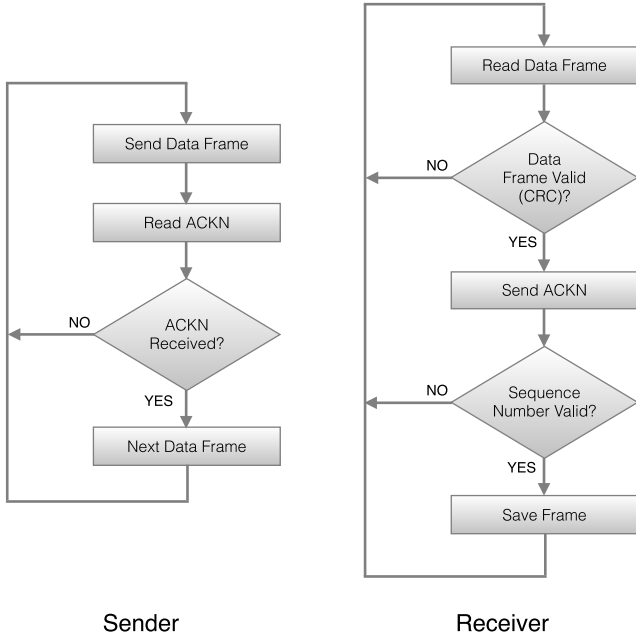

Fig. 3. Sender and receiver process control flow chart as implemented in our covert channel framework.

A. 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 [32], [36] 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 CRC-16 checksum is used to detect corruption. If a received packet is valid, a byte is used to acknowledge the sequence number. Otherwise the packet is retransmitted. The format of a packet is shown in Figure 2. The transmission algorithms are shown in Figure 3.

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-
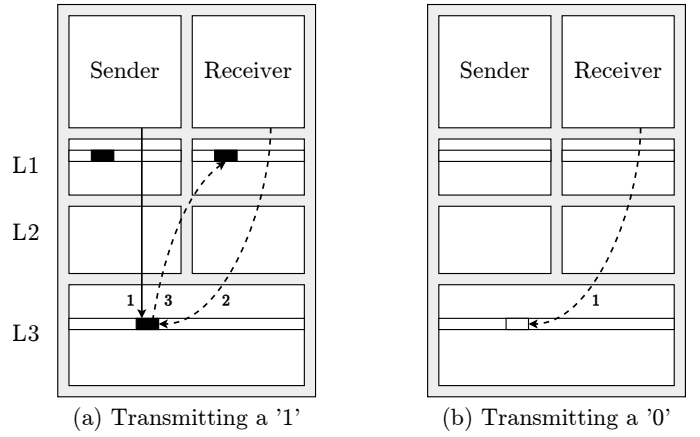

(a) Transmitting a '1'   (b) Transmitting a '0'

Fig. 4. Illustration of the *Flush+Flush* covert channel.

correction cannot be omitted.

B. Covert Channel Implementations

We first implemented the *Flush+Reload* covert channel. In this implementation, the sender and the receiver access the same shared library and run *Flush+Reload* attacks on a fixed set of offsets in the library to communicate. The sender accesses the memory location to transmit a 1, and stays idle to transmit a 0. The receiver monitors the shared cache line to receive the bits. It measures the time taken to reload the line in order to infer the bit. If the access is fast, it means the line is cached, and a 1 is received. If the access is slow, it means the line is served from the DRAM, and a 0 is received. The receiver then flushes the line for the transmission of the next bit. The bits retrieved are then parsed as a data frame according to the transmission protocol. The sender monitors cache hits on some memory locations using *Flush+Reload* too, to receive packet acknowledgments.

The second implementation is the *Flush+Flush* covert channel, illustrated by Figure 4. It works similarly to the *Flush+Reload* covert channel. As in *Flush+Reload*, the sender and the receiver access the same shared library. To transmit a 1 (Figure 4-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 inclusivity, 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 4-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. [32], Oren et al. [38], and Maurice et al. [36]. The sender transmits a 1 bit by priming a cache set. The receiver probes the same cache set. On our Haswell CPU this requires 16 memory accesses. By observing the access time, the receiver derives

what the other process did: a long access means a 1, whereas a short access means a 0. 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 [35] on physical addresses. This avoids the possibility of errors introduced by the timing-based eviction set computation. Second, we map the shared library into our address space to determine the physical address to attack. Yet, it is never accessed and unmapped even before the *Prime+Probe* attack is started. This adjustment is not required to perform the attack and does not influence it in any way. We assume that sender and receiver have agreed on the cache sets in a preprocessing step. This is practical even for a timing-based approach.

C. Performance Evaluation

Table IV 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 we achieve a transmission rate of 496 KB/s. This is significantly faster than previously published cache-based covert channels. At the same time the effective error rate is only 0.84%. The *Flush+Reload* covert channel also achieved the best 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 covert channel 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 $32 \cdot 8 \cdot 4096 \cdot 16 = 16\,\text{GB}$ 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 we achieve a transmission rate of 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* and *Flush+Reload* covert channels at 28 bytes packet size achieve a transmission rate significantly higher than the other state-of-the-art covert channels. Especially, the *Flush+Flush* covert channel is 6.7 times as fast as the fastest covert channel to date by Liu et al. [32] at a comparable error rate. However, we perform our attack on a recent Haswell CPU that has a cache replacement policy that is different to the one of older CPUs, such as Sandy Bridge. While the additional instructions executed add a small performance penalty, the faster CPU increases the performances slightly. However, compared our own *Prime+Probe* covert channel, *Flush+Flush* is 7.3 times faster.

D. Detectability

To be stealthy, both the sender and the receiver processes must be classified as benign. As shown in Table IV, the *Flush+Flush* attack with a packet size of 4 bytes is the only one to be classified benign for both sender and receiver process by our detection mechanism described in Section III. The *Flush+Flush* receiver with a 28-byte packet size is close to the detection threshold, but still classified as malicious. However, at a 5-byte packet size it is slightly below the detection threshold and thus classified as benign.

TABLE V. COMPARISON OF THE ACCURACY OF CACHE ATTACKS ON USER INPUT.

Attack Technique	Correct Detections	False Positives
Flush+Reload	961	3
Flush+Flush	747	73
Prime+Probe	—	—

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.

For all covert channels, an adversary can choose to reduce the transmission rate in order to be stealthier. This is achieved best with the *Flush+Flush* attack that is stealthier than *Flush+Reload* for a similar capacity, in the case of 4-byte packets. There is thus no advantage in reducing further the transmission rate of *Flush+Reload* compared to using *Flush+Flush*.

VI. SIDE-CHANNEL ATTACK ON USER INPUT

In this section, we consider an attack scenario where an unprivileged attacker eavesdrops on keystroke timings by performing a cache attack on a shared library.

A. Attack Implementation Using Flush+Flush

We attack an address in the GTK library `libgtk-3.so.0.1400.14` found by a Cache Template Attack [12]. The GTK library is the default user-interface framework on many Linux systems. The address we attack reacts on every keystroke using the *Flush+Reload* attack.

The *Flush+Flush* implementation is similar to the *Flush+Reload* implementation. The spy program loads the shared library. The spy constantly flushes the address, and derives when a keystroke occurred, based on the execution time of the `clflush` instruction.

B. Performance Evaluation

We compare the three attack techniques *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. Table V shows the results of the attack.

We see that *Flush+Reload* performs best, with 96.1% correctly detected keystrokes. At the same time, we measured only 3 false positives. This allows direct logging of keystroke timings.

Flush+Flush performs notably well, with 74.7% correctly detected keystrokes. However, we also find 73 false positives. That is more than one false positive in 2 seconds. This makes a practical attack much harder, but not completely impossible.

While *Prime+Probe* performs worse than *Flush+Flush* in the covert channel scenario, it works even worse for low

Attack Technique	Packet Size	Capacity	Error Rate	Sender References	Sender Misses	Sender Classification	Receiver References	Receiver Misses	Receiver Classification
Flush+Flush	28	496 KB/s	0.84%	1809.26	96.66	Malicious	1.75	1.25	Malicious
Flush+Reload	28	298 KB/s	0.00%	526.14	56.09	Malicious	110.52	59.16	Malicious
Flush+Reload	5	132 KB/s	0.01%	6.19	3.20	Malicious	45.88	44.77	Malicious
Flush+Flush	5	95 KB/s	0.56%	425.99	418.27	Malicious	0.98	0.95	Benign
Prime+Probe	5	67 KB/s	0.36%	48.96	31.81	Malicious	4.64	4.45	Malicious
Flush+Reload	4	54 KB/s	0.00%	0.86	0.84	Benign	2.74	1.25	Malicious
Flush+Flush	4	52 KB/s	1.00%	0.06	0.05	Benign	0.59	0.59	Benign
Prime+Probe	4	34 KB/s	0.04%	55.57	32.66	Malicious	5.23	5.01	Malicious

Test	Cache References	Cache Misses	Classification
Flush+Reload	5.140	5.138	Malicious
Flush+Flush	0.002	0.000	Benign

frequency cache side-channel attacks. We could not distinguish whether a cache hit was a correctly detected keystroke or one of many false positive cache hits. The results of this attack are thus not exploitable. Indeed, if the attacker is eavesdropping on non-repeatable low frequency events, a single false positive cache hit during the measurement destroys the whole measurement. In case of low frequency events, this is in the range of milliseconds to seconds. Moreover, as the keystroke is a user input, the attack is non-repeatable in contrast to attacker-controlled measurements such as encryptions.

The reason for the huge gap in performance between *Flush+Flush* and *Prime+Probe* in this scenario lies in a fundamental difference between these two attacks. *Flush+Flush* operates at the granularity of a single line, while *Prime+Probe* focuses on a cache set. By accessing the whole cache set, we do not measure the timing difference caused by a single cache hit and miss respectively, but instead the timing difference caused by several cache hits and several cache misses respectively. Thus it is a form of amplification of the timing difference used to transmit a bit. Without this amplification *Flush+Flush* performs better, which is the case in the side-channel scenario. This is true for all side-channel scenarios, as normal programs will rarely access several addresses in the same cache set dependent on the same secret event.

C. 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 VI 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 and thus a comparison of cache references or cache misses is not meaningful. *Flush+Reload* generates many cache misses, whereas *Flush+Flush* causes almost no cache misses at all.

Flush+Reload yields the highest accuracy in this side-channel attack, but it is easily detected. On the other hand, *Flush+Flush* is a viable and stealthy alternative to the

Flush+Reload attack as it is not classified as malicious by our mechanism presented in Section III.

VII. 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. Cryptographic algorithms have been the main focus of cache side-channel attacks in the past. Although appropriate countermeasures are already implemented in the case of AES [18], [24], [30], [43], we attack the OpenSSL T-Table-based AES implementation that is known to be susceptible to cache attacks [2], [3], [5], [13], [20], [21], [39], [52]. We compare the three attack techniques in a first round attack on AES, that can be implemented similarly using the different attack techniques. We use the AES T-table implementation from the OpenSSL library version 1.0.2 [37]. This AES implementation is disabled by default for security reason, but it is still contained in the source code for the purpose of comparing new and existing cache 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 $0 \leq i < 16$. These accesses are cached and an attacker is able to detect which accesses were made. Thus an attacker can derive possible values for $p_i \oplus k_i$ and derive possible key-byte values k_i in case p_i is known.

A. Attack Implementation Using Flush+Flush

As we have seen in the other scenarios, the implementation of *Flush+Flush* is very similar to *Flush+Reload*. This is also the case here. We perform a chosen-plaintext attack. Thus, the attacker triggers an encryption, choosing p_i while all p_j with $i \neq j$ are random. Not every value for p_i and p_j has to be tested in the first round attack. One cache line holds 16 T-Table entries. Thus, we can set the last bits of every p_i and p_j to zero, reducing the search space even further. 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.
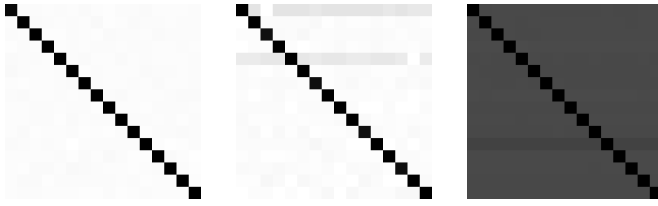
Fig. 5. Comparison of Cache Templates (address range of the first T-table) generated using *Flush+Reload* (left), *Flush+Flush* (middle), and *Prime+Probe* (right). In all cases $k_0 = $ `0x00`.

TABLE VII. NUMBER OF ENCRYPTIONS NECESSARY TO RELIABLY GUESS THE UPPER 4 BITS OF A KEY BYTE CORRECTLY.

Attack Technique	Number of Encryptions
Flush+Reload	250
Flush+Flush	400
Prime+Probe	4 800

B. Performance Evaluation

Figure 5 shows a comparison of cache templates generated using *Flush+Reload*, *Flush+Flush*, and *Prime+Probe*. The traces were generated using $1\,000\,000$ encryptions to create a visible pattern in all three cases. It is comparable to similar cache templates in published literature [12], [39], [46]. Table VII shows how many encryptions are necessary to determine the upper 4 bits correctly. Therefore, 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 1.6 times as many encryptions as *Flush+Reload*, but 12 times less than *Prime+Probe* to achieve the same accuracy.

C. Detectability

To evaluate the detectability, we again monitored the cache reference and cache miss events of the spy process, and compared the three cache attacks with each other, performing the same number of encryptions. Table VIII shows that *Prime+Probe* causes significantly more cache references and cache misses than the other attacks. *Flush+Reload* causes around 30% more cache references than *Flush+Flush*. *Flush+Flush* causes almost no cache misses at all.

Applying the detection mechanism from Section III, *Prime+Probe* and *Flush+Reload* can clearly be detected. *Flush+Flush* can only be detected based on the number of cache references, but not the number of cache misses. However, even for the cache references it is close to the detection threshold and an attacker could easily avoid detection by stretching the attack over a slightly longer period of time. Furthermore, the *Flush+Flush* attack took only 163 seconds whereas *Flush+Reload* took 215 seconds and *Prime+Probe* 234 seconds for the identical attack. As the detection measures cache references and cache misses per instruction TLB event per second, this clearly helps the detection of *Flush+Flush* here. However, if an attacker would slow *Flush+Flush* down to the same speed as *Prime+Probe*, *Flush+Flush* would remain undetected. Thus our measurements show that *Flush+Flush* is indeed a stealthy and fast alternative to *Flush+Reload* in this side-channel attack scenario.
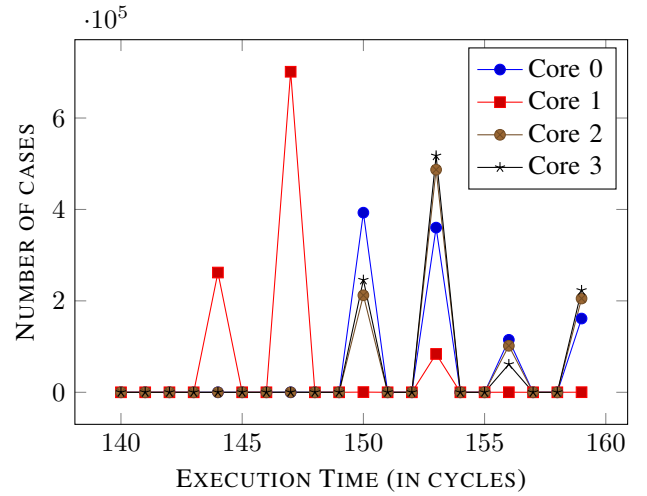

Fig. 6. Excerpt of the histogram when flushing an address that maps to core slice 1 from different cores. The lower execution time on core 1 shows the address maps to slice 1.

VIII. DISCUSSION

In this section, we detail some other findings following our study of how the `clflush` instruction behaves. We also detail how to evade detection by performance counters and possible countermeasures to *Flush+Flush* attacks.

A. Using `clflush` to Detect the Core on which a Process Runs

Apart from building cache covert channels and cache side-channel attacks we can also use the *Flush+Flush* attack to determine on which CPU core a process is running. Indeed, physical addresses statically map to cache slices using a complex addressing function. Each slice is interconnected by a ring bus, so that each core can access every slice. However, each core has a direct access to its local slice. We can thus measure a difference in timing with the `clflush` instruction when accessing the local slice rather than a remote slice. Figure 6 shows an excerpt of the execution time histogram when the process runs on one of the CPU cores. The access to an address that maps to slice 1 takes less cycles when the program runs on core 1 (for which slice 1 is the local slice), than when the program runs on any other core. If we determine the physical address of a memory access on a local slice, we can use the complex addressing function [35] to determine on which core the process runs.

This can be exploited to optimize cache covert channels. While the attack first runs on the last-level cache, the two processes can communicate to each other which core they are running on. If both processes run on the same system-assigned physical CPU core they either run on different virtual hyperthreading cores or time-share a single physical core. In either case they can then switch to a covert channel on the L1 or L2 cache instead of the last-level cache. Such a covert channel would have a higher performance as long as both processes remain on the same physical CPU core.

The same information can also be exploited to improve the Rowhammer attack [11]. The Rowhammer attack induces

Attack Technique	Cache References	Cache Misses	Execution Time in s	References (Normalized)	Misses (Normalized)	Classification
Flush+Reload	1 024 035 376	19 284 602	215	2 513.43	47.33	Malicious
Prime+Probe	4 221 994 794	294 897 508	234	1 099.63	76.81	Malicious
Flush+Flush	768 077 159	1 741	163	1.40	0.00	Malicious

random bit flips in DRAM modules by performing accesses to the same memory location with a high frequency. However, as modern CPUs have large caches, frequent accesses to the same memory location will typically be served from the cache. The `clflush` instruction is used to force the memory accesses to be served from DRAM on each access. We observed that running `clflush` on a local slice lowers the execution time of each Rowhammer loop round by a few cycles. As Gruss et al. [11] noted, the probability of bit flips increases as the execution time lowers, thus we can leverage the information about which core the program is executed on to improve the attack.

A similar timing difference also occurs upon memory accesses that are served from the local or a remote slice respectively. The reason for this timing difference is that the local cache slice has a direct connection to the CPU core while remote cache slices are connected via a ring bus. Thus the data has to travel through the ring bus before it arrives at the CPU core we are running on. 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 in this case.

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.

B. Evading Detection by Performance Counters

In addition to *Flush+Flush*, existing attacks can be modified to evade detection by performance counters. The goal is to change the patterns of cache references and cache misses to make them look like benign applications. This can be done in two ways.

First, it is possible to reduce the number of cache references and cache misses over time. For covert channels, this reduces the transmission rate. Our experiments show that *Prime+Probe* is always detected as malicious, even with a severely lowered transmission rate. For a similarly low transmission rate, *Flush+Flush* is already stealthier than *Flush+Reload*. Moreover, reducing cache references and misses makes side-channel attacks more difficult or impossible, since they need fine-grained measurements. In contrast, *Flush+Flush* attack on user input data can already be performed in a high frequency without detection.

Second, it is possible to increase the number of performance events that are used for normalization, in our case the instruction TLB events. Cache attacks like *Flush+Reload* can thus be modified to induce the same behavior, by using

additional memory accesses to shape the instruction TLB and cache behavior to a more ambiguous pattern.

However, these techniques cannot be applied to Rowhammer. Both, additional memory accesses and introducing delays reduce the memory access frequency too much to trigger bit flips.

C. Countermeasures to Flush+Flush Attacks

Given the stealthiness of the *Flush+Flush* attack, the possibility to improve covert channels by detecting on which CPU core a process runs on and the possibility to improve the Rowhammer attack using `clflush`, we suggest modifying the `clflush` instruction to counter these attacks. As shown in Figure 1, the difference in the execution time of `clflush` is 3 cycles on average on our test system. This is negligible in terms of performance. Furthermore, the `clflush` instruction is used only in rare situations. Most software does not use the `clflush` instruction at all. We propose making `clflush` a constant-time instruction. That is, if the CPU executes the `clflush` instruction and the address is found in the local last-level cache slice, it should add a cycle penalty to remove any observable timing difference. This would prevent the *Flush+Flush* attack completely, as well as information leakage on cache slices and CPU cores.

The stealthiness of *Flush+Flush*, compared to *Flush+Reload*, is due to the absence of a reload phase which causes less cache misses and cache references. With *Flush+Flush* a process transmits a bit with value 1 with a single memory access and a bit with value 0 without any memory access. *Prime+Probe* requires comparably many memory accesses to prime the cache set and to probe the cache set. This causes numerous cache references and cache misses. Yet, *Flush+Flush* still relies on an eviction phase, and thus still causes the other process to trigger cache misses. However, these cache misses are not trivial to detect at an application-level. A 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 solution called *informing loads* has been proposed by Kong et al. [29], however it is hardware-based and needs a change in the instruction set. This could also be implemented without hardware modifications by enabling the `rdtsc` instruction only in privileged mode as can be done by seccomp on Linux systems [33] since Linux 2.6.26 in 2008. Fogh [9] recently proposed to subsequently simulate the `rdtsc` in an interrupt handler, degrading the accuracy of measurements far enough to make cache attacks significantly harder.

Finally, making the `clflush` instruction privileged would prevent the attack as well. However, this would require changes

in hardware and could not be implemented in commodity systems.

IX. Related work

A. Detecting and Preventing Cache Attacks

While most of the contributions in terms of countermeasures focus on the prevention of attacks, a few of them aim at detection.

1) Detection: Zhang et al. [58] 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. The goal of HomeAlone is different from ours, as it does not explicitly seek to detect cache attacks but rather co-resident virtual machines. In contrast with HomeAlone, using performance counters to monitor cache attacks is less fine-grained, *i.e.*, we do not monitor individual cache sets. However, as HomeAlone uses a cache attack in a defensive way, the monitoring itself has a footprint on the cache usage, whereas the use of performance counters has not. Thus using performance counters does not cause a performance penalty to legitimate applications, and is not detectable by an attacker. Cache Template Attacks [12] can also be used to detect attacks on shared libraries and binaries as a user. By performing a systematic *Flush+Reload* attack on a specified address range attacks are detected reliably. However, such a permanent scan increases the system load and can only detect attacks in a small address range within a reasonable response time.

Using hardware performance counters has been proposed recently as a detection mechanism by Herath and Fogh [15] and Chiappetta et al. [7]. Herath and Fogh [15] proposed to monitor cache misses to detect *Flush+Reload* attacks and Rowhammer. An operating system service monitors the number of cache misses, and if it measures a peak, it will interrupt the process causing the cache activity. The operating system can then take further action to stop the attack, such as terminating the program causing the excessive cache activity. Simultaneously to our work, Chiappetta et al. [7] 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. Moreover, the correlation-based approach is not suited to detect Rowhammer. In contrast, we build an ad-hoc detection mechanism based on the ideas by Herath and Fogh [15], searching for performance counter values that do not occur in benign software. Additionally, we extend this approach by proposing a new detection criteria less likely to cause false positives.

Finally, Fogh [9] proposed to make the `rdtsc` instruction privileged to slow down malicious and benign software using the `rdtsc` instruction. Likewise, it can be done to prevent Rowhammer attacks and to detect cache attacks that use the `rdtsc` instruction.

2) Prevention: Countermeasures against cache attacks can be envisioned 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 [31], [50], [51] or altering the prefetcher policy [10]. These solutions all necessitate changes in the hardware or the instruction set and thus are not applicable in the near future, in contrast to system or application level changes. At the system level, page coloring provides cache isolation in software [26], [42]. Other works proposed a more relaxed isolation like Düppel [60] that repeatedly cleans caches that are time-shared, e.g., the L1 cache. However, these solutions cause performance issues, as they prevent an optimal use of the cache. Application-level countermeasures like [6] seek to find the source of information leakage and patch it. Leaks can be found with tools like Cache Template Attacks [12]. However, application-level countermeasures are bounded and cannot prevent every 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, and let the application or the system decide what action to take.

B. Usage of Hardware Performance Counters in Security

Hardware performance counters are traditionally used for performance monitoring. They have also been used in a few security scenarios. In defensive cases, they are used for the detection of anomalous behaviors, with cases such as malware detection [8], integrity checking of programs [34], control flow integrity [54], and binary analysis [53]. In offensive scenarios, Uhsadel et al. [49] used performance counters to profile the cache and derive a side-channel attack against AES. Bhattacharya and Mukhopadhyay [4] exploited the performance counters to profile the branch misses to attack RSA. Performance counters have also been used by Maurice et al. [35] to reverse-engineer the complex addressing function of the last-level cache of modern Intel CPUs.

C. 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 [32], [36]. *Flush+Reload* has been used to derive side-channels attacks [55], 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*. We thus provide an evaluation and a fair comparison between these different covert channels, in the same hardware setup and with the same protocol.

D. Side-Channel Attacks on User Inputs

Section VI 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. First, an attacker could install the `xinput` tool and use it to build a keylogger. The keylogger itself does not require root access in this case, however, installing `xinput` does require root access. The second option is to use the `/dev/input/event*` devices. Here the attacker could manipulate the access rights so that the keylogger again does not require root access. However, manipulating the access rights requires root access.

Software-based side-channel attacks have already proven to be a reliable way to eavesdrop on user input. Attacks either exploit differences in the execution time [47], peaks in CPU and cache activity graphs [44], or exploit system services to guess user input in a targeted process [57]. Zhang et al. [57] instrumented the `procfs` system on Linux to measure inter-keystroke timings. Subsequently, they were able to derive key sequences from inter-keystroke timings. Oren et al. [38] demonstrated that an attacker can use the *Prime+Probe* attack even from sandboxed JavaScript inside a browser to derive user activities, such as mouse movements. Gruss et al. [12] showed that auto-generated *Flush+Reload* attacks can be used to measure keystroke timings as well as identifying keys to a certain degree with high accuracy.

X. FUTURE WORK

We found that the performance of *Prime+Probe* attacks on Haswell CPUs is worse than on Sandy Bridge or even older CPUs. We think the reason for this lies in the new cache replacement policy that has been introduced with the Ivy Bridge architecture and is according to our measurements very similar to the one used in Haswell CPUs. This new cache replacement policy is a quad-age LRU algorithm combined with bimodal insertion policy [41]. We assume that this is the reason why LRU-like cache set priming that has worked on older CPUs does not work on more recent CPUs anymore. Gruss et al. [11] have presented different ways to find and implement good eviction on more modern CPUs. Although we have tried both their eviction strategy and the one by Liu et al. [32], we were not able to implement successful side-channel attacks on low frequency events using their LRU-like cache eviction strategy. Thus, we consider implementing a successful *Prime+Probe* attack on low frequency events on Ivy Bridge and Haswell future work.

XI. CONCLUSION

In this paper, we investigated the use of hardware performance counters to detect cache attacks. We found that existing cache attacks can be detected by monitoring cache references and cache misses, and we introduce a new criteria for detection. This motivates the introduction of the *Flush+Flush* attack, a novel cache attack that evades known detection mechanisms. We compared the *Flush+Flush* attack to other common cache attack techniques. Our results show that *Flush+Flush* attack is a viable alternative if detection mechanisms need to be evaded. Our *Flush+Flush* covert channel is the fastest cache covert channel published to date with a transmission rate of 496 KB/s which is 6.7 times faster than any previously published cache

covert channel. In all scenarios we found *Flush+Flush* to be stealthier than other cache attacks. To the best of our knowledge, this is the first work to draw the attention on the detectability of cache attacks. Indeed, this aspect has not been treated from the attacker perspective. Similarly, existing countermeasures focus predominantly on the prevention of cache attacks rather than on their detection. We expect our work to pave the way of future research in this new direction.

Moreover, while *Flush+Flush* attack is 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 software and does not introduce any interface changes. Thus, it is an effective countermeasure that should be implemented. Commodity hardware can make the `rdtsc` instruction privileged to prevent nanosecond-accurate measurements.

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.

REFERENCES

[1] "Linux man page for perf_event_open(2)," http://man7.org/linux/man-pages/man2/perf_event_open.2.html.

[2] O. Acıiçmez and ç. K. Koç, "Trace-Driven Cache Attacks on AES (Short Paper)," in *Proceedings of the 8th international conference on Information and Communications Security*, 2006, pp. 112–121.

[3] D. J. Bernstein, "Cache-timing attacks on AES," Department of Mathematics, Statistics, and Computer Science, University of Illinois at Chicago, Tech. Rep., 2005.

[4] S. Bhattacharya and D. Mukhopadhyay, "Who watches the watchmen?: Utilizing Performance Monitors for Compromising keys of RSA on Intel Platforms," *Cryptology ePrint Archive, Report 2015/621*, 2015.

[5] A. Bogdanov, T. Eisenbarth, C. Paar, and M. Wienecke, "Differential cache-collision timing attacks on AES with applications to embedded cpus," in *CT-RSA*, 2010, pp. 235–251.

[6] E. Brickell, G. Graunke, M. Neve, and J.-P. Seifert, "Software mitigations to hedge AES against cache-based software side channel vulnerabilities," *Cryptology ePrint Archive, Report 2006/052*, 2006.

[7] M. Chiappetta, E. Savas, and C. Yilmaz, "Real time detection of cache-based side-channel attacks using hardware performance counters," Cryptology ePrint Archive, Report 2015/1034, 2015, http://eprint.iacr.org/.

[8] J. Demme, M. Maycock, J. Schmitz, A. Tang, A. Waksman, S. Sethumadhavan, and S. Stolfo, "On the feasibility of online malware detection with performance counters," *ACM SIGARCH Computer Architecture News*, vol. 41, no. 3, pp. 559–570, 2013.

[9] A. Fogh, "Cache side channel attacks," online, 2015, http://dreamsofastone.blogspot.co.at/2015/09/cache-side-channel-attacks.html.

[10] A. Fuchs and R. B. Lee, "Disruptive Prefetching: Impact on Side-Channel Attacks and Cache Designs," in *Proceedings of the 8th ACM International Systems and Storage Conference (SYSTOR'15)*, 2015.

[11] D. Gruss, C. Maurice, and S. Mangard, "Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript," *arXiv:1507.06955v1*, July 2015.

[12] D. Gruss, R. Spreitzer, and S. Mangard, "Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches," in *USENIX Security Symposium*, 2015.

[13] D. Gullasch, E. Bangerter, and S. Krenn, "Cache Games – Bringing Access-Based Cache Attacks on AES to Practice," in *S&P'11*, 2011.

[14] B. Gülmezoğlu, M. S. Inci, T. Eisenbarth, and B. Sunar, "A Faster and More Realistic Flush+Reload Attack on AES," in *Constructive Side-Channel Analysis and Secure Design (COSADE)*, 2015.

[15] N. Herath and A. Fogh, "These are Not Your Grand Daddys CPU Performance Counters - CPU Hardware Performance Counters for Security," *Black Hat 2015 Briefings*, Aug. 2015. [Online]. Available: 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

[16] R. Hund, C. Willems, and T. Holz, "Practical Timing Side Channel Attacks against Kernel Space ASLR," in *2013 IEEE Symposium on Security and Privacy*, 2013, pp. 191–205.

[17] M. S. Inci, B. Gulmezoglu, G. Irazoqui, T. Eisenbarth, and B. Sunar, "Seriously, get off my cloud! Cross-VM RSA Key Recovery in a Public Cloud," *Cryptology ePrint Archive, Report 2015/898*, pp. 1–15, 2015.

[18] Intel, "Advanced Encryption Standard (AES) Instructions Set: White Paper," 2008.

[19] G. Irazoqui, T. Eisenbarth, and B. Sunar, "S$A: A Shared Cache Attack that Works Across Cores and Defies VM Sandboxing – and its Application to AES," in *S&P'15*, 2015.

[20] G. Irazoqui, M. S. Inci, T. Eisenbarth, and B. Sunar, "Fine grain Cross-VM Attacks on Xen and VMware are possible!" *Cryptology ePrint Archive, Report 2014/248*, 2014.

[21] ——, "Wait a minute! A fast, Cross-VM attack on AES," in *RAID'14*, 2014.

[22] ——, "Know thy neighbor: Crypto library detection in cloud," *Proceedings on Privacy Enhancing Technologies*, vol. 1, no. 1, pp. 25–40, 2015.

[23] ——, "Lucky 13 strikes back," in *AsiaCCS'15*, 2015.

[24] E. Käsper and P. Schwabe, "Faster and timing-attack resistant AES-GCM," in *Cryptographic Hardware and Embedded Systems (CHES)*, 2009, pp. 1–17.

[25] J. Kelsey, B. Schneier, D. Wagner, and C. Hall, "Side Channel Cryptanalysis of Product Ciphers," *Journal of Computer Security*, vol. 8, no. 2/3, pp. 141–158, 2000.

[26] T. Kim, M. Peinado, and G. Mainar-Ruiz, "StealthMem: system-level protection against cache-based side channel attacks in the cloud," in *Proceedings of the 21st USENIX Security Symposium*, 2012.

[27] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "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.

[28] P. C. Kocher, "Timing Attacks on Implementations of Diffe-Hellman, RSA, DSS, and Other Systems," in *Proceedings of the 16th Annual International Cryptology Conference (Crypto'96)*, 1996, pp. 104–113.

[29] J. Kong, O. Acıiçmez, J.-P. Seifert, and H. Zhou, "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)*, 2009, pp. 393–404.

[30] R. Könighofer, "A fast and cache-timing resistant implementation of the AES," in *CT-RSA*, 2008, pp. 187–202.

[31] F. Liu and R. B. Lee, "Random Fill Cache Architecture," in *IEEE/ACM International Symposium on Microarchitecture (MICRO'14)*, 2014, pp. 203–215.

[32] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-Level Cache Side-Channel Attacks are Practical," in *S&P'15*, 2015.

[33] lwn.net, "2.6.26-rc1 short-form changelog," https://lwn.net/Articles/280913/, May 2008.

[34] C. Malone, M. Zahran, and R. Karri, "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.

[35] C. Maurice, N. Le Scouarnec, C. Neumann, O. Heen, and A. Francillon, "Reverse Engineering Intel Complex Addressing Using Performance Counters," in *RAID*, 2015.

[36] C. Maurice, C. Neumann, O. Heen, and A. Francillon, "C5: Cross-Cores Cache Covert Channel," in *DIMVA*, 2015.

[37] OpenSSL, "Openssl: The open source toolkit for ssl/tls," http://www.openssl.org.

[38] Y. Oren, V. P. Kemerlis, S. Sethumadhavan, and A. D. Keromytis, "The Spy in the Sandbox – Practical Cache Attacks in Javascript," *arXiv: 1502.07373v2*, 2015.

[39] D. A. Osvik, A. Shamir, and E. Tromer, "Cache Attacks and Countermeasures: the Case of AES," in *CT-RSA 2006*, 2006.

[40] C. Percival, "Cache missing for fun and profit," in *Proceedings of BSDCan*, 2005.

[41] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer, "Adaptive insertion policies for high performance caching," *ACM SIGARCH Computer Architecture News*, vol. 35, no. 2, p. 381, 2007.

[42] H. Raj, R. Nathuji, A. Singh, and P. England, "Resource Management for Isolation Enhanced Cloud Services," in *Proceedings of the 1st ACM Cloud Computing Security Workshop (CCSW'09)*, 2009, pp. 77–84.

[43] C. Rebeiro, A. D. Selvakumar, and A. S. L. Devi, "Bitslice Implementation of AES," in *Cryptology and Network Security (CANS)*, 2006, pp. 203–212.

[44] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, "Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-Party Compute Clouds," in *CCS'09*, 2009.

[45] M. Seaborn, "Exploiting the DRAM rowhammer bug to gain kernel privileges," http://googleprojectzero.blogspot.com/2015/03/exploiting-dram-rowhammer-bug-to-gain.html, March 2015, retrieved on November 10, 2015.

[46] R. Spreitzer and T. Plos, "Cache-Access Pattern Attack on Disaligned AES T-Tables," in *Constructive Side-Channel Analysis and Secure Design (COSADE)*, 2013, pp. 200–214.

[47] A. Tannous, J. T. Trostle, M. Hassan, S. E. McLaughlin, and T. Jaeger, "New Side Channels Targeted at Passwords," in *ACSAC*, 2008, pp. 45–54.

[48] E. Tromer, D. A. Osvik, and A. Shamir, "Efficient Cache Attacks on AES, and Countermeasures," *Journal of Cryptology*, vol. 23, no. 1, pp. 37–71, Jul. 2010.

[49] L. Uhsadel, A. Georges, and I. Verbauwhede, "Exploiting hardware performance counters," in *5th Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC'08).*, 2008.

[50] Z. Wang and R. B. Lee, "New cache designs for thwarting software cache-based side channel attacks," *ACM SIGARCH Computer Architecture News*, vol. 35, no. 2, p. 494, Jun. 2007.

[51] ——, "A Novel Cache Architecture with Enhanced Performance and Security," in *IEEE/ACM International Symposium on Microarchitecture (MICRO'08)*, 2008, pp. 83–93.

[52] M. Weiß, B. Heinz, and F. Stumpf, "A Cache Timing Attack on AES in Virtualization Environments," in *Proceedings of the 16th International Conference on Financial Cryptography and Data Security (FC'12)*, no. 1, 2012, pp. 314–328.

[53] C. Willems, R. Hund, A. Fobian, D. Felsch, T. Holz, and A. Vasudevan, "Down to the bare metal: Using processor features for binary analysis," in *ACSAC'12*, 2012.

[54] Y. Xia, Y. Liu, H. Chen, and B. Zang, "CFIMon: Detecting violation of control flow integrity using performance counters," in *DSN'12*, 2012.

[55] Y. Yarom and K. Falkner, "Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack," in *USENIX Security Symposium*, 2014.

[56] Y. Yarom, Q. Ge, F. Liu, R. B. Lee, and G. Heiser, "Mapping the Intel Last-Level Cache," *Cryptology ePrint Archive, Report 2015/905*, pp. 1–12, 2015.

[57] K. Zhang and X. Wang, "Peeping Tom in the Neighborhood: Keystroke Eavesdropping on Multi-User Systems," in *USENIX Security Symposium*, 2009.

[58] Y. Zhang, A. Juels, A. Oprea, and M. K. Reiter, "HomeAlone: Co-residency Detection in the Cloud via Side-Channel Analysis," in *S&P'11*, 2011.

[59] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-Tenant Side-Channel Attacks in PaaS Clouds," in *CCS'14*, 2014.

[60] Y. Zhang and M. Reiter, "Düppel: retrofitting commodity operating systems to mitigate cache side channels in the cloud," in *CCS'13*, 2013.