Write Me and I'll Tell You Secrets – Write-After-Write Effects On Intel CPUs

Jan Philipp Thoma
jan.thoma@rub.de
Ruhr University Bochum
Bochum, NRW, Germany

Tim Güneysu
tim.gueneysu@rub.de
Ruhr University Bochum & DFKI
Bochum, NRW, Germany

ABSTRACT

There is a long history of side channels in the memory hierarchy of modern CPUs. Especially the cache side channel is widely used in the context of transient execution attacks and covert channels. Therefore, many secure cache architectures have been proposed. Most of these architectures aim to make the construction of eviction sets infeasible by randomizing the address-to-cache mapping.

In this paper, we investigate the peculiarities of write instructions in recent CPUs. We identify Write+Write, a new side channel on Intel CPUs that leaks whether two addresses contend for the same cache set. We show how Write+Write can be used for rapid construction of eviction sets on current cache architectures. Moreover, we replicate the Write+Write effect in gem5 and demonstrate on the example of ScatterCache [57] how it can be exploited to efficiently attack state-of-the-art cache randomization schemes. In addition to the Write+Write side channel, we show how Write-After-Write effects can be leveraged to efficiently synchronize covert channel communication across CPU cores. This yields the potential for much more stealthy covert channel communication than before.

CCS CONCEPTS

• **Security and privacy → Side-channel analysis and countermeasures**.

KEYWORDS

Side Channels, Microarchitecture, Cache Attacks

ACM Reference Format:
Jan Philipp Thoma and Tim Güneysu. 2022. Write Me and I'll Tell You Secrets – Write-After-Write Effects On Intel CPUs. In *25th International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2022), October 26–28, 2022, Limassol, Cyprus.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3545948.3545987

1 INTRODUCTION

It is no secret that the microarchitecture of recent CPUs is riddled with side channels that can often be exploited in ways that threaten the security of the whole system. Many of these side channels are the result of obvious and elementary CPU components that pave the way to achieve the performance levels to which we have become so accustomed. Among others, this includes caches and prefetchers. The way these components are intended to work generates a timing difference that can be observed by user-level processes. On the other hand, there are more subtle aspects of CPU internals and their implementation that lead to measurable timing differences without being essential for CPUs performance or security. This, for example, includes Intel's ring interconnect implementation for last level caches (LLCs) [35] or store-to-load forwarding [43].

Due to the vast performance discrepancy between the CPU core and the memory subsystem, the read- and write path's are subject to immense optimization efforts by the CPU developers. Each saved or predicted interaction with the memory can result in hundreds of saved clock cycles. Though the performance benefit of such optimizations stands without question, ongoing research has found many ways to exploit these to bypass essential security foundations. Early work in this area demonstrated how cache-timing can be used to reconstruct secret keys of AES [2, 34]. Over time, these attacks developed towards well-known attack-primitives like Prime+Probe [25, 34, 52] and Flush+Reload [62]. With these primitives, cache attacks evolved to be very efficient and further CPU components like the TLB moved into focus [12]. In 2018, the disclosure of Meltdown [24] and Spectre [23] shifted the momentum and severity of microarchitectural attacks. The following avalanche of transient execution attacks changed the understanding of the hardware as a trust anchor for secure system development; see generally [8]. The class of transient execution attacks goes beyond control flow speculation, i.e. by the branch predictor. For example, the MDS attacks [7, 54] exploit speculative data forwarding of read- and write operations.

During transient execution attacks, leaked data is usually recovered via a covert channel. Thereby, the attacker transmits data from the (speculative) victim context to their own process using timing peculiarities of CPU internals. Covert channels can also be used to communicate between co-located VMs in cloud environments [41]. Due to the simplicity and reliability of cache covert channels, Flush+Reload, Prime+Probe and derivatives[14, 39] are commonly used in this context. The bandwidth of such covert channels has shown to be more than sufficient to transmit large chunks of data [31]. However, synchronization across cores remains an issue and is frequently evaded by using self-clocking signals [50] with massive oversampling on the receiver end and multiple accesses on the sender side [31, 58].

Contributions. In this paper, we present Write+Write, a new write-based side channel on Intel CPUs that leaks whether two physical addresses collide in a specific range. This side channel is especially worrisome in face of the current development in cache side

RAID 2022, October 26–28, 2022, Limassol, Cyprus

ACM ISBN 978-1-4503-9704-9/22/10.
https://doi.org/10.1145/3545948.3545987

channel countermeasures. We replicate the behavior in *gem5* [27] and demonstrate an improved attack against state-of-the-art cache randomization on the example of ScatterCache [57]. Our attack requires further design constraints to be considered when implementing randomized caches. Secondly, we show how Write+Write affects traditional cache architectures and leverage the side channel for bottom-up construction of cache eviction sets. In doing so, we break the current speed records in eviction set construction. Third, we present a new, write-based technique, to synchronize processes across CPU cores. We show how this technique can be applied to establish a common clock signal for covert channels. Using our synchronization approach, each signal only needs to be transmitted once which greatly reduces the monitoring surface for detection mechanisms.

A version of this paper was sent to Intel for responsible disclosure prior to submission to RAID'22. Proof-of-concept code is available on GitHub[1].

Organization of this Paper. The following section introduces background on caches, cache side- and covert channels, as well as some internals of recent x86 CPUs. In Section 3, we introduce the Write+Write side channel and the foundation for our synchronization technique. We then present a Write+Write-based algorithm for rapid eviction set construction in Section 4.1. In Section 4.2, we adapt the algorithm for randomized caches and attack a gem5 implementation of ScatterCache. Third, we demonstrate the Write-After-Write-based cross-core synchronization for covert channel communication in Section 4.3. We discuss mitigation techniques and related work in Section 5 and Section 6 respectively. Finally, we conclude in Section 7.

2 BACKGROUND

In this section, we introduce some background on caches, covert channels, and the x86 microarchitecture.

2.1 Caches

The speed at which modern processors execute instructions greatly exceeds the speed of read and write operations from and to the memory. Since many programs rely on frequent memory accesses, this would normally cause a large number of stall cycles, waiting for the requested data to be fetched. Hence, apart from deeply embedded devices, virtually all current processors feature at least one level of cache.

Caches are small and fast memory modules located in close physical proximity to the CPU. Frequently used data is stored in the cache to accelerate memory operations and hide the latency of the main memory. Most desktop-level processors feature three levels of cache. The L1-cache is the smallest and fastest cache, followed by the slightly larger and slower L2-cache. Both L1- and L2-caches are typically duplicated for each physical CPU core. The last-level-cache (LLC) is the largest level of cache and usually shared among cores. A coherency protocol is implemented to keep the data consistent across all caches and the main memory; for details on recent Intel CPUs, see [32]. Furthermore, the LLC is usually inclusive which means that all entries of the L1 and L2 caches are also stored in the LLC. This brings performance benefits in multi-core systems - if a L2 cache miss occurs, the inclusiveness makes sure that *if* the data is cached in any other cores private cache, it is also cached in the LLC. Non-inclusive LLCs need to query other cores' private caches or maintain a directory [61] to make sure that these do not hold a modified copy of the requested data.

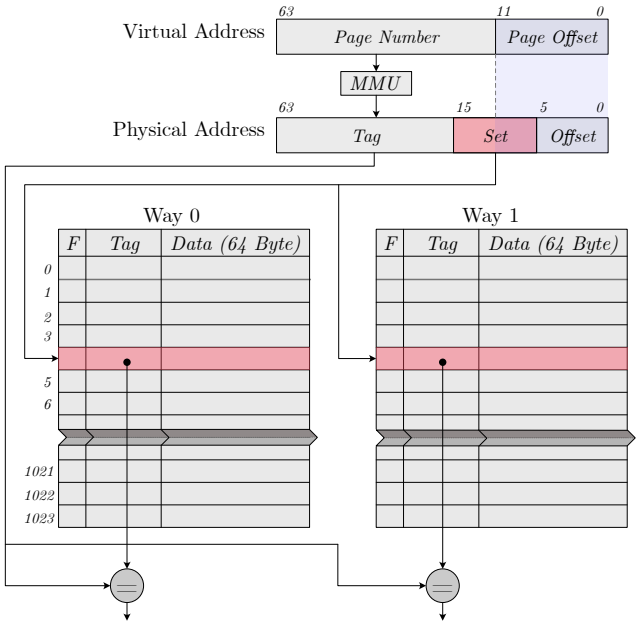

Figure 1: Exemplary architecture of a physically indexed two-way set-associative cache. The index bits of the physical address are used to determine the set (red). The replacement policy chooses which entry is replaced on a miss access.

Cache Internals & Addressing. Since low latency is a key design goal of caches, it is not practical to search the whole cache on every access. To accelerate the lookup, caches are usually implemented as set-associative structures. Each entry (*cache line*) holds 64 bytes of data alongside a tag, which is used to uniquely identify the cached address, and some flags including valid and dirty. As depicted in Figure 1, the physical address is divided into *tag*-, *set*- and *offset*-bits. The offset is used to select a 64-bit word from the cache line to be returned on read-access. The set-bits select the cache set (corresponding to the table row in each cache way in Figure 1). The remainder of the address (i.e. the tag) is stored alongside the data which together with the implicitly stored set index, uniquely identifies the physical address.

When a memory address is accessed, a cache lookup occurs and in each cache way, the tag stored at the index determined by the set bits of the address are compared to the tag of the accessed address. If the tags match in one cache way, a hit occurs and the data is returned with the specified offset. If the tags do not match in any cache way, a cache miss occurs and the data is requested from the next device in the memory hierarchy. When the request is served, the *replacement policy* selects one of the set entries to be replaced with the new data. Often, this policy is (pseudo)-least-recently-used

[1] https://github.com/Chair-for-Security-Engineering/Write-Write

((P)LRU) which replaces the entry that has not been used longest. Writes are handled analogously, although a distinction is made between write-back and write-through caches. Write-back caches store a modified version of the data until the entry gets evicted from the cache while write-through caches immediately forward the modification to the memory-side port. Recent LLCs are usually configured as write-back.

In addition to that, many processors use cache slices which can be imagined as load-balanced, parallel instantiations of caches to reduce the workload on each slice and increase the overall bandwidth of the cache. Each physical address is uniquely mapped to a single slice. Recent Intel processors implement complex cache indexing which derives the cache slice by using a recently revealed function that operates on "potentially all" address bits [19]. In [15], this complex addressing function was first reverse engineered manually, followed by [30] which utilizes a generic method based on hardware performance counters to reverse engineer the function for several Intel processors. Both works report a simple xor-based function to obtain the slice for each address.

Cache Side Channels. The design goal of caches is to accelerate slow memory accesses - hence, the fact that timing measurements can reveal whether or not some data was cached is conceptually unavoidable. An attacker can measure the latency of a memory access and therefore determine whether the accessed data was cached before the access. This effect has been exploited for numerous attacks including key-recovery on cryptographic schemes [2, 11], bypassing ASLR [13, 15], covert channels in shared cloud environments [31], and in the context of speculative execution attacks [8]. The latter - most notably by the disclosure of Spectre [23] and Meltdown [24] - hugely amplified the interest and awareness of cache side channels. The two most common attack vectors are Flush+Reload [62] and Prime+Probe [25, 34, 52]. Flush+Reload relies on shared memory between the attacker and the victim as well as the `clflush` instruction. This instruction takes a memory address as a parameter and flushes the corresponding data from all cache levels. If no data was cached for that address, the instruction has no effect. Prime+Probe on the other hand does not rely on shared memory or the `clflush` instruction. Instead, the attack makes use of eviction sets to flush the victim entry from the cache. An eviction set is a set of w addresses that map to the same cache set, where w equals the associativity of the cache. Any entries that are stored in that cache set prior to accessing the eviction set will be replaced by the eviction set. Since the eviction set addresses then occupy all entries of the set, the attacker can trigger the victim process and measure if the victim accessed that set by probing the eviction set addresses for a cache miss. If a cache miss occurs during the probing phase, the attacker learns that the victim accessed the cache set. Since the attacker does not have full control over the physical address, they can only partially control the set bits of the address, namely those that overlap with the page offset of the virtual address. However, the attacker can choose a large initial set of addresses that acts as an eviction set by sheer size, and then reduce this set to a minimal eviction set using algorithms proposed in [46, 55].

Cache Covert Channels. There are a large number of possibilities to transmit data from one process to another without an observer noticing. However, the timing behavior of caches is used disproportionately often in the context of microarchitectural attacks and covert channels, since it allows fast and fine-grained transmission. Often, Flush+Reload [62] is used for covert channels. Therefore, the receiver first makes sure that the shared address between sender and receiver is not cached using `clflush`. Note, that this shared address may be read-only. Next, the sender encodes one bit of the message by either accessing the shared address or not. The receiver then measures the latency for an access to the shared address. Only if the access results in a cache hit, the sender accessed the address. The used side channel is interchangeable for any other side channel, e.g., Flush+Flush [14] or Prime+Probe.

This process requires synchronization between sender and receiver which is not trivial. Usually, each symbol is repeated for a fixed timeframe and the sender and receiver perform their actions asynchronously. Due to the repetition, the average latency will reveal whether a zero or a one was transmitted. In order to decode the incoming data stream, often self-clocking signals like Manchester-Encoding are used [50].

Randomized Caches. In an effort to prevent the efficient construction of eviction sets, a variety of randomized cache architectures have been proposed [40, 49, 57]. These schemes randomize the address-to-cache-set mapping, such that the attacker cannot easily construct eviction sets, even if they have full control over the physical address. One physical address can map to different indices in different cache ways. This allows addresses to partially collide in one cache way but not the others and hence, weakens the properties of eviction sets. It has been shown that finding *fully congruent* eviction sets is not feasible in reasonable time [38, 57], i.e., it is not feasible to obtain sets of addresses that collide with the victim address in *every* cache way. Purnal *et al.* generalize the design proposals of randomized caches and present the Prime+Prune+Probe attack which is a generic attack on randomized caches based on probabilistic eviction sets [38]. Probabilistic eviction sets contain addresses that are known to collide in at least one cache way with the victim address. If the probabilistic eviction set contains enough of such addresses, the attacker has a high probability of occupying all possible entries of the victim address. By changing the randomization function frequently, attacks based on Prime+Prune+Probe can be prevented, albeit with some performance overhead. More recent proposals [42, 51] combine randomization with further measures to prevent Prime+Prune+Probe attacks by design. Both schemes aim to hide the effects of victim cache accesses by freeing entries in the cache before conflicts occur.

2.2 The x86 Microarchitecture

We now discuss some microarchitectural aspects of recent x86 processors. Thereby, we focus on Intel processors although the general information holds for AMD processors as well. Since most of the internals of these processors are not public, we rely on prior reverse engineering efforts and the sparse public documentation.

Store Architecture. Every fetched instruction is converted from the visible x86 instruction to one or more *μOPs* and is inserted to the pipeline. Once a write *μOP* is executed, the write is forwarded to the *store buffer* (SB). On the Skylake microarchitecture, the SB

can hold up to 56 entries [28]. Then, the L1 cache is queried. If the request results in a cache hit and the respective cache line is in modified or exclusive state (i.e. the line is owned by the cache), the data will be written into the L1 cache. Otherwise, a *request for ownership* (RFO) is issued and a *line fill buffer* (LFB) is allocated to track the outstanding write. On Sandy Bridge processors, there are 10 LFBs [16] although unofficial sources report 12 LFBs for more recent CPU generations. According to the documentation, the SB entry remains active until after the store instruction retires, i.e. the SB entry only retires after the L1 cache line is filled [16].

Serializing Instructions vs. Ordering Instructions. The x86 ISA offers a set of *serializing* instructions and *ordering* instructions that can be used to ensure the intended order of instructions and therefore prevent unwanted effects of out-of-order execution and speculation [18, Sec. 8.3]. The ordering instructions are `sfence`, `lfence` and `mfence` which are accessible from userspace. The store-fence (`sfence`) instruction ensures that all write instructions prior to the fence become globally visible before those after the fence [17, P. 4-599]. The load-fence (`lfence`) does the same for load instructions [17, P. 3-529] and the memory-fence (`mfence`) combines both fences to ensure that all loads and stores before the fence become globally visible before any load or store after the fence [17, P. 4-22].

Opposed to these memory-ordering instructions, serializing instructions enforce all modifications on the processor state made by any instruction before the serialization must be completed before the next instruction is fetched. This poses a very strong serialization since new instructions can only enter the pipeline after all prior tasks are finished. Importantly, serializing instructions also drain the SB with any outstanding write operations before the next instructions are fetched. On Intel processors there are three non-privileged serializing instructions, namely `cpuid`, `iret` and `rsm` [18, Sec. 8.3]. While the two latter perform actions that would cause significant side-effects for the following program execution, `cpuid` only affects the values of the registers `eax`, `ebx`, `ecx` and `edx`. This makes it a formidable candidate to serialize instructions in any non-privileged program. According to the AMD documentation, on AMD processors the `mfence` instruction is also a full serializing instruction [1, P. 206].

3 OBSERVATIONS ON WRITE-AFTER-WRITE

In this section we first provide details on the Write+Write side channel. We give a brief summary on the side channel, reverse engineer the exact collision criteria and reason about the origins of the side channel leakage. We then take a look at the channel noise which yields our second observation, namely the clock pattern in the write latency. Finally, we discuss the findings and identify affected CPUs.

3.1 Write+Write Side Channel

The Write+Write side channel exploits differences in the timing behavior of two write operations based on features of the physical address. In a nutshell, we observe that if a write operation is issued to a given address, a subsequent write to *some* addresses is slower than a subsequent write to *some other* addresses. We found in particular, that if the physical address of the first and the second write share some of the lower address bits, the second write will

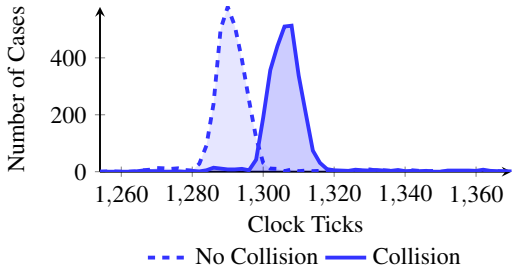

Figure 2: Distribution of write times over 3000 iterations for a conflicting address a random address.

be slower than if they do not share those bits. We reverse engineer the exact bits of the address matching function in Section 3.2. In the following, we refer to addresses that match by this function as *colliding addresses*.

A minimal proof-of-concept pseudocode is shown in Listing 1. We inserted additional instructions that enforce the execution order during the measurement and export the timestamp of the `rdtscp` instruction. For now, we set the goal to find whether a candidate address collides with a given target address and therefore causes a slower write access during the measurement. To test this, the target address is first flushed from the cache using the unprivileged `clflush` instruction of x86. The flushing can be done at any time during the attack as long as it is made sure that the data is not cached when accessed during the measurement. Then, a write operation is issued to the candidate address. The write is followed by a `cpuid` instruction which is crucial for the success of Write+Write. It makes sure that the first write instruction is retired before the timing measurement begins. Note, that Write+Write does not work if an ordering instruction like `mfence` is used instead. Opposed to ordering instructions, `cpuid` crucially also drains the internal store buffer. In the final step, the latency of a write operation to the uncached target address is measured. The distribution of the measured write latency for an address that is known to collide with the target (solid) and one that is known to not collide with the target (dashed) on an Intel Xeon E-2224G (Coffee Lake) is shown in Figure 2. Therefore, we repeatedly measured the write latency to the target address with a random candidate address *directly followed* by the measurement with a candidate that collides with the target. From the figure it is clear that the distributions can be distinguished easily.

Listing 1: AT&T syntaxed pseudo-code assembly for the Write+Write PoC.

```
CLFLUSH ([target])
MOVQ rax, ([candidate])
CPUID
RDTSCP
MOVQ rdx, ([target])
RDTSCP
```

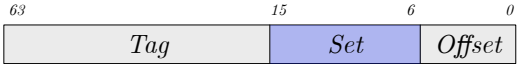

Figure 3: The bits used for the lower address match are highlighted in blue. The address is divided into the L3 cache addressing parts.

3.2 Collision Criteria

We now focus on reverse engineering the criteria under which two addresses collide, and therefore influence the write latency of each other. For this, we again use the Coffee Lake Intel Xeon E-2224G, however we later verified that the observations hold for all tested Intel CPUs, listed in Table 1. For the reverse engineering, we ran tests where we fix the target address and perform Write+Write on each address of a large array to find those colliding. We gather those addresses that led to an increased latency for further analysis. Using the *libtea* framework [10], we analyzed several properties of the addresses and found that the physical address of each analyzed address matches the target in the 10-bit range between bit 6 and 15 as shown in Figure 3. We verified this by allocating and testing possible physical addresses that only differ in the bit-range of interest and found that none of these candidates influenced each other. This rules out the possibility that the function combines some parts using a more complex technique (as it is for example the case for the cache-slice selection). We repeated this process multiple times to account for false positives and the influence of noise.

We further attempted to mount Write+Write across multiple processor cores and hyperthreads. Therefore, we tried a synchronized and a non-synchronized variant. Both variants split the Write+Write-code in two threads, one flushing the target address and measuring the access time to it, the other repeatedly writing to the candidate address. The synchronized variant utilizes mutexes to ensure the correct order of instructions, the non-synchronized variant performs the operations in a loop without synchronization. We did not find clear indications that Write+Write can be exploited across hyperthreads or CPU cores. We therefore conclude that either the addressing function implements some additional context-awareness (e.g., by matching the id of the originating core), the noise level makes it immensely hard to observe the effect, or the observed hardware structure is not shared among cores / hyperthreads. The load- and store buffers are believed to be partitioned in recent CPUs [22]. Hence, it is likely that the hardware that causes Write+Write-leakage is also partitioned.

The measurable timing difference of Write+Write is either an artifact of a false dependency check within the CPU core, or a conflicting use of some hardware resource that processes the write instructions. All our tested CPUs use exactly the 10 Bits identified with Write+Write for L2 and L3 cache indexing. Hence, we suspect that the simultaneous write access to the two addresses causes a collision in the set addressing process which yields the measurable timing difference. To support this hypothesis, we attempted to swap the write instructions for non-temporal writes, i.e., writes that do not affect the cache. After that, Write+Write no longer works. We believe that the process of set allocation is similar in the L2 and L3 cache. Since the store buffers are partitioned in recent

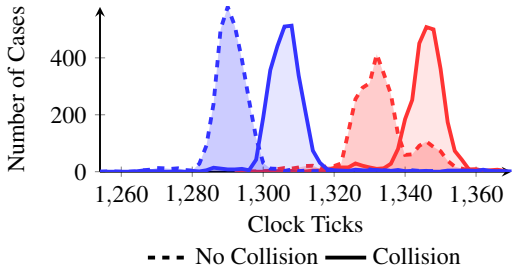

Figure 4: Two non-successive executions of Write+Write. For each execution, one address that is known to collide with the target (solid) is compared to an address that does not collide (dashed).

Intel CPUs [22] we suspect that the hardware for allocating the cache sets is also partitioned and the structural issue that causes the measurable timing delay is to be found within this logic. Since the LLC allows for cross-core attacks, we focus on the implications on the LLC set-contention in the following. We show how Write+Write can be used for efficient LLC attacks in Section 4.1.

3.3 Dealing with Noise

Figure 4 shows two non-successive executions of the PoC code. For each of these distributions, the target and the candidate address are repeatedly measured in alternating order. It is clear that while the distribution itself appears to be similar for each run, the ideal threshold that distinguishes colliding addresses from those that do not collide, varies drastically. As a result, it is not possible to make a decision based on a single measurement or even distribution. However, for measurements that are taken in close succession like the alternating measurements that make up the colliding and non-colliding distribution, a distinction is simple. Hence, the channel is only stable over short temporal periods. This distinguishes the Write+Write side channel from many other CPU side channels like Flush+Reload and Prime+Probe where a threshold can be established which reliably decides the two distributions. The reason for the temporal instability of the channel can be found in the average write latency to any address.

Listing 2: AT&T syntaxed pseudo-code to measure the write latency.

```
cpuid
rdtscp
movq rdx, ([address])
cpuid
rdtscp
```

We measure the write latency using the code in Listing 2 in a loop. The initial `cpuid` instruction ensures no unfinished write instructions are in the pipeline at the beginning of the measurement. The second `cpuid` ensures that the measurement is only stopped when the write is completed.

Figure 5 depicts the moving average of the resulting latency measurement. Surprisingly, the graph represents a rather sharp clock signal. Later in this paper we show how this can be leveraged for

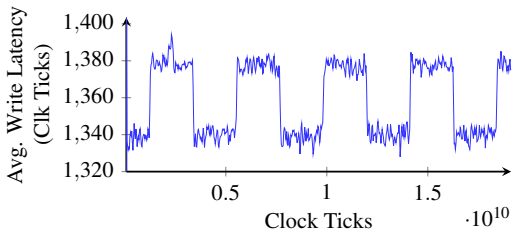

Figure 5: Moving average of the write instruction latency on the Xeon E-2224G.

cross core synchronization. We suspect that the observed behavior is an artifact of a CPU internal state-machine. Although the high- and low-level of the signal appear to be stable in the figure, we found that it can slowly change over time which might be due to the dynamic frequency adaption of the CPU.

To filter the noise and still be able to distinguish colliding addresses, it is therefore required to take a comparative approach. In other words, the results of a measurement are only valuable in comparison to another measurement taken in close succession. Since the addresses collide on 10 bits, the probability of a collision for a randomly chosen address is 2^{-10}. Hence, by choosing a random address to compare the measurement against, the attacker has a high probability of successfully gathering the addresses that collide with the target. Since some of the bits can even be influenced by the virtual address, the attacker can also make sure that the random address does not collide with the target. While it is sufficient to test multiple iterations of accessing the target address combined with the candidate address, directly followed by multiple iterations of accessing the target address combined with the random address (resulting essentially in Figure 2), we find that a better way of distinguishing the two addresses is to toggle between accessing the target- and the candidate address every second iteration. This results in an access pattern of `T-T-C-C-...` which avoids most of the prefetcher effects that are otherwise present. The measurements are summed for the candidate- and the random address respectively such that afterwards, the mean latency of both addresses can be computed. By subtracting the mean latency of the iterations with the random address from the mean latency of the iterations with the candidate address, we can test whether the distributions have a large difference in their mean value and hence conclude if the candidate collides with the target address. If the two addresses did not collide, then the distribution resulting from Write+Write with the random address is similar to the distribution of Write+Write with the candidate address, resulting in a small difference in means. We found that a threshold of 10 clock cycles difference in means after 30 iterations with each address gives a reliable indication of whether the two addresses collide on each tested CPU. The pseudocode is given in Listing 3. It is beneficial to write the code directly in assembly, using conditional moves instead of branch instructions. This prevents unwanted effects from the branch predictor and mis-speculation.

Listing 3: C-flavored pseudo code for same-process testing if two addresses collide with Write+Write.

Table 1: List of tested CPUs for Write-After-Write effects.

CPU	Architecture	W+W	Clk
Intel Xeon E-2224G	Coffee Lake	✓	✓
Intel Xeon W-3223	Cascade Lake	✓	✓
Intel i5-8259U	Coffee Lake	✓	✓
Intel i5-8265U	Whiskey Lake	✓	✓
Intel i7-7600U	Kaby Lake	✓	✓
AMD Ryzen5 5600H	Zen3	✗	✗

```
size_t *random = (size_t*) malloc(8);
bool decision = true;
for(int i=0; i < 2*RUNS; i++;){
  decision = i%4 < 2 ? 0 : 1;
  if (decision)
    sum_1 += w+w(target, candidate);
  else
    sum_2 += w+w(target, random);
}
if((sum_1/RUNS)-(sum_2/RUNS) > TH){
  // collision
}else{ /* no collision */ }
```

3.4 Discussion

We tested our implementation of the Write+Write side channel on various different CPUs which are listed in Table 1. All tested Intel CPUs showed the behavior described above. We therefore assume that most of the recent Intel CPUs will be vulnerable to Write-After-Write effects. We adapted Write+Write to an AMD CPU but were unable to identify similar effects and therefore have no indication to believe that other AMD CPUs are affected. ARM and RISC-V also feature serializing instructions and may therefore show similar behavior to the Write-After-Write clock and could be subject of further studies in future work.

Opposed to other well-known side channel attacks on modern microprocessors like Flush+Reload and Prime+Probe, the Write+Write channel relies on a write operation and does not work if an address is only read. Furthermore, it is not possible to mount Write+Write attacks across process boundaries. Therefore, Write+Write cannot directly be used to leak data from other processes. However, in combination with the aforementioned side channels, Write+Write and the clock synchronization can be useful tools in the hands of an attacker. We show how Write+Write can be used to construct eviction sets for traditional caches (Section 4.1) and on side-channel hardened architectures like ScatterCache [57] (Section 4.2). Finally, we show how the hidden clock signal can be used to synchronize covert channels (Section 4.3).

4 EXPLOITING WRITE-AFTER-WRITE

In this section we demonstrate how Write+Write can be exploited to rapidly create eviction sets. We start by attacking traditional caches on real CPUs. Then we use a gem5 implementation of ScatterCache [57] to show how Write+Write would affect randomized

caches. Finally, we use the Write-After-Write clock for cross-core synchronization of covert channels.

Unless stated otherwise, all CPUs run an unmodified version of Ubuntu 20.04. We did not disable any security / performance features or isolate cores. We use the Intel Xeon E-2224G as our main evaluation platform.

4.1 Write+Write for Rapid Cache Attacks

Prime+Probe [25, 34, 52] is one of the most widely used cache attacks. Therefore, the attacker needs to be able to efficiently construct eviction sets, allowing them to reliably observe accesses to the victim address. The state-of-the art algorithms [46, 55] obtain such eviction sets using a top-down approach. They reduce a large set of addresses that randomly include an eviction set and then filter all addresses that are not required for a minimal eviction set. Using Write+Write, it is possible to construct eviction sets using a bottom-up approach that iteratively adds addresses to the eviction set without any privileges. As we will show, this approach is much faster compared to the top-down approach. Moreover, behavioral detection mechanisms can likely be bypassed since the methodology is drastically different compared to current algorithms and therefore, the fingerprint for detection changes.

In the following, we first define the attacker model, then describe our methodology and evaluate the performance and reliability on various processors.

Attacker Model. The attacker's goal is to create an eviction set for a known target address. We assume that this address is either directly accessible, or the attacker has access to an address that contends for the same LLC cache set as the target address; i.e. the i-bit after the offset of the physical addresses match. If the address is not directly accessible, the attacker can obtain a colliding address by priming the cache and then observe which address is evicted after triggering the victim process (basically one iteration of Prime+Prune+Probe [38]). We *do not* require the attacker to know any physical addresses or the mapping of virtual to physical addresses. Furthermore, we do not make use of huge pages which are not always available. From a microarchitectural perspective, we assume that the CPU is vulnerable to Write+Write as described in Section 3.

Methodology. As described in our analysis in Section 3.2, Write+Write allows the attacker to test whether two virtual addresses map to the same cache set. This does not inherently result in addresses that collide in the LLC since the L2 cache also introduces Write+Write leakage. Though the L2 cache uses the same index bits on our evaluation CPUs, it is not partitioned into slices. Hence, with Write+Write, the attacker can identify physical addresses that collide in the cache set-index but not necessarily the cache slice. For attacks on the LLC, the attacker needs to sort out addresses that do not map to the target slice. For CPUs with complex cache indexing, an undocumented hash function is used to map the address to a cache slice. Only if the slice and the set / index of two addresses collide, the two addresses can potentially evict each other. The slice addressing function has been reverse engineered in [15, 20, 30]. Our approach does not require any knowledge about this function.

The algorithm to construct eviction sets using Write+Write is shown in Algorithm 1. Therefore, we first allocate a sufficiently large memory area. The algorithm takes as input a pointer to the target address, a pointer to the memory area of size *mem_size*, and the number of repetitions for each candidate. Since some of the cache set bits can be directly controlled from the virtual address space, we align the lower 12 bits of the first virtual address to be tested with the lower 12 bits of the victims virtual address. As discussed in Section 3.3, the results are only meaningful when compared to another measurement in close succession. Previously we mentioned that the target address can be tested alternating with the candidate address and a random address. In that case, the timing difference is reliably measured if the candidate address collides. Therefore, the attacker needs to make sure that the random address maps to a different cache set using some bits of the virtual address. However, we found that for performance reasons, a better approach is to test two candidate addresses (i.e. both are 2^{12} byte aligned to the target) in parallel and compare the timing of these instead of a random address. This way, a timing difference will be observed if one of the addresses collide with the target but neither if none or both do. Since it is relatively unlikely that two successive candidate addresses map to the same cache index, two strategies are possible: The coverage optimized variant aims to retrieve the most conflicts from a memory range. In that case, the two candidate addresses are $base + i * 2^{12}$ and $base + i * 2 * 2^{12}$ and i is increased by 2^{12} in each step. This way, each address is tested twice which reduces the probability of missing a conflict. The performance optimized version increases i by $2 * 2^{12}$ in each iteration which does not detect if both candidate addresses collide with the target but instead doubles the execution speed. In the following, we use the performance variant, as shown in the algorithm.

To reduce the error rate, each pair of addresses is tested multiple times and the results are averaged. If the absolute difference in means of the two candidate addresses is larger than a threshold, one of the candidate addresses collides with the target. The sign of the difference indicates which of the candidate addresses collides. When a collision is observed, the address is added to the preliminary eviction set. The attacker frequently tests if the eviction set is functional by measuring whether it evicts the target address. As long as the eviction set is not functional, the attacker continues searching for Write+Write collisions. When the preliminary eviction set is functional, the attacker can choose to remove false positives and addresses that map to different slices by removing one address at a time from the set and testing whether the remaining addresses still form an eviction set. This step is not strictly necessary since the initial set is also functional but depending on the use-case, it may be important for the attacker to obtain a minimal eviction set.

Performance and Reliability. In the following we evaluate the Write+Write-based eviction set assembly on different target CPUs. The previously mentioned Xeon W-3223 CPU is the only one of our test-sample implements a non-inclusive LLC. We therefore do not consider the Xeon W-3223 for the eviction set use-case. In Figure 6, the confidence level of a Write+Write-based observation after $2n$ measurements is depicted. This includes n measurements with the first candidate address that are compared to n measurement to the second candidate address. The True-Positive-Rate (TPR) rises

Algorithm 1 WRITE+WRITE-based eviction set construction for traditional caches.

```
Input: *target, *mem, mem_size, rep
ev ← ∅;
start ← align(mem, target);
for i = 0; i < mem_size; i += 2 · 0x1000 do
    sum_0 ← 0; sum_1 ← 0;
    candidate_0 ← start + i;
    candidate_1 ← start + i + 0x1000;
    for j = 0; j < rep; j ++ do
        decision ← (j & 0x2) >> 1;
        if decision == 0 then
            sum_0 += w+w(target, candidate_0);
        else
            sum_1 += w+w(target, candidate_1);
        end if
    end for
    avg_0 ← sum_0/(rep/2);
    avg_1 ← sum_1/(rep/2);
    if avg_1 − avg_0 > TH then
        ev ← ev ⋃ candidate_0;
    else if avg_1 − avg_0 < −TH then
        ev ← ev ⋃ candidate_1;
    end if
    if test_evset(ev) == true then
        break;
    end if
end for
ev ← reduce(ev);                                  ▷ Optional.
return ev
```

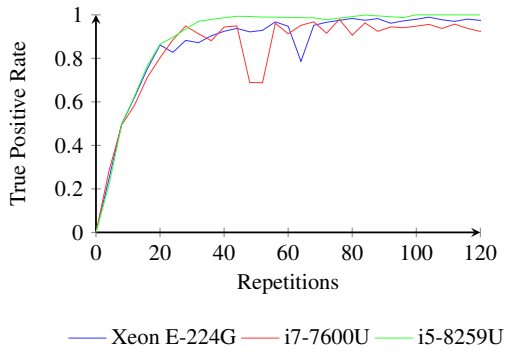

Figure 6: TPR of a address classification as colliding with the victim address in dependence of the number of repeated measurements. Each data point was averaged over 30 executions.

sharply after the first few repetitions and then converges towards 1. The characteristic is similar for all CPUs with the exception of small outliers. These may be due to scheduler interruptions or system noise during the measurement.

Table 2: Performance evaluation of WRITE+WRITE-based eviction set construction. The results are averaged over 500 runs. The time includes filtering of false-positives and addresses that map to other slices.

	This Work	Song *et al.* [46]
Xeon E-2224G	26.6 ms (96%)	156 ms (60%)
i5-8259U	40.6 ms (86%)	146 ms (71%)
i5-8265U	72.5 ms (81%)	228 ms (59%)
i7-7600U	21.1 ms (95%)	133 ms (68%)

For the construction of eviction sets it is important that WRITE+WRITE reliably detects colliding addresses. Therefore, we now investigate the coverage, i.e. how many addresses from the memory area collide with the target and how many collisions are found using WRITE+WRITE. We use the libtea framework [10] to compare the addresses that map to the same cache set to those returned by WRITE+WRITE. Each measurement for WRITE+WRITE is repeated 30 times in order to achieve a good TPR. We found that in this configuration, WRITE+WRITE detects about 90% of the colliding addresses.

Table 2 shows the performance for eviction set construction of all tested CPUs including the reduction to a minimal eviction set and compares it to the currently fastest algorithm for eviction set construction by Song *et al.* [46]. We found that the optimal performance for WRITE+WRITE-based eviction set construction can be achieved using a tradeoff between WRITE+WRITE repetitions and the amount of false-positive classifications for collisions. As shown in Figure 6, the TPR of a collision classification becomes very high for more than 30 repetitions of WRITE+WRITE. However, our experiments revealed that the runtime of the eviction set construction algorithm is minimal with about 10 to 15 repetitions. This leads to more false positives in the preliminary eviction set which increases the runtime of the eviction set reduction to a minimal eviction set but reduces the time to probe for conflicts in the first part of the algorithm.

We executed the code by Song *et al.* on our evaluation CPUs to get a clearer picture of the performance difference[2]. For all tested CPUs, the WRITE+WRITE-based eviction set construction outperforms the previous approach by a factor of three to six. The success rate is also very high throughout all our experiments. Runs that have been classified as failing mostly include only one address that is wrongly classified as collision. In such a case, the false address could be exchanged for a different colliding address without much additional computing time.

4.2 Attacks on Randomized Caches

The search for effective countermeasures to thwart cache side channels has peaked in a number of cache architectures that randomize the cache index using (partial) address encryption [42, 49, 51, 57] to prevent efficient eviction set generation. Cache randomization is generally considered to make attacks more difficult [45], even

[2]Our results generally meet the numbers reported in their paper. However, due to the vast configurability, it may be that there are slightly more optimal configurations. We do not expect major deviations.

though attacks like Prime+Prune+Probe [4, 38] are still feasible on pure index-randomization schemes. More recent designs try to encounter such attacks with further security mechanisms [42, 51]. Hence, we believe to see some form of index randomization to be adapted by major CPU vendors in the not-too-distant future.

Since the principle of contention is still present in randomized caches, i.e. two addresses *can* still collide in the cache, it is likely that the implementation of the entry selection remains similar and hence, Write+Write-like leakage may still exist.

Methodology. To demonstrate the threats of Write+Write leakage in the randomized cache setting, we implement the Write+Write behavior in the CPU simulator gem5 [27]. On traditional caches, Write+Write causes an increased write latency when two addresses map to the same cache index. In randomized caches, addresses may have an index collision in one cache way, but not the others. In our implementation, the increased latency occurs when two successive write operations are issued and the address of the second write can map to the same index in the cache way in which the first write is stored. This follows our assumption that Write+Write is caused by a conflict in the simultaneous set allocation of two write operations. The intention is that the first write is assigned to a set, and then the second write needs to be placed. If a second write can map to the entry in which the first one was placed, the replacement policy needs to wait until the first write has updated the replacement data. We chose a conservative approach where the other option would be that a timing difference is measurable if the two addresses can collide in any cache way. The effect on the security on randomized caches would be equal, however, the latter approach would accelerate the attack even further.

For the randomized cache, we implement ScatterCache [57] with random replacement policy in gem5. Thus, each address is randomized in every cache way individually, yielding w independent cache indices in a w-way cache. The attacker model is similar to the one in the non-randomized setting, although the aim is no longer to construct a minimal eviction set but instead a probabilistic one; see [38]. That is, since constructing a minimal eviction set would require the attacker to find w fully congruent addresses. This has been shown to be infeasible [57].

The algorithm to construct a probabilistic eviction set using Write+Write is similar to the algorithm in the non-randomized cache and shown in Algorithm 2. The main difference is that the candidate addresses are no longer aligned to the target address since the attacker cannot influence the lower bits used for set selection from the virtual address space. This increases the search space for colliding addresses significantly. Each cache line holds 64 byte of data, hence, the attacker needs to probe for colliding addresses in a 64-byte stride. Smaller offsets would result in addresses that map to the same cache line while larger offsets would skip potentially conflicting addresses. A further difference is that the attacker can no longer deterministically test if the eviction set is complete. The test needs to be conducted multiple times and the attacker needs to determine whether the eviction set evicts the target with the expected probability p_e. The optional reduction to a minimal eviction set also differs from the traditional algorithm. Instead of removing one address at a time and probing if the eviction set is still functional, the attacker can prime the target multiple times and remove those addresses that are never evicted by the target.

Algorithm 2 Write+Write-based eviction set construction for randomized caches.

Input: $*target$, $*mem$, mem_size, rep
$ev \leftarrow \emptyset$;
for $i = 0$; $i < mem\_size$; $i \mathrel{+}= 2 \cdot 0x40$ **do**
  $sum_0 \leftarrow 0$; $sum_1 \leftarrow 0$;
  $candidate_0 \leftarrow mem + i$;
  $candidate_1 \leftarrow mem + i + 0x40$;
  for $j = 0$; $j < rep$; j ++ **do**
    $decision \leftarrow (j \,\&\, 0x2) >> 1$;
    if $decision == 0$ **then**
      sum_0 += w+w($target, candidate_0$);
    else
      sum_1 += w+w($target, candidate_1$);
    end if
  end for
  $avg_0 \leftarrow sum_0/(rep/2)$;
  $avg_1 \leftarrow sum_1/(rep/2)$;
  if $avg_1 - avg_0 > TH$ **then**
    $ev \leftarrow ev \bigcup candidate_0$;
  else if $avg_1 - avg_0 < -TH$ **then**
    $ev \leftarrow ev \bigcup candidate_1$;
  end if
  if $test\_evset\_ratio(ev) \approx p_e$ **then**
    $break$;
  end if
end for
$ev \leftarrow$ reduce(ev); ▷ Optional.
return ev

Performance Evaluation. We configured gem5 to use the O3 CPU model equipped with a small 64 kB, 2-way associative L1 cache and a larger 1 MB, 8 way associative L2 cache. Both cache levels use our ScatterCache implementation. The randomization function is a round reduced version of PRINCE [3] with equal keys in the L1 and L2 cache. This way, the generalized eviction set evicts entries from both levels of cache. In practice, this reduces the complexity of cache randomization, since the address encryption only needs to be performed once for cache lookups in any cache level.

We implement our attack and execute it in gem5's syscall emulation (SE) mode. The SE mode executes the binaries in an isolated environment without any operating system or parallel processes. Calls to OS functions are handled by gem5 directly. Hence, in our setup there is no noise or disturbance by scheduler decisions or parallel processes. The numbers reported in the following therefore represent the best-case for an attacker.

Constructing a probabilistic eviction set with $p_e = 90\%$ for an 8-way cache requires about 143 addresses that collide with the target in at least one cache way as shown in [38]. We executed the attack 10 times and the average runtime was 267 ms. Furthermore, we verified that the correctness of the constructed eviction set by probing whether it evicts the target with the expected probability.

To compare our attack to the established Prime+Prune+Probe attack [38], we also implement this attack in the same setup. Prime+Prune+Probe repeatedly fills large parts of the cache and accesses the prime-set until there are no more evictions within this set. Then, the target address is accessed which with some probability evicts one of the attacker controlled addresses. If an eviction is observed, the address is added to the generalized eviction set. We found that priming 50% of the cache leads to a reasonable probability of observing an eviction while keeping the amount of conflicts in the prune phase low. Using the same settings, we executed the attack 10 times and the average runtime was 2.324 s. Hence, the Write+Write-based attack outperforms Prime+Prune+Probe by a factor of 10 in this cache configuration. In practice, priming and pruning a large part of the cache is difficult in the presence of noise induced by parallel processes. Any such process may cause an eviction of the pruned set which leads to a false-positive observation. Write+Write is less affected by such noise since only two addresses are used to perform the measurement. Therefore, we expect that our attack would perform even better in real-world scenarios compared to Prime+Prune+Probe.

4.3 Cross-Core Synchronization

If an attacker wants to communicate over a covert channel, they must make sure that the sender- and receiver process are properly synchronized. This is not trivial since there is no direct way for the sender to communicate to the receiver that the next symbol starts and in many scenarios, there is no common clock (e.g. when using virtualization). In prior work this problem is often evaded by using edge detection during post-processing of the received signal [21] or using self-clocking encodings such as the phase / Manchester encoding [31, 50, 58]. However, the former suffers from noise being classified as edges and limitations on how short a single symbol can be, while the latter reduces the channel entropy since two bits are needed to transmit one bit of information. Moreover, if for example, the Flush+Reload channel shall be used for transmission, the sender and receiver have no way of coordinating the flush/reload and access steps. In practice, the sender and receiver perform the flush and the access in parallel which on average leads to the low reload latency on the receiver end. However, this approach is not very stealthy. If the sender and receiver were to share a precise clock source, they could coordinate the flush and access in a way that each probe by the receiver yields precisely one bit of message. This is exactly what the Write-After-Write clock achieves.

In the following, we demonstrate how changes in the average write latency can be leveraged for cross-core synchronization of multiple processes. We show how covert channels can synchronize a sender and receiver process by observing the average write latency to an arbitrary address. Importantly, this address does not need to be shared between the sender and receiver process. Due to the synchronization, the sender only needs to send each symbol once and the receiver only once measures the access time to read the symbol. This makes our approach much more stealthy compared to previous techniques.

Attacker Model. We assume that the attacker controls two processes on the target device. The goal is to transmit a message from one process to the other over a covert channel. The sender and the receiver process execute in parallel but not necessarily on the same physical CPU core. For covert channels that require shared memory (i.e. Flush+Reload [62] and Flush+Flush [14]), we assume that both processes can access a shared memory resource. For Prime+Probe, this is not required. Furthermore, we assume that both parties have access to a precise timer. Should `rdtsc` not be available, such timers can easily be constructed as shown by Schwarz *et al.* [44].

Methodology. As shown earlier (c.f. Figure 5), the average write latency to a given address periodically switches between a *high* and a *low* state. The resulting signal already resembles a very sharp clock signal. We verified that this change in the write latency is synchronized across multiple CPU cores. The raw measurement is shown in Figure 7 (❶). While the average latency is reasonably stable, the latency of single write instructions can vary massively, making trivial classification to the low, or the high state infeasible. Therefore, we compute a running average of the write latency (❷) and then perform edge detection on that data (❸). To generate the clock signal, we hence instantiate a loop that measures the write latency to a given address constantly. It does not matter whether the address is cached, as long as it is either always cached or always not cached. A ring buffer is used to compute a moving average. Moreover, we implement a second ring buffer that stores delta between the measured time and the current moving average. This way, the average of the second ring buffer is close to zero if the mean write latency is stable, but if the average write latency changes abruptly, the mean over the second ring buffer will peak briefly. Using this technique, a simple threshold value is sufficient to detect changes in the write latency and therefore, generate the clock signal. If the average change in write latency is above a threshold (e.g., 15) and the current clock is low, the signal changes to high and vice versa. As depicted in Figure 7 (❹), the synchronization is highly accurate and even in the selected small time frame, no visible error can be observed. We introduce two metrics to evaluate the accuracy of the received clock signal. The cycle-to-cycle jitter measures the mean difference of clock periods of two successive cycles. It calculates as $J_{cc} = mean(|T_j - T_{j+1}|)\forall j$. To quantify the measurement error of two processes observing the Write-After-Write clock, we define the synchronization error as the mean difference between the detection of a clock edge of two processes. It calculates as $S_{cc} = mean(|T_j^{(P0)} - T_j^{(P1)}|)\forall j$.

Until now, the clock period is fixed by the characteristics of the write latency. However, if the synchronization error is small, the sender and receiver can split each clock period in smaller chunks and hence increase the bandwidth. Therefore, both the sender and receiver need to keep track of the average clock period. The edges of the Write-After-Write clock serve as synchronization marks from which both the sender and receiver separate the expected period in n timeframes, each of which is used to transmit one bit of message. In the following, we use the Flush+Reload covert channel to transmit a message from the sender to the receiver process. If a '1'-bit is to be transmitted, the sender will access the shared memory address on the rising edge of the Write-After-Write clock. If a '0'-bit is transmitted, the sender *does not* access that address. The receiver then measures the access (read) latency to the shared address on each falling clock edge and flushes the shared address afterwards

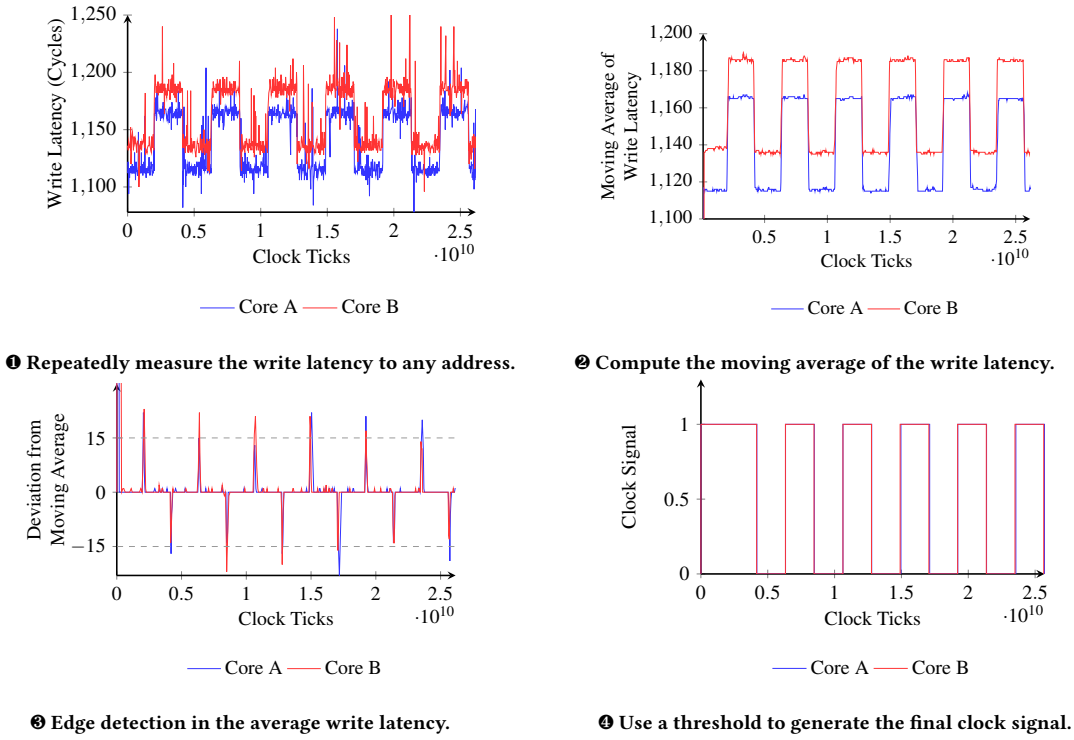

Figure 7: Illustration of the process to generate a synchronized clock signal between two processes, based on an Intel Xeon E-2224G.

Table 3: Statistics for various CPUs averaged over 100 iterations. The average jitter and the synchronization error is shown in percent of the mean clock period.

CPU	Period (clk)	J_{CC}/clk	S_{CC}/clk
Xeon E-2224G	$4.295x10^9$	0.13%	0.02%
Xeon W-3223	$4.294x10^9$	0.02%	0.03%
i5-8259U	$4.295x10^9$	0.09%	0.03%
i5-8265U	$4.587x10^9$	10.03%	0.02%
i7-7600U	$4.157x10^9$	5.1%	0.05%

to prepare for the next symbol. On average, transmitting one bit of message therefore only requires 0.5 memory accesses on the sender side, and a single memory access and a cache flush instruction on the receiver side.

Performance and Reliability. We first measure some characteristics of the Write-After-Write clock on our target CPUs. Therefore, we execute two processes on each CPU that both measure the write latency. The results are shown in Table 3. The measured clock period is similar in all our measurements. For the two Xeon processors and the i5-9259U, the jitter is low, indicating a very stable clock period. However, on the i5-8265U and the i7-7600U the, the jitter is much higher. On these CPUs, we experienced a large number of outliers during the measurement which reduces the accuracy of the observed clock signal. However, the synchronization error is very low on all tested CPUs, i.e. both processes are equally affected by the noise and hence, synchronization is still provided.

We now use the Write-After-Write clock to synchronize a covert channel communication using Flush+Reload on the Xeon E2224-G CPU. Both the sender and receiver process outsource the clock generation to a separate thread. This way, we get the highest possible sampling rate of the write latency which improves the accuracy of the retrieved clock signal. We furthermore schedule each process (sender, receiver and two clock threads) on different CPU cores to avoid heavy noise disturbance. During a transmission two kinds of errors may occur: We classify a flipped bit as a *transmission error* and a missing or added bit as a *clock error*. Transmission errors occur if the covert channel is noisy or the threshold is not optimally chosen. Clock errors are an artifact of failed clock synchronization. This may happen if one of the processes gets descheduled by the scheduler and therefore misses a clock edge, or if the clock signal is disturbed by system noise. In our implementation, clock errors also occur if the receiver process is stopped too late or too early which might lead to additional or missing symbols. Since such errors are easily detected in the final message, we do not count them into the error rates.

To compare the sent and the received data and classify the errors, we use the Needleman-Wunsch algorithm [33]. The algorithm originates in bioinformatics and can be used to identify matches, mismatches and gaps in two input vectors. Mismatches correspond

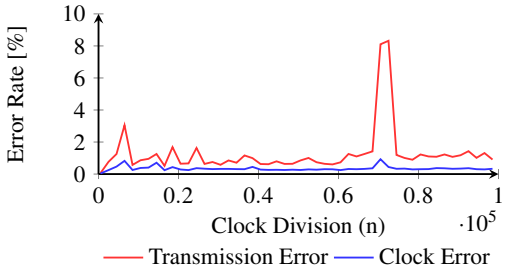

Figure 8: Transmission errors and clock errors in percent as a function of the clock division rate on the Intel Xeon E-2224G processor.

to transmission errors and gaps correspond to synchronization errors during the transmission. We configure the match and mismatch scores to 1 and -1 respectively. We set the score for gaps to -8 to prevent false classifications as gaps. We do not implement any error correction during the transmission which could reduce the amount of gaps and mismatches.

Figure 8 shows the average transmission- and clock error rate in percent. We therefore compute the average over eight transmissions of 1kB data over the covert channel using the write-synchronization method. With the exception of one outlier at $n \approx 75,000$, the error rates are very low. The transmission error rate which is purely influenced by the accuracy of Flush+Reload is at about 1% while the clock error rate is significantly smaller between 0.2% to 0.8%. The outlier may be explained by a scheduled task that interrupted one of the processes. Since the Xeon E-2224G CPU has only four cores, any additional process directly competes for CPU time. We achieved a maximum transmission rate of up to 2 kB/s. Since the error rates are still low at this rate, we suspect that the bottleneck of the transmission speed lies in the communication between the clock generating thread and the sender / receiver. We also tested the code on an AMD Ryzen 5 5600H but did not observe the same clock-like characteristic in the write latency.

5 MITIGATION

The mitigation of the Write-After-Write effects requires modification on the microarchitectural level by re-designing the aspects of the CPU that lead to the behavior exploited in this work. This, however, requires significant change to the design which cannot be applied to currently deployed CPUs. A less intrusive way to prevent Write-After-Write effects would be a change in the behavior of the `cpuid` instruction. There is no apparent reason why the instruction should be unprivileged and serializing. The fencing instructions (`[s,l,m]fence`) are well defined and sufficient to prevent speculative execution attacks which is a valid use-case for many programs. To the best of our knowledge, beyond that, there is no further scenario for an unprivileged serializing instruction.

In-line with previous research on cache side channels, making the `clflush` instruction privileged would render Write+Write infeasible in unprivileged environments. However, this does not affect the clock synchronization based on the average write latency.

6 RELATED WORK

In this section, we briefly summarize related work.

Side Channels. There is a vast variety of different side channels in modern CPUs. In the following, we focus on memory-related side channels in desktop- and server-grade CPUs; for recent surveys, see [26, 47, 48]. Caches are the most commonly used source of leakage. Early timing-based attacks could be used to recover cryptographic keys, among others of AES [2], DES [53], and RSA [6] by measuring the overall runtime of the program. Eviction-based cache side channels increase the attack resolution since they allow the attacker to trace single cache accesses by the victim. Evict+Time [34] compares the execution time of a program before and after some cache entries were evicted to reconstruct an AES key. Flush+Reload [62] flushes a shared cache line and measures whether it will be reloaded by the victim. Flush+Flush [14] operates similarly but instead of measuring the reload-latency, it measures the latency of a second `clflush` instruction. These attacks require shared memory between the attacker and the receiver. Prime+Probe [34, 52] instead uses eviction sets to evict entries from the cache. Reload+Refresh [5] is a variant of Prime+Probe that reduces the amount of cache misses and exploits the replacement policy of caches. Both attacks rely on eviction sets [25] that reliably evict a target entry from the cache. An algorithm for finding such eviction sets has been presented in [55] and improved in [46]. Several randomization-based cache designs have been presented to prevent the construction of eviction sets [42, 49, 51, 56, 57]. The Prime+Prune+Probe attack [4, 38] targets randomized caches and constructs generalized eviction sets, albeit much less performant compared to regular caches. Other side channels in the memory hierarchy have been discovered, most notably on TLBs [9, 12], DRAM [37], and the on-chip ring interconnect [35]. The group of MDS attacks [7, 54] exploit speculative behavior in Intel's store buffers.

Covert Channels. Cross-VM covert channel communications have been studied in real-world environments on AWS systems in [41, 60]. Wu *et al.* use a memory-bus-based covert channel in [58], and Xiao *et al.* exploit memory deduplication for covert-communication [59]. Cache-based covert channels have been presented in [14, 29, 36, 60]. In [31] it has been shown that cache covert channels can even be used to establish ssh connections between the communication partners.

7 CONCLUSION

We investigated the microarchitectural peculiarities of write instructions on recent Intel processors. We discovered Write+Write, a new side channel that leaks set contention in the cache architecture. We used Write+Write for bottom-up construction of cache eviction sets and in doing so, broke current speed records for eviction set construction. Furthermore, we demonstrated that attacks on randomized caches can be accelerated significantly if Write+Write leakage is present. Therefore, we implemented ScatterCache in gem5 and benchmarked our attack against the recent Prime+Prune+Probe attack. We found that the Write+Write-based attack outperforms current attacks by a factor of 10 and expect an even larger advantage in real-world implementations.

That is, since the Write+Write algorithm for eviction set construction is much less susceptible to noise by parallel processes.

Moreover, we developed a new approach to synchronize processes across CPU cores. The clock-like nature of the noise in the write latency allows for accurate synchronization and therefore more stealthy covert channel transmissions.

ACKNOWLEDGMENTS

Funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy - EXC 2092 CASA - 390781972 and by the DFG under the Priority Program SPP 2253 Nano Security (Project RAINCOAT - Number: 440059533). "Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of DFG or other funding agencies." Date of this document: June 20th, 2022

REFERENCES

[1] AMD. 2021. AMD64 ArchitectureProgrammer's Manual Volume 2: System Programming. *AMD Documentation* 1, 64 (2021), 64.

[2] Daniel J. Bernstein. 2005. *Cache-timing attacks on AES.* Technical Report. University of Illinois at Chicago.

[3] Julia Borghoff, Anne Canteaut, Tim Güneysu, Elif Bilge Kavun, Miroslav Knezevic, Lars R. Knudsen, Gregor Leander, Ventzislav Nikov, Christof Paar, Christian Rechberger, Peter Rombouts, Søren S. Thomsen, and Tolga Yalçin. 2012. PRINCE - A Low-Latency Block Cipher for Pervasive Computing Applications - Extended Abstract. In *Advances in Cryptology - ASIACRYPT 2012 - 18th International Conference on the Theory and Application of Cryptology and Information Security, Beijing, China, December 2-6, 2012. Proceedings (Lecture Notes in Computer Science, Vol. 7658)*, Xiaoyun Wang and Kazue Sako (Eds.). Springer, Beijing, China, 208–225. https://doi.org/10.1007/978-3-642-34961-4_14

[4] Thomas Bourgeat, Jules Drean, Yuheng Yang, Lillian Tsai, Joel S. Emer, and Mengjia Yan. 2020. CaSA: End-to-end Quantitative Security Analysis of Randomly Mapped Caches. In *53rd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2020, Athens, Greece, October 17-21, 2020*. IEEE, 1110–1123. https://doi.org/10.1109/MICRO50266.2020.00092

[5] Samira Briongos, Pedro Malagon, Jose M. Moya, and Thomas Eisenbarth. 2020. RELOAD+REFRESH: Abusing Cache Replacement Policies to Perform Stealthy Cache Attacks. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 1967–1984. https://www.usenix.org/conference/usenixsecurity20/presentation/briongos

[6] David Brumley and Dan Boneh. 2005. Remote timing attacks are practical. *Comput. Networks* 48, 5 (2005), 701–716. https://doi.org/10.1016/j.comnet.2005.01.010

[7] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. 2019. Fallout: Leaking Data on Meltdown-resistant CPUs. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz (Eds.). ACM, 769–784. https://doi.org/10.1145/3319535.3363219

[8] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin Von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. 2019. A systematic evaluation of transient execution attacks and defenses. In *28th USENIX Security Symposium (USENIX Security 19)*. 249–266.

[9] Shuwen Deng, Wenjie Xiong, and Jakub Szefer. 2019. Secure TLBs. In *Proceedings of the 46th International Symposium on Computer Architecture, ISCA 2019, Phoenix, AZ, USA, June 22-26, 2019*, Srilatha Bobbie Manne, Hillery C. Hunter, and Erik R. Altman (Eds.). ACM, 346–359. https://doi.org/10.1145/3307650.3322238

[10] Catherine Easdon, Michael Schwarz, Martin Schwarzl, and Daniel Gruss. 2022. Rapid Prototyping for Microarchitectural Attacks. In *USENIX Security Symposium*.

[11] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. 2017. Cache Attacks on Intel SGX. In *Proceedings of the 10th European Workshop on Systems Security, EUROSEC 2017, Belgrade, Serbia, April 23, 2017*, Cristiano Giuffrida and Angelos Stavrou (Eds.). ACM, 2:1–2:6. https://doi.org/10.1145/3065913.3065915

[12] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2018. Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks. In *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, William Enck and Adrienne Porter Felt (Eds.). USENIX Association, 955–972. https://www.usenix.org/conference/usenixsecurity18/presentation/gras

[13] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. 2017. ASLR on the Line: Practical Cache Attacks on the MMU. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*. The Internet Society. https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/aslrcache-practical-cache-attacks-mmu/

[14] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. 2016. Flush+Flush: A Fast and Stealthy Cache Attack. In *Detection of Intrusions and Malware, and Vulnerability Assessment - 13th International Conference, DIMVA 2016, San Sebastián, Spain, July 7-8, 2016, Proceedings (Lecture Notes in Computer Science, Vol. 9721)*, Juan Caballero, Urko Zurutuza, and Ricardo J. Rodríguez (Eds.). Springer, 279–299. https://doi.org/10.1007/978-3-319-40667-1_14

[15] Ralf Hund, Carsten Willems, and Thorsten Holz. 2013. Practical Timing Side Channel Attacks Against Kernel Space ASLR. In *20th Annual Network and Distributed System Security Symposium, NDSS 2013, San Diego, California, USA, February 24-27, 2013*. The Internet Society. https://www.ndss-symposium.org/ndss2013/practical-timing-side-channel-attacks-against-kernel-space-aslr

[16] Intel. 2016. Intel 64 and IA-32 Architectures Optimization Reference Manual. *Intel ISA Documentation* (2016).

[17] Intel. 2016. Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 2 (2A, 2B, 2C & 2D):Instruction Set Reference, A-Z. *Intel ISA Documentation* (2016).

[18] Intel. 2016. Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3A: System Programming Guide, Part 1. *Intel ISA Documentation* (2016).

[19] Intel. 2020. Intel Architecture Instruction Set Extensions and Future Features Programming Reference. *Intel ISA Documentation* (2020).

[20] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. 2015. Systematic Reverse Engineering of Cache Slice Selection in Intel Processors. In *2015 Euromicro Conference on Digital System Design, DSD 2015, Madeira, Portugal, August 26-28, 2015*. IEEE Computer Society, 629–636. https://doi.org/10.1109/DSD.2015.56

[21] Manuel Kalmbach, Mathias Gottschlag, Tim Schmidt, and Frank Bellosa. 2020. TurboCC: A Practical Frequency-Based Covert Channel with Intel Turbo Boost. *arXiv preprint arXiv:2007.07046* (2020).

[22] David Kanter. 2012. *Intel's Haswell CPU Microarchitecture - Haswell Memory Hierarchy.* Technical Report.

[23] Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*.

[24] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading Kernel Memory from User Space. In *27th USENIX Security Symposium (USENIX Security 18)*.

[25] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. 2015. Last-Level Cache Side-Channel Attacks are Practical. In *2015 IEEE Symposium on Security and Privacy (SP)*. IEEE. https://doi.org/10.1109/sp.2015.43

[26] Xiaoxuan Lou, Tianwei Zhang, Jun Jiang, and Yinqian Zhang. 2021. A Survey of Microarchitectural Side-channel Vulnerabilities, Attacks, and Defenses in Cryptography. *ACM Comput. Surv.* 54, 6 (2021), 122:1–122:37. https://doi.org/10.1145/3456629

[27] Jason Lowe-Power, Abdul Mutaal Ahmad, Ayaz Akram, Mohammad Alian, Rico Amslinger, Matteo Andreozzi, Adrià Armejach, Nils Asmussen, Srikant Bharadwaj, Gabe Black, Gedare Bloom, Bobby R. Bruce, Daniel Rodrigues Carvalho, Jerónimo Castrillón, Lizhong Chen, Nicolas Derumigny, Stephan Diestelhorst, Wendy Elsasser, Marjan Fariborz, Amin Farmahini Farahani, Pouya Fotouhi, Ryan Gambord, Jayneel Gandhi, Dibakar Gope, Thomas Grass, Bagus Hanindhito, Andreas Hansson, Swapnil Haria, Austin Harris, Timothy Hayes, Adrian Herrera, Matthew Horsnell, Syed Ali Raza Jafri, Radhika Jagtap, Hanhwi Jang, Reiley Jeyapaul, Timothy M. Jones, Matthias Jung, Subash Kannoth, Hamidreza Khaleghzadeh, Yuetsu Kodama, Tushar Krishna, Tommaso Marinelli, Christian Menard, Andrea Mondelli, Tiago Mück, Omar Naji, Krishnendra Nathella, Hoa Nguyen, Nikos Nikoleris, Lena E. Olson, Marc S. Orr, Binh Pham, Pablo Prieto, Trivikram Reddy, Alec Roelke, Mahyar Samani, Andreas Sandberg, Javier Setoain, Boris Shingarov, Matthew D. Sinclair, Tuan Ta, Rahul Thakur, Giacomo Travaglini, Michael Upton, Nilay Vaish, Ilias Vougioukas, Zhengrong Wang, Norbert Wehn, Christian Weis, David A. Wood, Hongil Yoon, and Éder F. Zulian. 2020. The gem5 Simulator: Version 20.0+. *CoRR* abs/2007.03152 (2020). arXiv:2007.03152 https://arxiv.org/abs/2007.03152

[28] Julius Mandelblat. 2015. Technology Insight: Intel's Next Generation Microarchitecture Code Name Skylake. In *Intel Developer Forum*.

[29] Clémentine Maurice, Christoph Neumann, Olivier Heen, and Aurélien Francillon. 2015. C5: Cross-Cores Cache Covert Channel. In *Detection of Intrusions and Malware, and Vulnerability Assessment - 12th International Conference, DIMVA 2015, Milan, Italy, July 9-10, 2015, Proceedings (Lecture Notes in Computer Science, Vol. 9148)*, Magnus Almgren, Vincenzo Gulisano, and Federico Maggi (Eds.). Springer, 46–64. https://doi.org/10.1007/978-3-319-20550-2_3

[30] Clémentine Maurice, Nicolas Le Scouarnec, Christoph Neumann, Olivier Heen, and Aurélien Francillon. 2015. Reverse Engineering Intel Last-Level Cache Complex Addressing Using Performance Counters. In *Research in Attacks, Intrusions, and Defenses - 18th International Symposium, RAID 2015, Kyoto, Japan, November 2-4, 2015, Proceedings (Lecture Notes in Computer Science, Vol. 9404)*, Herbert Bos, Fabian Monrose, and Gregory Blanc (Eds.). Springer, 48–65. https://doi.org/10.1007/978-3-319-26362-5_3

[31] Clémentine Maurice, Manuel Weber, Michael Schwarz, Lukas Giner, Daniel Gruss, Carlo Alberto Boano, Stefan Mangard, and Kay Römer. 2017. Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud.. In *NDSS*, Vol. 17. 8–11.

[32] Daniel Molka, Daniel Hackenberg, Robert Schöne, and Wolfgang E. Nagel. 2015. Cache Coherence Protocol and Memory Performance of the Intel Haswell-EP Architecture. In *44th International Conference on Parallel Processing, ICPP 2015, Beijing, China, September 1-4, 2015*. IEEE Computer Society, 739–748. https://doi.org/10.1109/ICPP.2015.83

[33] Saul B. Needleman and Christian D. Wunsch. 1970. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology* 48, 3 (1970), 443–453. https://doi.org/10.1016/0022-2836(70)90057-4

[34] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache Attacks and Countermeasures: The Case of AES. In *Topics in Cryptology – CT-RSA 2006*. Springer Berlin Heidelberg, Berlin, Heidelberg, 1–20. https://doi.org/10.1007/11605805_1

[35] Riccardo Paccagnella, Licheng Luo, and Christopher W. Fletcher. 2021. Lord of the Ring(s): Side Channel Attacks on the CPU On-Chip Ring Interconnect Are Practical. In *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, Michael Bailey and Rachel Greenstadt (Eds.). USENIX Association, 645–662. https://www.usenix.org/conference/usenixsecurity21/presentation/paccagnella

[36] Colin Percival. 2005. Cache missing for fun and profit.

[37] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. 2016. DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks. In *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*, Thorsten Holz and Stefan Savage (Eds.). USENIX Association, 565–581. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/pessl

[38] Antoon Purnal, Lukas Giner, Daniel Gruss, and Ingrid Verbauwhede. 2021. Systematic analysis of randomization-based protected cache architectures. In *42th IEEE Symposium on Security and Privacy*, Vol. 5.

[39] Antoon Purnal, Furkan Turan, and Ingrid Verbauwhede. 2021. Prime+Scope: Overcoming the Observer Effect for High-Precision Cache Contention Attacks. In *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*, Yongdae Kim, Jong Kim, Giovanni Vigna, and Elaine Shi (Eds.). ACM, 2906–2920. https://doi.org/10.1145/3460120.3484816

[40] Moinuddin K Qureshi. 2018. CEASER: Mitigating conflict-based cache attacks via encrypted-address and remapping. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 775–787.

[41] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. 2009. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *Proceedings of the 2009 ACM Conference on Computer and Communications Security, CCS 2009, Chicago, Illinois, USA, November 9-13, 2009*, Ehab Al-Shaer, Somesh Jha, and Angelos D. Keromytis (Eds.). ACM, 199–212. https://doi.org/10.1145/1653662.1653687

[42] Gururaj Saileshwar and Moinuddin K. Qureshi. 2021. MIRAGE: Mitigating Conflict-Based Cache Attacks with a Practical Fully-Associative Design. In *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, Michael Bailey and Rachel Greenstadt (Eds.). USENIX Association, 1379–1396. https://www.usenix.org/conference/usenixsecurity21/presentation/saileshwar

[43] Michael Schwarz, Claudio Canella, Lukas Giner, and Daniel Gruss. 2019. Store-to-Leak Forwarding: Leaking Data on Meltdown-resistant CPUs. *CoRR* abs/1905.05725 (2019). arXiv:1905.05725

[44] Michael Schwarz, Clémentine Maurice, Daniel Gruss, and Stefan Mangard. 2017. Fantastic Timers and Where to Find Them: High-Resolution Microarchitectural Attacks in JavaScript. In *Financial Cryptography and Data Security - 21st International Conference, FC 2017, Sliema, Malta, April 3-7, 2017, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 10322)*, Aggelos Kiayias (Ed.). Springer, 247–267. https://doi.org/10.1007/978-3-319-70972-7_13

[45] Wei Song, Boya Li, Zihan Xue, Zhenzhen Li, Wenhao Wang, and Peng Liu. 2021. Randomized Last-Level Caches Are Still Vulnerable to Cache Side-Channel Attacks! But We Can Fix It. In *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*. IEEE, 955–969. https://doi.org/10.1109/SP40001.2021.00050

[46] Wei Song and Peng Liu. 2019. Dynamically Finding Minimal Eviction Sets Can Be Quicker Than You Think for Side-Channel Attacks against the {LLC}. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*. 427–442.

[47] Chao Su and Qingkai Zeng. 2021. Survey of CPU Cache-Based Side-Channel Attacks: Systematic Analysis, Security Models, and Countermeasures. *Secur. Commun. Networks* 2021 (2021), 5559552:1–5559552:15. https://doi.org/10.1155/2021/5559552

[48] Jakub Szefer. 2019. Survey of Microarchitectural Side and Covert Channels, Attacks, and Defenses. *J. Hardw. Syst. Secur.* 3, 3 (2019), 219–234. https://doi.org/10.1007/s41635-018-0046-1

[49] Qinhan Tan, Zhihua Zeng, Kai Bu, and Kui Ren. 2020. PhantomCache: Obfuscating Cache Conflicts with Localized Randomization.. In *NDSS*.

[50] Andrew S Tanenbaum and David J Wetherall. 1996. *Computer networks*. Prentice hall. I–XVII pages.

[51] Jan Philipp Thoma, Christian Niesler, Dominic A. Funke, Gregor Leander, Pierre Mayr, Nils Pohl, Lucas Davi, and Tim Güneysu. 2021. ClepsydraCache - Preventing Cache Attacks with Time-Based Evictions. *CoRR* abs/2104.11469 (2021). arXiv:2104.11469

[52] Eran Tromer, Dag Arne Osvik, and Adi Shamir. 2010. Efficient Cache Attacks on AES, and Countermeasures. *J. Cryptol.* 23, 1 (2010), 37–71. https://doi.org/10.1007/s00145-009-9049-y

[53] Yukiyasu Tsunoo, Teruo Saito, Tomoyasu Suzaki, Maki Shigeri, and Hiroshi Miyauchi. 2003. Cryptanalysis of DES Implemented on Computers with Cache. In *Cryptographic Hardware and Embedded Systems - CHES 2003, 5th International Workshop, Cologne, Germany, September 8-10, 2003, Proceedings (Lecture Notes in Computer Science, Vol. 2779)*, Colin D. Walter, Çetin Kaya Koç, and Christof Paar (Eds.). Springer, 62–76. https://doi.org/10.1007/978-3-540-45238-6_6

[54] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2019. RIDL: Rogue In-Flight Data Load. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*. IEEE, 88–105. https://doi.org/10.1109/SP.2019.00087

[55] Pepe Vila, Boris Köpf, and José F Morales. 2019. Theory and practice of finding eviction sets. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 39–54.

[56] Zhenghong Wang and Ruby B. Lee. 2007. New cache designs for thwarting software cache-based side channel attacks. In *34th International Symposium on Computer Architecture (ISCA 2007), June 9-13, 2007, San Diego, California, USA*, Dean M. Tullsen and Brad Calder (Eds.). ACM, 494–505. https://doi.org/10.1145/1250662.1250723

[57] Mario Werner, Thomas Unterluggauer, Lukas Giner, Michael Schwarz, Daniel Gruss, and Stefan Mangard. 2019. ScatterCache: Thwarting Cache Attacks via Cache Set Randomization. In *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, Nadia Heninger and Patrick Traynor (Eds.). USENIX Association, 675–692. https://www.usenix.org/conference/usenixsecurity19/presentation/werner

[58] Zhenyu Wu, Zhang Xu, and Haining Wang. 2012. Whispers in the Hyper-space: High-speed Covert Channel Attacks in the Cloud. In *Proceedings of the 21th USENIX Security Symposium, Bellevue, WA, USA, August 8-10, 2012*, Tadayoshi Kohno (Ed.). USENIX Association, 159–173. https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/wu

[59] Jidong Xiao, Zhang Xu, Hai Huang, and Haining Wang. 2012. A Covert Channel Construction in a Virtualized Environment. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security* (Raleigh, North Carolina, USA) *(CCS '12)*. Association for Computing Machinery, New York, NY, USA, 1040–1042. https://doi.org/10.1145/2382196.2382318

[60] Yunjing Xu, Michael Bailey, Farnam Jahanian, Kaustubh R. Joshi, Matti A. Hiltunen, and Richard D. Schlichting. 2011. An exploration of L2 cache covert channels in virtualized environments. In *Proceedings of the 3rd ACM Cloud Computing Security Workshop, CCSW 2011, Chicago, IL, USA, October 21, 2011*, Christian Cachin and Thomas Ristenpart (Eds.). ACM, 29–40. https://doi.org/10.1145/2046660.2046670

[61] Mengjia Yan, Read Sprabery, Bhargava Gopireddy, Christopher W. Fletcher, Roy H. Campbell, and Josep Torrellas. 2019. Attack Directories, Not Caches: Side Channel Attacks in a Non-Inclusive World. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*. IEEE, 888–904. https://doi.org/10.1109/SP.2019.00004

[62] Yuval Yarom and Katrina Falkner. 2014. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014*, Kevin Fu and Jaeyeon Jung (Eds.). USENIX Association, 719–732. https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom