Double Trouble: Combined Heterogeneous Attacks on Non-Inclusive Cache Hierarchies

Antoon Purnal*
imec-COSIC, KU Leuven

Furkan Turan*
imec-COSIC, KU Leuven

Ingrid Verbauwhede
imec-COSIC, KU Leuven

*Equal contribution joint first authors

Abstract

As the performance of general-purpose processors faces diminishing improvements, computing systems are increasingly equipped with domain-specific accelerators. Today's high-end servers tightly integrate such accelerators with the CPU, e.g., giving them direct access to the CPU's last-level cache (LLC).

Caches are an important source of information leakage across security domains. This work explores *combined* cache attacks, complementing traditional co-tenancy with control over one or more accelerators. The constraints imposed on these accelerators, originally perceived as limitations, turn out to be advantageous to an attacker. We develop a novel approach for accelerators to find eviction sets, and leverage precise double-sided control over cache lines to expose undocumented behavior in non-inclusive Intel cache hierarchies.

We develop a compact and extensible FPGA hardware accelerator to demonstrate our findings. It constructs eviction sets at unprecedented speeds ($< 200\,\mu s$), outperforming existing techniques with one to three orders of magnitude. It maintains excellent performance, even under high noise pressure. We also use the accelerator to set up a covert channel with fine spatial granularity, encoding more than 3 bits per cache set. Furthermore, it can efficiently evict shared targets with tiny eviction sets, refuting the common assumption that eviction sets must be as large as the cache associativity.

1 Introduction

Heterogeneous computing yields great increases in performance and energy efficiency with specific processing capabilities for certain tasks. Recently, FPGAs have emerged in datacenters for providing such capabilities. They can be used to accelerate wide-scale data center services, such as machine learning applications. More interestingly, cloud service providers give control of FPGAs to customers, who can implement custom accelerators and integrate them into their applications. After initial steps by Amazon's AWS, which already allows users to rent FPGA-supported instances, AMD and Intel acquired the two main FPGA manufacturers (resp. Xilinx and Altera). The aim is to aid customers in their transition through easy and low-overhead integration of software and hardware. As of yet, their security is not fully mature [57]. At the same time, economic incentives attract infrastructure providers to multi-tenancy, i.e., hosting multiple (distrusting) entities on the same physical machine. This work evaluates the impact of heterogeneous multi-tenancy on the quintessential shared hardware component: the cache hierarchy.

Caches play a fundamental role in high-performance computing. By serving the majority of memory requests from fast levels of storage close to the processor (CPU), they overcome the bottleneck caused by comparatively slow memory. Equally fundamental, however, is the timing side channel they introduce, as access latencies depend on access patterns of co-located processes. While some attack techniques rely on cache flushes and shared memory [14, 70], others work with contention [21, 33]. To determine the access patterns of the victim, contention-based attacks employ so-called *eviction sets*, i.e., sets of addresses that contend for cache resources.

Over time, the cache side channel was proven effective to extract keys from cryptographic implementations [3, 17, 41, 70], retrieve user input [13, 40, 50], or infer kernel secrets [12, 16, 23]. Caches have also been used to establish covert channels [33, 37] and are a key enabler of recent transient execution attacks [25, 31]. The ongoing switch to non-inclusive cache hierarchies for high-end CPUs was believed to thwart several attack classes, but this belief has been disproven [66]. However, non-inclusive hierarchies remain relatively unexplored compared to their inclusive counterparts.

The lion's share of the cache attack literature considers CPU processes targeting other processes. Recent studies introduced some heterogeneity, whether it be peripheral devices attacking CPU processes [11, 27, 62], or CPU processes attacking peripherals [55]. This work identifies combined microarchitectural attacks as a threat deserving further examination (cf. Table 1). It is becoming increasingly common to control multiple entities (i.e., devices), which differ in computational capabilities and access to the memory subsystem.

Table 1: Positioning of threat models considered in this work

Victim / Attacker	CPU	Secondary
CPU	Last-Level Cache [21, 33, 70] Coherence Directory [66] **This work**	Packet Chasing [55]
Secondary	Grand Pwning Unit [11] JackHammer [62]	NetCAT [27]
Combined	**This work**	

Intel's Data-Direct IO (DDIO) [18] is a prominent technology that gives PCIe devices direct access to the CPU's last-level cache (LLC). For instance, existing FPGA-accelerated cloud platforms make use of PCIe-based FPGA accelerator cards. DDIO provokes an interesting dynamic in the cache hierarchy, which CPU processes observe through the lens of their private caches, whereas PCIe (DDIO) devices, from now on referred to as *secondary devices*, observe it directly through the LLC. This double view is especially interesting in non-inclusive cache hierarchies, where the LLC is less amenable to direct interaction [66]. This work investigates the collusion of microarchitectural attackers in heterogeneous systems, applied to the widely-used DDIO technology, which already has been shown to bear security implications [27, 55, 62].

In light of the growing interest in heterogeneous computing, this paper seeks to answer the following questions:

How precisely can combined attackers control shared cache state? Can the constraints imposed on DDIO devices turn out to be advantageous? Do common assumptions remain valid in the face of combined attackers?

In this paper, we study combined heterogeneous attacks on emerging non-inclusive Intel cache hierarchies. We identify a set of properties governing the interaction between the CPU and DDIO devices. Attacks originating from secondary devices, e.g., network cards [27] or FPGAs [62], perceive these properties as limitations. For combined attackers, who can dispatch between CPU and secondary device, they enable otherwise infeasible techniques. Ultimately, this leads us to challenge common assumptions, as summarized in Table 2. When relevant, we instantiate secondary devices with FPGAs.

Contributions. Summarized, our main contributions are:

- We explore key primitives for combined cache attacks and discover a new DDIO-related structure in the LLC.
- We develop a fast and reliable procedure for secondary devices to find eviction sets in non-inclusive Intel caches.
- We leverage precise LLC manipulation for reliable eviction with fewer congruent addresses than there are ways.
- We design an FPGA accelerator that implements the aforementioned techniques and make it openly available:

https://github.com/KULeuven-COSIC/Double-Trouble

Table 2: Challenging common understanding

Common Understanding	Our Finding
Eviction set construction reached speed limits [60]	**Accelerating Eviction Set Construction (Section 4)**
Secondary devices only allocate to DDIO region in LLC [18, 27, 62]	**Discover undocumented DDIO$^+$ region (Section 5)**
Non-inclusive LLC: needs directory conflicts [66]	**Eviction from private cache through LLC (Section 6.1)**
Cannot evict from remote socket without `flush` [22, 68]	**Flushless cache attacks across sockets (Section 6.1)**
Minimal eviction set is as large as associativity [33, 60]	**Reliable Eviction with Tiny Eviction Sets (Section 6.2)**
Amplitude-based encoding precluded by self-eviction [37]	**Modulation and Multi-bit Symbols (Section 8.2)**

2 Background

2.1 Heterogeneous Computing

As the limits of general-purpose computers are pushed, domain-specific computation gains importance. Companies have started playing games with custom chips, combining CPUs with accelerators, e.g., for machine learning or networking. Instead of inefficiently increasing CPU core counts, these architectures complement CPUs with custom accelerators, offering high-performance computation, often at low power. Popular examples are Google's TPU and Apple's M1.

On the server side, FPGA-attached CPUs serve the grounds to play this game. With their hardware programmability, FPGAs allow users to implement custom accelerators for their specific needs. Some cloud providers (e.g., AWS) already provide homemade accelerators to customers, or a marketplace where accelerators can be sold or rented. CPU giants have acquired FPGA manufacturers (Intel-Altera, AMD-Xilinx), and are working on tight integration of CPUs and FPGAs.

Today, these platforms attach FPGAs to CPUs as PCIe accelerator cards. On the CPU side, kernel drivers and APIs enable applications to communicate with their hardware accelerators. On the FPGA side, Control and Status Registers (CSRs) offer basic data transfers, and Direct Memory Access (DMA) allows the FPGA to access system memory. In this paper, we focus on the latter, as it interacts with the CPU memory subsystem. Although we focus on FPGA-specific terminology, many conclusions carry over to other PCIe-connected devices, e.g., Network Interface Cards (NICs) or Thunderbolt.

2.2 Cache Organization

Modern cache hierarchies comprise multiple levels. Lower cache levels are closer to the CPU, and are usually smaller and faster than higher levels. Typical Intel processors have three cache levels, with L1 and L2 caches private to each core, and the last-level cache (L3, or LLC) shared between cores.

Caches are organized as arrays of *cache lines* of, typically, 64 bytes. Most caches are *set-associative*, meaning that they are partitioned in *sets*. Each cache line maps to exactly one set, based on an indexing function applied to their address. The *associativity* refers to the number of lines that can reside simultaneously in the same set, i.e., the number of *ways*, *W*.

When a requested line is not present in a cache (i.e., a *cache miss*), it is usually installed after propagating the request to the next level. In the absence of empty ways, the *cache replacement policy* determines the line to *evict* in favor of the incoming one. Lines mapped to the same set are *congruent*. Contending for the same resource, they can evict each other.

The *inclusion* invariants of the cache hierarchy determine whether cache lines can reside simultaneously in multiple levels. A cache is *inclusive* w.r.t. another (lower-level) cache if every line in the latter must also be present in the former. *Exclusive* caches cannot have lines in common. Caches that do not satisfy either invariant are *non-inclusive*. Historically, Intel LLCs are inclusive, but to keep up with increasing core counts, non-inclusive LLCs are becoming commonplace [38].

Contemporary LLCs are partitioned in *slices*, with an undocumented and architecture-dependent mapping [36]. For large core counts, the slices are interconnected with a mesh architecture [38]. High-end systems can have multiple CPU sockets, connected with a coherent memory hierarchy.

2.3 Data-Direct IO (DDIO)

Direct Cache Access (DCA) [15] is a mechanism developed for the fast exchange of Ethernet frames between CPUs and Network Interface Cards (NICs). Instead of accessing main memory, NICs interact directly with the CPU's LLC to alleviate memory bottlenecks and cache thrashing, improving I/O performance [9, 10, 19, 29, 34]. DDIO [18] is Intel's implementation of DCA, available on server-grade CPUs. It is enabled on such CPUs by default, and PCIe devices (NICs, FPGAs, etc.) transparently interact with the LLC instead of memory.

Unfortunately, specific DDIO behavior is largely undocumented, especially for non-inclusive cache hierarchies. Some works partially reverse-engineer it [27, 62]. Section 3.2 covers known and yet unknown DDIO behavior in detail.

2.4 Cache Attacks

The observation that the execution time of a program depends on its control flow and interaction with the cache hierarchy characterized the first generation of cache attacks [3, 26, 43]. Later, the case was made for co-located attackers, i.e., those running code on the same physical platform as potential victims. By manipulating and observing the cache state, they observe much more fine-grained access patterns [41, 44].

Arguably the strongest technique is Flush+Reload, where the attacker flushes a target line from the cache (e.g., using `clflush` on `x86`), and later reads it to determine whether the victim accessed it in the meantime. In the absence of `clflush` [30], an attacker can evict the shared line instead, which is referred to as Evict+Reload [13]. Both techniques require shared memory with the victim (e.g., KSM [2]). In contrast, Prime+Probe only relies on cache contention. In particular, the attacker occupies an entire cache set with her own lines, waits, and afterwards loads these lines again. If another process has accessed lines congruent to those of the attacker, this will be reflected in the attacker's access latency. Due to its low requirements, Prime+Probe has been mounted from restricted environments [27, 40].

The target cache needs to be shared between attacker and victim. Initial attacks considered same-core attackers and targeted the L1 cache [41, 44]. Later attacks managed to target the LLC [21, 33, 70], enlarging the threat to cross-core attacks.

Until recently, cross-core Evict+Reload and Prime+Probe relied explicitly on the inclusive nature of the LLC. In this case, eviction from the LLC implies invalidation in all lower-level caches to preserve the inclusion invariant [21, 33]. This allows to evict lines from other cores' private caches.

Non-inclusive Caches. In non-inclusive caches, lines in L2 are not necessarily present in the LLC. In fact, they rarely are, since loads from memory are installed in L1/L2, skipping the LLC. Conversely, contention on the LLC alone does not invalidate lines in the lower-level caches of other cores. To overcome this problem for non-inclusive Intel CPUs, Yan et al. [66] propose contention on the *coherence directory* (CD), also referred to as the *snoop filter*. The CD tracks lines present in lower-level caches, and is inclusive to accelerate coherence transactions with other cores [72]. On Intel CPUs, the LLC and CD share the same slice and set mapping.

This paper considers cross-core attacks in non-inclusive Intel caches. The targets are the LLC and CD, and lines are *congruent* when they share the same LLC/CD set and slice.

2.5 Eviction Set Construction

Prior to a Prime+Probe or Evict+Reload attack, the attacker constructs *eviction sets*, i.e., sets of congruent addresses. If physical addresses and their mapping to LLC/CD sets and slices are known, finding congruent addresses is trivial.

In practice, however, the attacker is limited on both fronts. First, unprivileged processes observe virtual addresses, organized in 4 KiB or 2 MiB pages, and do not know the virtual-

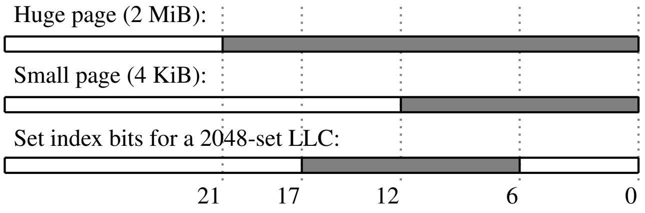


Figure 1: Control over cache set index depends on page sizes

Table 3: Platforms Used for Experimentation

Platform	CPU *Intel Xeon*	Arch.	Core Count	FPGA *Intel PAC*
ACE 1	Plat. 8180	SKL-SP	28	A10
ACE 2	Plat. 8280	CLX-SP	28	S10 (x2)
Local	Silver 4208	CLX-SP	8	A10 (x2)

SKL: Skylake, CLX: Cascade Lake

Cache	Info	Ways	Size *per Core*
L1	Core-Private	8	32 KB
L2	Core-Private	16	1 MB
LLC	Shared, Non-Inclusive	11	1.375 MB

to-physical address translation. Therefore, physical address control is limited to the page frame bits, which are unaffected by translation (cf. Figure 1). Second, the slicing function is undocumented and architecture-dependent [36, 66].

Liu et al. [33] construct eviction sets for inclusive LLCs. Vila et al. [60] accelerate it by improving the time complexity from quadratic to linear. Yan et al. [66] find LLC/CD eviction sets in non-inclusive Intel caches. To that end, they introduce helper sets that are congruent in L2 but not the LLC. Techniques for non-inclusive caches are currently underdeveloped w.r.t. inclusive caches. Moreover, because of the indirect interaction with the LLC, their noise-resilience is unclear.

2.6 Experimental Setup

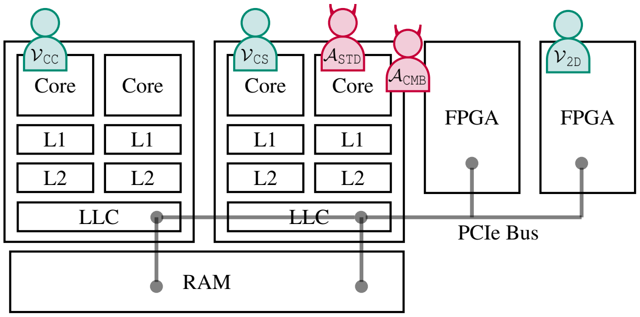
We work remotely on Intel Labs (IL) Academic Compute Environment (ACE), with dual-socket Xeon Platinum CPUs (28 cores/slices per socket). We also use a local lab setup, with dual-socket Xeon Silver CPUs (8 cores/slices per socket). All platforms have non-inclusive LLCs. The platforms utilize Intel's PCIe-based FPGA accelerator cards called Programmable Acceleration Cards (PACs), either with Arria 10 (A10) or Stratix 10 (S10) family FPGAs. Table 3 summarizes the platforms and their cache hierarchy.

A basic FPGA design can transparently interact with the memory subsystem over DDIO. At a high level, it can read and write to memory, and distinguish between access latencies (L2/LLC/RAM) based on immutable timing sources. A detailed description of our implementation is deferred to Section 7.

3 Double Trouble: Combined Cache Attacks

3.1 Threat Model

The main threat model in this work is the combined attacker ($\mathcal{A}_{\texttt{CMB}}$). As indicated in Figure 2, she controls at least one CPU core and a secondary device connected over DDIO. In our case, an FPGA is used as the secondary device. The attacker can dispatch operations to software and hardware, and share memory between them. For completeness, we also consider traditional attackers ($\mathcal{A}_{\texttt{STD}}$) without a secondary device.



$\mathcal{V}_{\texttt{CC}}$
Core
Core
L1
L1
L2
L2
LLC
$\mathcal{V}_{\texttt{CS}}$
$\mathcal{A}_{\texttt{STD}}$
$\mathcal{A}_{\texttt{CMB}}$
Core
Core
L1
L1
L2
L2
LLC
FPGA
$\mathcal{V}_{\texttt{2D}}$
FPGA
PCIe Bus
RAM

Figure 2: Combined attackers control a CPU process and secondary device. We consider three victim types ($\mathcal{V}_{\texttt{CC}}$,$\mathcal{V}_{\texttt{CS}}$,$\mathcal{V}_{\texttt{2D}}$).

The attacker has no privileges and does not know the slice mapping. We do not assume the availability of a `clflush` instruction (in accordance to, e.g., [66]). Although our findings do not strictly require huge memory pages, we assume them to be available, as they are enabled by default on server-grade platforms with FPGA acceleration (e.g., OPAE [20]).

To navigate the heterogeneity in attacker and victim properties, Table 4 summarizes our main results and indicates the configurations to which they apply. We distinguish between the degree of co-location: attacker and victim running on different cores ($\mathcal{V}_{\texttt{CC}}$), on different sockets ($\mathcal{V}_{\texttt{CS}}$), or a victim secondary device attached to the attacker socket ($\mathcal{V}_{\texttt{2D}}$).

Section 4 introduces a new algorithm for swift and reliable eviction set construction, overturning the limitations of secondary devices. Section 5 exposes undocumented behavior in the LLC, which we use to obtain an intra-cache-set granularity. Section 6 shows how standard attackers can evict shared lines without CD contention, and how combined attackers can do so with tiny eviction sets. These results require shared memory between the attacker and victim. Section 7 describes our implementation of an FPGA hardware accelerator, which is evaluated in Section 8.

Table 4: Applicability of the main results of this work

Capabilities	Attacker		Victim			Shared Memory
	$\mathcal{A}_{\texttt{STD}}$	$\mathcal{A}_{\texttt{CMB}}$	$\mathcal{V}_{\texttt{CC}}$	$\mathcal{V}_{\texttt{CS}}$	$\mathcal{V}_{\texttt{2D}}$	
Eviction Set Finding		✓	✓	✓	✓	
Intra-set Granularity		✓	✓		✓	
Eviction without CD	✓	✓	✓	✓		✓
Reduced Eviction		✓	✓	✓		✓

3.2 Key Properties

3.2.1 Spatially Limited Interaction With LLC

To prevent thrashing, DDIO devices only interact with a fraction of the last-level cache (LLC) [18]. Lines *read* by the secondary device are not allocated in the LLC [27, 62], but they are served from the LLC if already present. Lines *written* by the secondary device are allocated, but only to a limited number of ways (two by default) in every set [10, 27].

In contrast to previous suggestions [18, 27], we find that the replacement policy is not LRU (cf. rationale in Appendix C).

Non-Default DDIO Configurations. The number of LLC ways to which DDIO can write-allocate is, by default, two [10, 27]. However, this can be configured in the `IIO_LLC_WAYS` Model Specific Register (MSR) [10, 32], with a bitmask that represents the cache ways used by DDIO. The minimal and default setting is `0x600`. More ways can be activated by setting more bits, provided that the selected ways are consecutive. In this paper, we denote the number of DDIO ways as D.

The default configuration ($D = 2$) is, arguably, the most important to study from a security point of view. In accordance with prior studies [27, 62], we focus our attention on this configuration, and assume it unless otherwise indicated. However, when relevant, we generalize our findings to $2 \leq D \leq W$.

Property #1: Spatially limited LLC interaction.
Secondary devices interact directly with the LLC. Only writes trigger cache line placement, which is statically constrained to a limited number of ways (two by default).

3.2.2 Reading Without Consequences

The property that secondary devices do not read-allocate in the cache has an underappreciated corollary: it allows attackers to *query the cache state without disturbing it*. We illustrate the implications of this power with two relevant examples.

Example: Counting LLC Entries. Consider how to infer how many (out of N) lines are cached in the LLC. This can be done by measuring the access latency of all N lines, counting those within the predetermined LLC timing range.

CPU-only attackers are limited in their accuracy. Consider the case where at least one line is not cached. Measuring the access latency of this line allocates it in the cache. In this process, it may evict the other lines, perturbing the measurement.

The combined attacker, in contrast, can measure the cache state *reliably*, i.e., a partial measurement does not endanger the validity of the full measurement, and *repetitively*, i.e., several measurements of the same state can be combined.

Example: Eviction Candidate. For lines already in the LLC, do secondary reads influence the *eviction candidate*, i.e., the line to be evicted upon installation of a new congruent line?

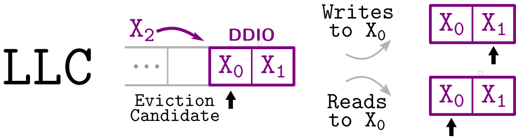


Figure 3: Secondary reads do not change eviction candidates.

Consider Figure 3, where X_0, X_1 and X_2 are congruent lines. X_0 and X_1 are placed in the DDIO region with secondary writes. X_1 is written repeatedly to ensure that X_0 is the eviction candidate. Indeed, we observe that writing X_2 evicts X_0.

In a first experiment, the secondary device *writes* a few times to X_0. We observe that placement of X_2 now evicts X_1, so the earlier write to X_0 changed the eviction candidate to X_1. The second experiment performs several *reads* of X_0. If the replacement policy records these reads, the eviction candidate should change to X_1. Still, we find that placing X_2 always evicts X_0. We conclude that secondary reads, in contrast to writes, do not influence the LLC replacement policy state.

Finding #2: Non-destructive secondary reads.
Secondary reads are non-destructive. Reading uncached lines does not trigger cache allocation, and reading LLC lines does not influence their replacement policy state.

3.2.3 Two-sided Cache Hierarchy Manipulation

Combined attackers can approach non-inclusive cache hierarchies from two sides. We now cover known and novel primitives to trigger movement between L2[1] and the LLC. Moving or copying lines to the LLC is useful, as lines evicted from the LLC are invalidated in all cache levels [66]. Moving lines from the LLC to L2 is useful to invalidate specific LLC ways. Our repository supports these findings with experiments.

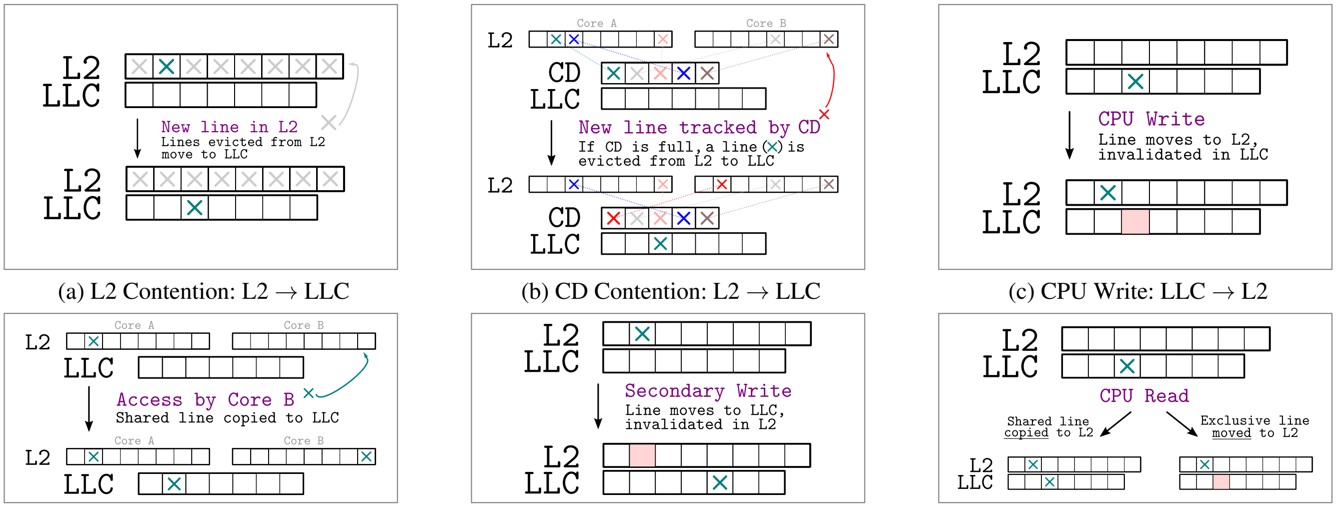
L2 to LLC. The most straightforward way to move an L2 line to the LLC is to access sufficient addresses mapped to the same L2 set (cf. Figure 4a). This technique can only be used by processes running on the core tied to the specific L2 cache.

Yan et al. [66] demonstrate the eviction of lines from remote private caches, i.e., L2 caches associated with other cores. They do this by generating contention on the coherence directory CD (cf. Figure 4b), the inclusive structure co-located with the LLC that keeps track of lines in L2 caches.

On our test platforms, we observe that when a line is accessed by two processes on different cores, it is copied to the LLC. The line then co-exists in the LLC and both private L2 caches (cf. Figure 4d). We find this novel technique to be accurate and simple to move a target line to the LLC. However, note that this primitive is limited to attacker-readable lines.

Specifically for the combined attacker, there is yet another primitive available (cf. Figure 4e). The secondary device writes the line, which moves it to the LLC. Now dirty, the line is invalidated from all other caches to maintain coherence. Note that this primitive is limited to attacker-writable lines.

[1] Since L2 is inclusive w.r.t. L1, for our purposes they can be consolidated.


L2
LLC
New line in L2
Lines evicted from L2 move to LLC
Core A
Core B
CD
New line tracked by CD
If CD is full, a line (×) is evicted from L2 to LLC
CPU Write
Line moves to L2, invalidated in LLC
Access by Core B
Shared line copied to LLC
Secondary Write
Line moves to LLC, invalidated in L2
CPU Read
Shared line copied to L2
Exclusive line moved to L2

(a) L2 Contention: L2 → LLC

(b) CD Contention: L2 → LLC

(c) CPU Write: LLC → L2

(d) Shared Access: L2 → LLC

(e) Secondary Write: L2 → LLC

(f) CPU Read: LLC → L2 (or → L2&LLC)

Figure 4: Techniques for combined attackers to manipulate the cache hierarchy.

LLC to L2. A line can be moved from the LLC to L2 by read or write requests from CPU cores. Lines that are written (Figure 4c) move from the LLC to L2, invalidating the LLC line for coherence purposes [67]. If the line is read (Figure 4f), we find that lines that only exist in the LLC (e.g., in *modified* or *exclusive* state) move back to L2 and are invalidated in the LLC. In contrast, lines present in multiple caches (e.g., in *shared* or *forward* state) are *copied* to L2, i.e., they remain allocated in the LLC. As these observations are partially inconsistent with prior work [66, 67], we share our rationale in Appendix B.

Replacement policies prefer to install incoming lines in empty ways (if present) to avoid unnecessary eviction of useful lines. Some primitives are able to produce empty ways by invalidation. We refer to these ways as *magnet ways*.

Finding #3: Precise manipulation of cache hierarchy. Combined attackers can accurately migrate lines between cache levels, and invalidate them from selected caches.

4 Fast Eviction Set Finding using DDIO

The eviction set construction problem is the following: given a target cache line, find EV congruent lines in the designated cache (of associativity W). We review existing algorithms for the LLC, and expose a new method for combined attackers, enabled by limitations of secondary devices (**#1**, **#2**).

4.1 Reduction Algorithms

Figure 5 depicts the structure of traditional eviction set construction algorithms. First, an initial set of candidate addresses is constructed [33], for which the size depends on attacker control over physical address bits [60]. Then, it is reduced to a *minimal* eviction set, i.e., a set, typically of size $\text{EV} \geq W$, that no longer contains any non-congruent addresses.

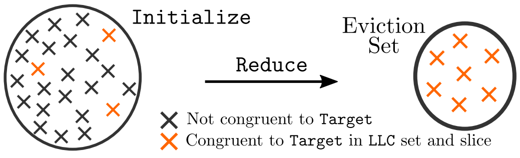


Figure 5: Traditional reduction-based algorithms

The reduction is an iterative procedure based on a *congruence test* that removes a portion of the current set and tests whether the remainder still evicts the target. If so, the portion is not necessary for eviction and can be discarded. Initial algorithms for inclusive caches [21, 33, 40] remove one element at a time, leading to quadratic complexity in the initial set size. Vila et al. [60] propose to perform the congruence test on groups of addresses instead, achieving linear complexity.

Yan et al. [66] develop a reduction algorithm for non-inclusive Intel LLCs with a custom congruence test (cf. Section 2.4). Their algorithm has quadratic complexity, but may be amenable to similar improvements [60]. However, it appears to need several eviction and measurement iterations to cope with the complications of non-inclusive caches.

4.2 Acceleration with `Discover-Expand`

Secondary devices generally perceive a smaller LLC associativity (**#1**), with $D = 2$ by default. As a result, $D+1$ congruent addresses are sufficient to manifest contention. In addition, the non-destructive reads (**#2**) enable a congruence test that determines whether a single address *is* congruent with a target, instead of whether a pool of addresses *contains* some that are congruent. These properties lead to an effective search algorithm, based on expansion rather than reduction (cf. Figure 6).

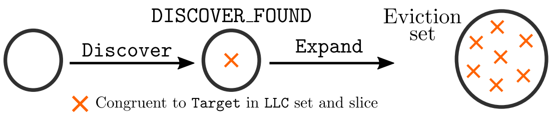


Figure 6: Expanding eviction set construction. For $D=2$, Discover only needs to find a single congruent address.

The algorithm takes as input a target address (TARGET), desired eviction set size (EV), and stride representing attacker-controllable address bits (STRIDE). The output is an eviction set, i.e., EV addresses that are mapped to the same LLC set and slice as TARGET. The algorithm does not build an initial set, and comprises two phases: Discover obtains the first $D-1$ congruent addresses, and Expand completes the eviction set.

Algorithm 1 Eviction Set Construction: Discover finds the first $D-1$ congruent addresses, Expand finds the others

Input: TARGET: an address for the eviction set, EV: desired eviction set size, STRIDE: indicates attacker-controlled bits
Output: es eviction set

```
             1:  es[] ← empty list                          a new address from
            ┌ 2:  do                                         search space with
            │ 3:      secondary_write(TARGET)                STRIDE
            │ 4:      secondary_write(es[0,1,..,len(es)-1])
            │ 5:      do
  Discover  ┤ 6:          TEST_ADDRESS ← new candidate
            │ 7:          secondary_write(TEST_ADDRESS)
            │ 8:      while secondary_read(TARGET) is fast
            │ 9:      append TEST_ADDRESS to es          DISCOVER_FOUND:
            └10:  while len(es) < D-1                    addresses found
            ┌11:  do                                     in Discover,
            │12:      secondary_write(TARGET)            es[0,1,..,D-2]
            │13:      secondary_write(es[0,1,..,D-2])
            │14:      do
  Expand    ┤15:          TEST_ADDRESS ← new candidate
            │16:          secondary_write(TEST_ADDRESS)
            │17:      while secondary_read(TARGET) is fast
            │18:      append TEST_ADDRESS to es
            └19:  while len(es) < EV
```

4.2.1 Discover: Finding the First $D-1$ Addresses

First, the Discover phase writes TARGET, installing it in the LLC's DDIO ways. Then, it iteratively writes a new candidate address and afterwards measures the access latency of TARGET. If the latter has been evicted (from LLC to RAM), the candidate is determined to be congruent with it.

As the DDIO associativity is D, it is expected that $D-1$ congruent test addresses are overlooked for every congruent candidate that is actually added to the eviction set. However, as soon as $D-1$ elements have been found (a set denoted as DISCOVER_FOUND), the Discover phase can terminate in favor of the more efficient Expand phase.

4.2.2 Expand: Finding More Addresses

The Expand phase writes TARGET and DISCOVER_FOUND to occupy all D DDIO ways. Then, similar to Discover, new candidates are written, each time measuring the access latency of TARGET to determine whether the candidate is congruent. This step is repeated to obtain the full eviction set.

4.2.3 Algorithmic Complexity

The stride of the search depends on control over physical address bits (cf. Section 2.5) and the number of LLC/CD slices (cf. Section 2.2). As the slice mapping is unknown, candidates can, at best, be selected based on their LLC set index bits.

Our test LLCs have 2048 sets with 8 or 28 slices (SLICES). Huge pages allow to configure all set index bits, so the congruence probability for candidate addresses is SLICES^{-1}, assuming lines are distributed among slices uniformly at random.

The congruence probability directly relates to the expected number of candidates to test in the Expand phase. In the Discover phase, however, the first $D-1$ congruent candidates may be missed. In first order, the expected number of candidates to test is $(D-1)\cdot[D\cdot\mathsf{SLICES}]+(W-D+1)\cdot[\mathsf{SLICES}]$. Section 8.1 evaluates accuracy and speed in practice.

Algorithm 2 Verification algorithm to check congruence

Input: TARGET, DISCOVER_FOUND, CANDIDATE
Output: boolean: are the inputs congruent?

```
1:  secondary_write(TARGET);                TARGET and CANDIDATE
2:  secondary_write(DISCOVER_FOUND);        are interchanged in
3:  secondary_write(CANDIDATE);             consecutive runs
4:  return true if secondary_read(TARGET) is fast
```

4.2.4 Low-level Aspects

Replacement Policy. The simplified algorithm assumes LRU replacement. However, we find that repeated writes influence the eviction candidate (cf. Appendix C). To ensure that TARGET is the eviction candidate, the accelerator performs the secondary_write on lines 4/13 of Algorithm 1 twice.

Verification. To increase robustness against false positive errors (e.g., due to noise), Algorithm 2 optionally performs additional checks. It is called with TARGET, DISCOVER_FOUND and the address to be tested, and performs two permuted verification tests. This way, a false positive candidate will only pass if there is noise in two different cache sets.

As the verification needs $D+1$ addresses (including TARGET), the accuracy of the DISCOVER_FOUND set itself is confirmed together with the first address of the Expand phase. If it fails, both addresses are discarded and Discover is restarted. As soon as DISCOVER_FOUND is verified, all subsequent addresses can be verified in isolation. To increase confidence, multiple verification repetitions can be performed.

Table 5: Access sequences to cache line L to prepare its state, with the resulting cache level for L, whether L is modified, and whether a secondary write allocates L to the DDIO region

Access Sequence	Level	Modified	DDIO
① `sw_flush(L)`	RAM		✓
② `sw_write(L)`	L2	✓	
③ `sw_flush(L); sw_read(L)`	L2		✓
④ `hw_write(L); sw_read(L)`	L2	✓	

(`sw_*` and `hw_*` denote actions by CPU and secondary device, resp.)

Write Limitation. The routine requires write access to TARGET. However, an LLC-congruent address to the intended target can be used as input to the algorithm (cf. Section 8.3).

5 Structure of the LLC Set

This section revisits the interaction between DDIO devices and the LLC (**#1**). Secondary writes to uncached lines are known to allocate in the LLC, specifically to one of the DDIO ways [10, 27, 62]. It is worth investigating whether this behavior holds for lines that are already cached, e.g., in L2. Contrary to expectation, it appears to depend on the state of the line.

Another Region in the LLC Set. Consider a test set of N LLC-congruent addresses ($N \geq 2$). To fix their cache level and state, the elements of the test set first undergo the access sequences (① - ④) as given by Table 5. Then, they are written by the secondary device. Finally, the secondary device counts how many remain in the LLC (cf. Section 3.2.2). All read and write operations are repeated to remove the influence of the replacement policy, and we consider 10 000 measurements.

For all access sequences, we observe that subsequent secondary writes allocate to the LLC, consistent with current understanding. However, consider what happens as we now apply contention to the DDIO region. This contention is achieved with secondary writes to another uncached contention set (access sequence ①, which is known to lead to DDIO allocation).

For sequences ① and ③, no test lines remain after DDIO contention (i.e., they were allocated to the DDIO region). However, for sequences ② and ④, both addresses remain in the LLC (i.e., they must have been placed *elsewhere*). In conclusion, cache lines in specific states are secondary write-allocated to the LLC, but *outside of the DDIO region.*

In the remainder of this work, we refer to this unknown region as the DDIO$^+$ region. We can only speculate on the condition to be placed in this region, but we observe modified lines to be assigned to it (e.g., sequences ② and ④). The rationale for this behavior is not immediately obvious and we assume it to be an undocumented performance optimization.

Associativity. We now infer the structure of the LLC set, starting with the associativity of the DDIO and DDIO$^+$ subregions.

For the DDIO region, the secondary device writes N congruent addresses and counts how many remain in the LLC. The associativity is the largest N for which none are consistently evicted. As expected, we observe an associativity of D.

For the DDIO$^+$ region, we perform a similar experiment, but with access sequence ④ preceding the secondary write. We observe a DDIO$^+$ associativity of 2, irrespective of D.

The LLC set. To learn which ways belong to the DDIO/DDIO$^+$ region on the Xeon Silver, we extend the mapping technique of Farshin et al. [10]. In particular, it uses Intel CAT [19] to let a software process evict specific LLC ways as specified by a bitmask. The correlation between the bitmask and evictions of DDIO/DDIO$^+$ lines reveals the composition of these regions.

For the DDIO region, we use a CAT mask that spans D ways, i.e., the DDIO associativity. First, the secondary device writes D test lines. Then, software generates contention in the ways specified by the mask. Finally, the secondary device counts the test lines in the LLC. The mask corresponding to the D leftmost ways results in all test lines being evicted; shifting it one to the right evicts $D-1$ lines, etc. No lines are evicted if the mask does not overlap with any of the D leftmost ways.

For the DDIO$^+$ region, we use a similar methodology with a 2-way CAT mask. Access sequence ④ is used to produce lines in the DDIO$^+$ state. The CAT mask corresponding to the 2 rightmost ways results in both DDIO$^+$ lines being evicted.

LLC Model. Figure 7 summarizes the inferred LLC structure for our local platform (Xeon Silver). CPU memory traffic allocates to all ways in the set. Secondary devices only write-allocate to the DDIO or DDIO$^+$ regions (depending on the state of the written line). The DDIO region is contiguous and has associativity D, growing from the most-significant ways. The DDIO$^+$ region is contiguous and has associativity 2, and covers the least-significant ways. In the event that the DDIO and DDIO$^+$ regions overlap (i.e., for $D \geq 10$), the least-significant ways accommodate both lines in the DDIO and DDIO$^+$ state.

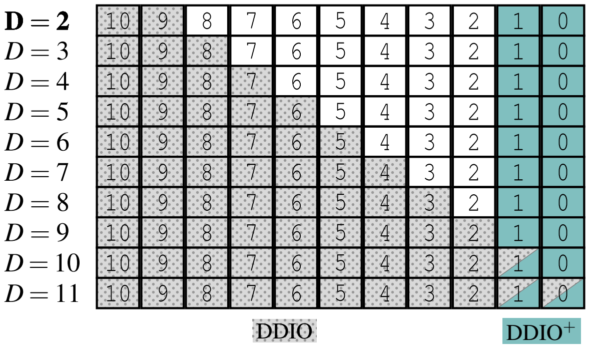


Figure 7: Model of the LLC set (Xeon Platinum Silver 4208)

Finding #1 (rev.): Spatially limited LLC interaction.
Another portion of the LLC set is malleable by secondary devices. This region (DDIO$^+$) has associativity two.

6 Revisiting Cache Eviction

With magnet ways (**#3**), this section challenges the concept of *minimal* eviction sets through efficient eviction with fewer elements than the cache associativity (**#1**). First, as a stepping stone of independent interest, we evict shared lines from a victim's private caches without directory contention. For simplicity, we consider the default DDIO configuration ($D = 2$).

6.1 Eviction without Coherence Directory

Algorithm 3 evicts a shared line from remote victim caches with the shared access method. First, the attacker prepares the target in L2, and waits. If the victim accesses the target, it resides in attacker and victim L2, *and* the LLC (cf. Figure 4d). Second, the attacker evicts it from the LLC which, if it was indeed there, invalidates the copies in all L1/L2 caches. Otherwise, the target remains in the attacker's L2. This invalidation is not strictly required for non-inclusive LLCs, but happens in practice [67]. LLC eviction (line 3) is implemented by accessing W lines with threads on different cores (Figure 4d).

Algorithm 3 Eviction without Coherence Directory (CD)

On the right, the LLC and L2 states of victim ($L2_V$) and attacker ($L2_A$) are shown for each operation on the target address (T).

```
                          $L2_V$ $L2_A$  LLC        $L2_V$ $L2_A$  LLC
1: att_CpuRead            [   ] [  T] [   ]      [   ] [  T] [   ]
2: vic_CpuRead ?             ✓ (access)             ✗ (no access)
                          [  T] [  T] [  T]      [   ] [  T] [   ]
3: att_CpuEvict_LLC       [   ] [   ] [   ]      [   ] [  T] [   ]
4: att_CpuTime                 RAM                     L2
```

Algorithm 3 can also be inverted, making it slightly more complex (details in Appendix D). Surprisingly, the inverted version can evict lines from L2 caches in remote sockets.

6.2 Reduced Eviction

Combined attackers can produce magnet ways, i.e., empty ways to attract incoming lines (cf. Section 3.2.3). Assuming a magnet way in the DDIO region, a combined attacker can evict a target line from a victim cache by, first, triggering its LLC allocation (where the magnet will attract it) and second, evicting it from the DDIO region with only two congruent addresses. Without magnet ways, the target may be installed anywhere in the LLC set, and W addresses are needed to reliably evict it. Algorithm 4 shows how to produce and exploit DDIO magnet ways using a tiny eviction set of four addresses (0-3). Strictly speaking, two congruent lines suffice. However, to make eviction repeatable, we suffer from DDIO$^+$ behavior and rely on our model LLC structure to overcome it (cf. Section 5).

First, the secondary device writes lines 0-1, followed by CPU writes. Afterwards, the secondary device writes them again, while the CPU reads the target line. This terminates the preparation phase, with the target in the attacker's L2, two magnet ways in the DDIO region, and lines 0-1 in DDIO$^+$.

Algorithm 4 Eviction with Reduced Eviction Set

On the right, the attacker's L2 and LLC (with DDIO$^+$ and DDIO) are shown, with lines 0-3, LLC magnet ways (M), and target (T) .

```
Prepare:                          L2       LLC
  1: att_SecWr (0,1)           [     ] [    |    1 0]
  2: att_CpuWr (0,1)           [  1 0] [    |    M M]
  3: att_SecWr (0,1)           [     ] [1 0 |    M M]
  4: att_CpuRd (T)             [    T] [1 0 |    M M]
Wait:
  5: ? vic_CpuRd (T)          ✓ (access)                ✗ (no access)
                       [     ] [1 0 |    M T]    [    T] [1 0 |    M M]
Measure & Reinstate:
  6: att_SecWr (2,3)   [     ] [1 0 |    3 2]    [    T] [1 0 |    3 2]
  7: att_CpuTime (T)           RAM                       L2
  8: att_CpuWr (2,3)   [3 2 T] [1 0 |    M M]    [3 2 T] [1 0 |    M M]
  9: att_SecWr (2,3)   [    T] [3 2 |    M M]    [    T] [3 2 |    M M]
Reiterate from line 5, swapping the roles of 0-1 and 2-3.
```

Then, the attacker waits. If the victim accesses TARGET, it moves to the LLC DDIO region, attracted by the magnet ways. Afterwards, secondary writes to 2-3 evict the DDIO region (0-1 cannot serve this purpose, as they are modified and allocate to DDIO$^+$ instead). A CPU timing measurement of TARGET reveals it to reside either in L2 (no victim access) or in RAM (victim access). Finally, 2-3 are written by the CPU and the secondary device, recreating the magnet ways in the DDIO region and placing 2-3 in DDIO$^+$ instead of 0-1. The next iteration can now start, and 0-1 swap roles with 2-3.

The reduced eviction can also be inverted, making it work across CPU sockets on our platforms (cf. Appendix D).

7 Implementation

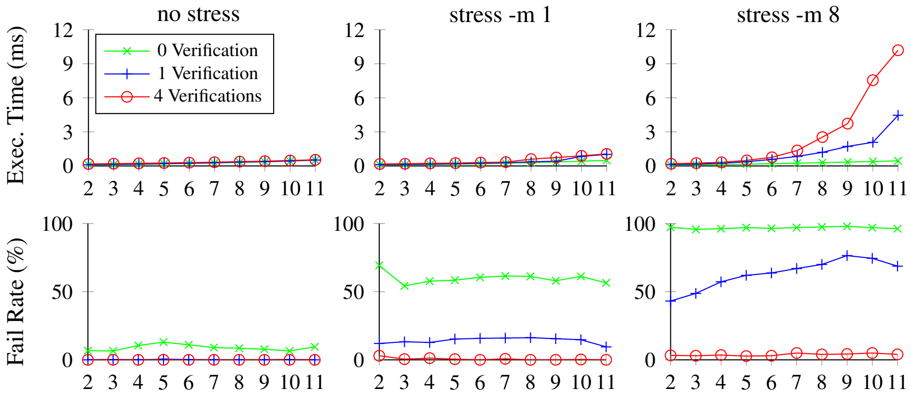
We work with the FPGA-accelerated cloud platforms of Section 2.6 and implement a hardware (HW) module to demonstrate our findings. This section explains its functionality.

Read-Write Primitives. As we work with Intel FPGAs, we use Open Programmable Acceleration Engine (OPAE) to integrate FPGA acceleration into software applications. OPAE divides the FPGA's programmable fabric into two parts; a blue-bitstream pre-programmed by Intel, and a green-bitstream that implements the user's hardware accelerators. The blue-bitstream acts as a bridge between accelerators and software, and provides them with Direct Memory Access (DMA).

DMA and a timing source are essential components for cache-timing experiments. With OPAE, a DMA operation consists of two transactions; one for asserting memory read and write requests, and another to monitor the completion of this request. To measure latency, we create a counter-based timing source on the FPGA, similar to Weissman et al. [62].

Stress Level	# of Verf.	Exec. Time (ms)	Fail Rate (%)
No	0	0.10	6.72
	1	0.12	0.00
	4	0.17	0.00
-m 1	0	0.09	69.27
	1	0.12	11.99
	4	0.17	3.00
-m 8	0	0.08	97.25
	1	0.13	43.15
	4	0.19	3.24

(a) Detailed evaluation for $D = 2$ (default)



(b) The effect of D (on X axis) on the performance and accuracy.

Figure 8: Performance of HW-accelerated eviction set construction on our Xeon Silver setup ($\text{EV} = W$, avg. of 1000 runs).

It counts the cycles that expire between requests and replies (at 400 MHz), allowing to distinguish the memory level that serves the request (L2, LLC or RAM), as given in Appendix A. The FPGA counter is not synchronized to the CPU counter.

The hardware design also features a set of software-configurable registers to instruct the actions of the accelerator, e.g., to perform a timed read or write at a given address. More sophisticated instructions are described next.

Fast Eviction Set Finding. We extend the hardware module with an advanced state machine that implements the fast eviction-set finding algorithm introduced in Algorithm 1.

The algorithm is fully encapsulated in hardware and proceeds without additional interaction. Moreover, a few settings are exposed to users. Essential settings are the target for which to find the eviction set, the desired size, the number of verification repetitions, and the timing threshold (LLC vs. RAM accesses). The delay between consecutive memory operations can also be configured to ensure their in-order execution.

Access Sequences. The hardware module supports encapsulating commonly used sequences of read and write operations to reduce HW-SW interaction overhead, e.g., for fast reduced eviction (cf. Section 6.2). Again, the configuration of these sequences happens with software-accessible registers.

8 Evaluation and Discussion

This section evaluates three applications of our accelerator, demonstrating: (1) the speed and accuracy of eviction set construction, (2) a covert channel encoding information in the number of evicted lines, and (3) reduced eviction in practice.

8.1 HW-Accelerated Eviction Set Finding

8.1.1 Performance

Figure 8 presents the performance and accuracy of hardware-accelerated eviction set construction, measured on our local Xeon Silver platform (8 slices, cf. Section 2.6). We consider sets of size $\text{EV} = W = 11$, and conduct the measurements on both idle and noisy systems. For the latter, we use `stress` to emulate moderate and high noise, resp. with `-m 1` and `8`.

We report *end-to-end* execution time, i.e., we do not exclude any preparation steps or interactions with hardware. The existence of even a single false positive in the set classifies it as *failed*, even if the remainder of the set is correct. The candidate verification (cf. Algorithm 2) significantly reduces such false positives at the cost of slightly increased runtime.

Hardware-accelerated eviction set construction is very effective. On our local platform, the accelerator can find an LLC/CD eviction set in around 120 µs on idle systems, or 200 µs under stress. For the default DDIO configuration, the throughput of one accelerator (in EVS/s, eviction sets per second) reaches more than 8000 EVS/s for noise-free systems, and around 5000 EVS/s for very high noise. Such speeds allow to map out the entire 11 MB LLC in ≈ 2 s. For reduced eviction sets ($\text{EV} = 4$), the average throughput is around 16 kEVS/s. On the ACE platforms, the accelerator is roughly 3 times slower, since the slice count increases from 8 to 28.

Influence of *D*. For non-default DDIO configurations (i.e., $D > 2$), the performance of eviction set construction decreases with D. As D grows, the associativity perceived by the accelerator increases (**#1**), which increases the relative weight of the `Discover` phase. Especially for high noise pressure, this impacts the performance for two main reasons (cf. Section 4.2.3). First, during discovery, more guesses are required to detect a congruence, as detection may only occur for every $D-1$ congruent guesses. Second, false positives have adverse effects, as all DISCOVER_FOUND are discarded upon a failed verification test, in contrast to just a single discarded address during `Expand`. All in all, our eviction set construction is still accurate and faster than related work. Moreover, $D = 2$ is the default configuration. To our knowledge, no benchmarking tools exist to help users decide when to change it.

8.1.2 Comparison

Expansion-based Methods. As covered in Section 4.2, expansion-based methods work well when the congruence test is non-disturbing. To our knowledge, the only other expansion-based method[2] is our work on Prime+Scope [47], which obtains a repeatable congruence test by exploiting the properties of lines that are cached in multiple levels simultaneously.

Although the algorithm based on Prime+Scope bears a similar structure to Algorithm 1, it differs in several important aspects. First, it is concerned with a different threat model. Second, the non-destructive measurement relies on fundamentally different properties. Third, the Prime+Scope version for non-inclusive Intel LLCs needs to orchestrate two attacker threads on different cores to allocate in the LLC (cf. Figure 4d). Finally, CPU-based attackers perceive the full LLC associativity, in contrast to the accelerator (e.g., $D=2$).

Reduction-based Methods. For reference, we also compare the accelerator to reduction-based methods. Yan et al. [66] proposed the state-of-the-art reduction algorithm for non-inclusive caches. As their code is not available, we are unable to fairly compare to them. Moreover, their work may be amenable to similar optimizations as those shown for inclusive caches [60]. To have a meaningful data point for comparison, we compare with the highly optimized implementations for *inclusive caches* by Vila et al. [60]. Arguably, this serves as an upper bound for performance in non-inclusive caches, because of the obstacles identified in prior work [66]. On the other hand, our accelerator is agnostic to LLC inclusion, so we expect it to apply to inclusive LLCs with similar performance.

Results. Like the accelerator, we measure the performance of the Prime+Scope [47] code on the local Xeon Silver platform. For the Vila et al. [60] code, we match their CPU and configuration. Importantly, the number of slices is the same for both platforms under consideration (cf. Section 4.2.3).

As Table 6 shows, our HW accelerator achieves good end-to-end performance. Compared to the algorithm based on Prime+Scope (P+S), it is another order of magnitude faster. Depending on the noise level, it is between two and three orders of magnitude faster than reduction-based methods, at the cost of a small decrease in accuracy.

Note, however, that the threat model for the hardware accelerator is different. It assumes a secondary device but no code execution, whereas CPU-only methods assume code execution (native or otherwise [40, 60]) but no secondary device.

8.1.3 Robustness, Simplicity and Stealth

Robustness. The accelerator maintains good performance for high noise (cf. Figure 8), with several contributing factors. The timing source is a noise-free HW counter, and interference with other processes is limited to the DDIO region. The algorithm itself is also robust. False-negative errors only increase the execution time, and most false-positive errors are detected by Algorithm 2. Provided that the `Discover` phase is successful, remaining false positives do not affect the other addresses in the `Expand` phase, as every address is individually tested for congruence. In contrast, reduction algorithms may have to implement backtracking [60] to avoid getting stuck, as false positives trigger the removal of congruent addresses.

Table 6: End-to-end performance (1000 runs) of our accelerator compared to Prime+Scope [47] on our local setup, and optimized reduction-based algorithms [60].

Impl.	CPU & Cache	Stress Level	Error Rate (%)	Exec. Time (msec)
Ours*	Skylake-SP Xeon Slv. 4208 11-way LLC Non-Inclusive	no	0.0	0.17
		-m 1	0.5	0.17
		-m 8	3.5	0.17
P+S [47]		no	0.0	1.11
		-m 1	0.0	1.24
		-m 8	0.3	3.56
[60]†	Skylake Core i5-6500 12-way LLC Inclusive	no	0.0	19
		-m 1	0.0	35
		-m 3	0.0	206

∗ for the default DDIO configuration with $D=2$.

† with initial set size 120, while other works do not use an initial set.

Simplicity. Fully described by a few lines of pseudocode, the accelerator is simple to understand. It does not use hierarchy-specific techniques, e.g., directory contention or helper eviction sets [66]. The accelerator interacts with a D-way set-associative view of the LLC which, at least for small D, is a great simplification. Furthermore, it is oblivious to associativity, replacement policy and noise in low-level caches. The same FPGA bitstream is used on all our test platforms.

Stealth. Due to its speed, eviction set construction is hard to detect at runtime. Moreover, the timing source is implemented on the FPGA fabric, so accesses to it are invisible to the CPU or blue-bitstream. Finally, it is essentially a state machine, requesting read and write operations to the blue-bitstream and timing them. Its resource utilization is very low, making it a hard-to-notice attachment. Appendix F covers detailed utilization numbers for the accelerator, showing that it barely increases compared to Intel's Hello World baseline.

8.2 Amplitude-Based Covert Channel

We implement a covert channel between combined attackers, demonstrating the precise control over the cache hierarchy with only a few congruent addresses. It transfers information by manipulating DDIO ($D=2$) and DDIO$^+$ ways simultaneously and independently. Moreover, due to non-perturbing reads (**#2**), it reliably encodes amplitude information in the

[2] One exception is Prime+Prune+Probe [46], which applies to a new class of randomized protected caches that are not in use today.

Table 7: Covert channel with prime (PR), transmit (TX) and probe (PB). Symbols are always transferred over DDIO, and optionally over DDIO$^+$ ($\underline{\text{optional operations underlined}}$). Parties use their own eviction sets, resp. `A,B,`$\cdots$ and `1,2,`$\cdots$

	Receiver	Sender
PR	$\underline{\texttt{CpuWr(C,D)}}$ $\underline{\texttt{SecWr(C,D)}}$ `SecWr(A,B)`	`CpuWr(3,4)`
TX		$\underline{\texttt{SecWr(3/4)}}$ `SecWr(1/2)`
PB	$\underline{\texttt{SecTime(C,D)}}$ `SecTime(A,B)`	

PR: Receiver puts `A,B` into DDIO, and `C,D` into $\underline{\text{DDIO}^+}$.

TX: Sender may replace `A,B,C,D` with `1,2,3,4`.

PB: Receiver checks if `A,B` are still in DDIO, and `C,D` in $\underline{\text{DDIO}^+}$.

signal (i.e., the *number* of evicted ways), which performs poorly for traditional attackers due to self-eviction [37]. It does not use shared memory; the parties use their own eviction sets (agreed upon in advance, e.g., using HW-acceleration).

Table 7 shows two versions; one over the DDIO region, and one that combines DDIO/DDIO$^+$ regions. Both consist of three stages. First, the receiver primes (PR) the DDIO lines (if applicable also in DDIO$^+$). Second, the sender transmits (TX) a symbol by overwriting zero, one or two of the receiver's DDIO (and DDIO$^+$) lines. Third, the receiver probes (PB) its lines to determine how many have been evicted.

The DDIO channel encodes a ternary symbol, i.e., $\log_2 3 = 1.58$-bits per cache set. For transferring 512 packets of 256 symbols, we achieve 264 Kbps bandwidth (BW) with 2.26 % Symbol Error Rate (SER). The DDIO/DDIO$^+$ channel encodes two ternary symbols, i.e., $\log_2(3 \cdot 3) = 3.17$ bits per cache set. In this case, the BW is slightly lower at 211 Kbps with 2.20 %SER, because of extra interfacing with hardware.

Comparison. We use a shared timestamp counter to synchronize transmitter and receiver, as well as a known preamble. Our implementation is open to optimizations, e.g., common engineering practices like synchronization or error correction, or transmission over multiple sets.

Although the covert channel only serves to illustrate the fine-grained spatial capabilities of combined attackers, we briefly compare it with closely related implementations. Weissman et al. [62] build a covert channel from FPGA to CPU. It achieves 95 Kbps and, like ours, can be improved. Yan et al. [66] establish a cross-core channel using CD contention on a non-inclusive LLC, and achieve 0.2 Mbps. To our knowledge, the fastest cross-core covert channels with unshared memory achieve 2–4 Mbps [42, 45, 47]. Ours is an order of magnitude slower, mostly because of hardware interfacing overhead, but is open to improvements.

Table 8: Eviction patterns for shared lines, their eviction set size, eviction rate, number of accesses and attacker model

Pattern	EV	Ev. Rate	Accesses	$\mathcal{V}_{\text{CC}}$	$\mathcal{V}_{\text{CS}}$	$\mathcal{A}$
CD-11-9 [66]	11	$\approx$ 25%	216	✓		$\mathcal{A}_{\text{STD}}$
CD-13-9 [66]	13	$\approx$ 95%	234	✓		$\mathcal{A}_{\text{STD}}$
CD-14-9 [66]	14	$\approx$100%	252	✓		$\mathcal{A}_{\text{STD}}$
L2-16-9/LLC-11 [66]	16+11	$\approx$100%	144+22	✓		$\mathcal{A}_{\text{STD}}$
No CD (Alg. 3)	11	$\approx$100%	22	✓	✓	$\mathcal{A}_{\text{STD}}$
Reduced (Alg. 4)	4	$\approx$100%	6	✓	✓	$\mathcal{A}_{\text{CMB}}$

8.3 Reduced Eviction

Eviction Rate. Table 8 compares the eviction rates of our new patterns with those reported by Yan et al. [66] for shared lines. For all patterns, the goal is to evict a shared line, currently in another core's L2, from the hierarchy. We achieve a near-perfect eviction rate, with fewer accesses and the bonus of working across sockets (cf. Appendix D). Our CD-less eviction (Algorithm 3) implements LLC eviction with two threads each writing EV = 11 addresses once, generating direct LLC contention. The reduced eviction implements Algorithm 4.

Yan et al. observe that using unshared lines to evict shared lines from remote L2 caches through the CD has unsatisfactory results. To overcome this, they use shared lines, instantiating two threads that repeatedly access a CD eviction set; e.g., two threads iterating 9 times over a set of size EV = 13 (CD-13-9 in Table 8) yield an eviction rate of $\approx$ 95% and 234 accesses. They also propose a pattern with contention on L2 (EV = 16) and CD (EV = 11) simultaneously, and attribute its success to bypassing the CD replacement policy. Our work suggests that the underlying mechanism is actually an instance of Algorithm 3; replacing shared access (Figure 4d) with L2 contention (Figure 4a) to transfer the line to LLC. Hence, what appeared to be CD contention is actually LLC contention.

End-to-end Example: AES. We demonstrate the feasibility of reduced and cross-socket eviction with the OpenSSL 1.0.1e AES T-Tables implementation, a now-standard target for side-channel research. We do not claim algorithmic improvements and simply refer to the illustrative synchronous first-round attack [41, 56] to show the feasibility of our techniques (cf. Appendix E for attack details). In short, the attacker evicts the cache lines containing the T-Tables, triggers encryptions with known plaintext bytes, and monitors access patterns to the tables. Through statistical differences between table accesses, the attacker learns the upper half of every secret key byte.

We use the hardware accelerator to construct eviction sets and assume the victim binary to reside in small read-only pages. To showcase reduced eviction, we early-abort the accelerator for EV = 4. To overcome the writing limitation, we construct the eviction sets indirectly for addresses with the same small page offset, and then test whether they contend with the tables in the LLC/CD. This test can work with several mechanisms (cf. Figure 4). We select Algorithm 4 for its speed and reliability. To construct all necessary sets on the Xeon Platinum 8180 (ACE1, 28 slices), we observe a median runtime of 194 ms and perfect accuracy over 100 runs.

Limitations. For reduced eviction, if both magnet ways are occupied before TARGET is installed, e.g., due to noise, the target ends up outside of the DDIO region. Though this rarely happens on our setup, normal behavior can be reinstated by evicting the full set once. For cross-socket reduced eviction, the target can become stuck in the victim socket if the victim reads it while still in the attacker socket's LLC.

Results. In the absence of noise, reduced eviction consistently reveals the subkeys within 300 traces, both in same- and cross-socket attacks (cf. Appendix E). However, robustness in the latter case is significantly lower, and we recommend using an eviction set with full associativity (cf. Algorithm 5).

9 Related Work

9.1 Cache-based Side Channel Attacks

Table 9: Non-inclusive Cache Attacks (Shared Memory)

Contribution	Flushless	Cross-Socket	Single-Thread	Reduced Eviction
Lipp [30]	✓	✗	✓	✗
Irazoqui [22]	✗	✓	✓	✗
Yan (F+R) [66]	✗	✗	✓	✗
Yan (E+R) [66]	✓	✗	✗	✗
Ours ($\mathcal{A}_{\texttt{STD}}$)	✓	✓	✓	✗
Ours ($\mathcal{A}_{\texttt{CMB}}$)	✓	✓	✗	✓

Non-inclusive Cache Attacks. Lipp et al. [30] mount Flush+Reload and Evict+Reload on small non-inclusive ARM caches. Irazoqui et al. [22] illustrate that Flush+Reload applies to all caches in the same coherence domain, even across sockets. These attacks bypass non-inclusive LLCs by relying on self-eviction [30], or using `clflush` [22]. Yan et al. [66], in contrast, propose a multi-threaded Evict+Reload to reliably evict the shared target (cf. Section 8.3). We show an Evict+Reload without CD manipulation ($\mathcal{A}_{\text{STD}}$), and one with four addresses ($\mathcal{A}_{\text{CMB}}$), refuting that eviction sets must cover the full cache associativity. Table 9 positions our work within this subset of related work.

In the absence of shared memory, the attacker can mount Prime+Probe [66] or Prime+Scope [47] on the coherence directory, leveraging our eviction set construction.

Cross-CPU. Yao et al. [68, 69] present a flushless cross-socket covert channel based on non-uniform memory access and cache coherence. They rely on a cooperating transmitter to evict the target (i.e., covert channel). Our cross-socket channel does not have this requirement, showing that shared memory is a security risk even when `clflush` is disabled and the victim is the only tenant on a CPU socket. A noteworthy non-cache cross-socket side-channel attack is Drama [45].

Secondary devices. Frigo et al. [11] accelerate microarchitectural attacks with the GPU, which is also connected to the cache hierarchy (though differently than DDIO devices). Their focus is Rowhammer-based fault injection [24, 39, 53, 58].

Weissman et al. [62] instantiate the secondary device as an FPGA. They leverage non-destructive reads (**#2**) to accelerate Rowhammer, and study cache attacks from CPU to FPGA and vice-versa. However, the statically constrained DDIO region provides challenges for FPGA-based attacks.

Kurth et al. [27] also construct eviction sets with a secondary device, i.e., a NIC. Their network-based threat model faces more challenges: it takes about five minutes to produce 64 eviction sets over the network. However, since properties **#1** and **#2** hold, our eviction set algorithm may accelerate it.

Taram et al. [55] describe a CPU process that infers network memory access patterns by the NIC (based on DDIO).

9.2 Countermeasures

Constant-time programming successfully thwarts the combined attacker explored in this paper, as it removes vulnerable code patterns. It is now common practice to harden cryptographic implementations, and several techniques have been proposed, e.g., [1, 8, 28, 49, 64]. However, access patterns can reveal other secrets, such as user input [13, 50], browsing behavior [40, 54], or model parameters [65]. Additionally, capturing all side-channel leaks remains difficult in practice [51].

Hardware-based countermeasures have attracted attention in recent years, and are generally based on, e.g., partitioning the cache [7, 32], randomizing the address-to-index mapping [48, 61, 63], or approximating fully associative caches [6, 52]. For non-inclusive cache hierarchies, SecDir [67] hardens the coherence directory explicitly. However, existing hardware-based proposals non-trivially interact with DDIO/DDIO^{+} regions. Additionally, such countermeasures must make explicit all potential transfers between cache levels, as undocumented transfers (cf. Section 6.1) might endanger their security.

Runtime detection using on-die counters [4, 5, 71] could be generalized to combined attackers. It should be investigated whether they sufficiently capture accelerator activity. For FPGAs, they can be embedded in the blue bitstream.

Some works propose limiting access to high-resolution timers [35, 59]. Such countermeasures do not generally thwart combined attacks, as they can bring their own timing source.

Invalidating the findings of Section 3.2 counteracts the results in this work. The accelerator would suffer if writes occupy the full set (**#1**), or reads alter LLC state (**#2**). However, the performance implications are significant, and increasing DDIO access to the cache improves attacks from accelerators alone [27, 62]. The precise manipulation of the cache hierarchy (**#3**) seems to be fundamental to DDIO and is non-trivial to disable. An exception is the unexpected cross-socket transfer (cf. Section 6.1). This transaction is not essential to maintain coherence. We believe it to be a performance heuristic.

10 Conclusion

Heterogeneous multi-tenancy is a dangerous trend, providing attackers with ever-more expressive primitives to manipulate shared microarchitectural state. This work exposed undocumented behavior in non-inclusive Intel caches and DDIO. Leveraging these insights, we developed a proof-of-concept FPGA hardware accelerator to shatter speed records for eviction set construction, build covert channels with multi-bit symbols, and evict lines from the cache with tiny sets.

Acknowledgments

We thank the anonymous USENIX Security reviewers for their insightful feedback. This research is partially funded by the European Research Council (ERC #695305) and the Flemish Government (FWO project TRAPS). It is also supported by CyberSecurity Research Flanders (#VR20192203) and a generous gift from Intel. Antoon Purnal is supported by a grant of the Research Foundation - Flanders (FWO).

References

[1] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. Verifying Constant-time Implementations. In *USENIX Security Symposium*, 2016.

[2] Andrea Arcangeli, Izik Eidus, and Chris Wright. Increasing Memory Density by using KSM. In *Proceedings of the Linux Symposium*, 2009.

[3] Daniel J Bernstein. Cache-timing attacks on AES, 2005.

[4] Samira Briongos, Gorka Irazoqui, Pedro Malagón, and Thomas Eisenbarth. Cacheshield: Detecting Cache Attacks Through Self-observation. In *ACM Conference on Data and Application Security and Privacy (CODASPY)*, 2018.

[5] John Demme, Matthew Maycock, Jared Schmitz, Adrian Tang, Adam Waksman, Simha Sethumadhavan, and Salvatore Stolfo. On the Feasibility of Online Malware Detection with Performance Counters. *ACM SIGARCH Computer Architecture News*, 2013.

[6] Ghada Dessouky, Tommaso Frassetto, and Ahmad-Reza Sadeghi. HybCache: Hybrid Side-Channel-Resilient Caches for Trusted Execution Environments. In *USENIX Security Symposium*, 2020.

[7] Leonid Domnitser, Aamer Jaleel, Jason Loew, Nael Abu-Ghazaleh, and Dmitry Ponomarev. Non-Monopolizable Caches: Low-Complexity Mitigation of Cache Side Channel Attacks. *ACM Transactions on Architecture and Code Optimization (TACO)*, 2012.

[8] Goran Doychev, Boris Köpf, Laurent Mauborgne, and Jan Reineke. CacheAudit: A Tool for the Static Analysis of Cache Side Channels. In *USENIX Security Symposium*, 2013.

[9] Alireza Farshin, Amir Roozbeh, Gerald Q Maguire Jr, and Dejan Kostić. Make the Most Out of Last Level Cache in Intel Processors. In *EuroSys Conference*, 2019.

[10] Alireza Farshin, Amir Roozbeh, Gerald Q Maguire Jr, and Dejan Kostić. Reexamining Direct Cache Access to Optimize I/O Intensive Applications for Multi-hundred-gigabit Networks. In *USENIX Annual Technical Conference (ATC)*, 2020.

[11] Pietro Frigo, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Grand Pwning Unit: Accelerating microarchitectural attacks with the GPU. In *IEEE Symposium on Security and Privacy (S&P)*, 2018.

[12] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016.

[13] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache Template Attacks: Automating Attacks on Inclusive Last-level Caches. In *USENIX Security Symposium*, 2015.

[14] David Gullasch, Endre Bangerter, and Stephan Krenn. Cache Games–Bringing Access-based Cache Attacks on AES to Practice. In *IEEE Symposium on Security and Privacy (S&P)*, 2011.

[15] Ram Huggahalli, Ravi R. Iyer, and Scott Tetrick. Direct Cache Access for High Bandwidth Network I/O. In *32st International Symposium on Computer Architecture (ISCA)*, 2005. doi:10.1109/ISCA.2005.23.

[16] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical Timing Side Channel Attacks against Kernel Space ASLR. In *IEEE Symposium on Security and Privacy (S&P)*, 2013.

[17] Mehmet Sinan Inci, Berk Gulmezoglu, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Cache Attacks Enable Bulk Key Recovery on the Cloud. In *Cryptographic Hardware and Embedded Systems (CHES)*, 2016.

[18] Intel. Intel Data Direct I/O Technology Overview. https://www.intel.co.jp/content/dam/www/public/us/en/documents/white-papers/data-direct-i-o-technology-overview-paper.pdf, 2012.

[19] Intel. Intel CAT: Improving Real-Time Performance by Utilizing Cache Allocation Technology. https://software.intel.com/content/www/us/en/develop/articles/introduction-to-cache-allocation-technology.html, 2015.

[20] Intel. Open Programmable Acceleration Engine: Libraries. https://github.com/OPAE/opae-libs/blob/master/include/opae/buffer.h#L28, 2021.

[21] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. S$A: A Shared Cache Attack That Works Across Cores and Defies VM Sandboxing – and Its Application to AES. In *IEEE Symposium on Security and Privacy (S&P)*, 2015.

[22] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Cross Processor Cache Attacks. In *ACM SIGSAC Asia Conference on Computer and Communications Security (AsiaCCS)*, 2016.

[23] Yeongjin Jang, Sangho Lee, and Taesoo Kim. Breaking Kernel Address Space Layout Randomization with Intel TSX. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016.

[24] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors. *ACM SIGARCH Computer Architecture News*, 2014.

[25] 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. Spectre Attacks: Exploiting Speculative Execution. In *IEEE Symposium on Security and Privacy (S&P)*, 2019.

[26] Paul C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Advances in Cryptology - CRYPTO*, 1996.

[27] Michael Kurth, Ben Gras, Dennis Andriesse, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. NetCAT: Practical Cache Attacks From the Network. In *IEEE Symposium on Security and Privacy (S&P)*, 2020.

[28] Adam Langley. ctgrind—checking that functions are constant time with Valgrind, 2010. *URL https://github.com/agl/ctgrind*, 2010.

[29] Sheng Li, Hyeontaek Lim, Victor W Lee, Jung Ho Ahn, Anuj Kalia, Michael Kaminsky, David G Andersen, O Seongil, Sukhan Lee, and Pradeep Dubey. Architecting to Achieve a Billion Requests per Second Throughput on a Single Key-value Store Server Platform. In *International Symposium on Computer Architecture (ISCA)*, 2015.

[30] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. ARMageddon: Cache Attacks on Mobile Devices. In *USENIX Security Symposium*, 2016.

[31] 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. Meltdown: Reading Kernel Memory from User Space. In *USENIX Security Symposium*, 2018.

[32] Fangfei Liu, Qian Ge, Yuval Yarom, Frank Mckeen, Carlos Rozas, Gernot Heiser, and Ruby B Lee. Catalyst: Defeating Last-level Cache Side Channel Attacks in Cloud Computing. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016.

[33] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-Level Cache Side-Channel Attacks Are Practical. In *IEEE Symposium on Security and Privacy (S&P)*, 2015.

[34] Ilias Marinos, Robert NM Watson, and Mark Handley. Network Stack Specialization For Performance. *ACM SIGCOMM Computer Communication Review*, 2014.

[35] Robert Martin, John Demme, and Simha Sethumadhavan. Timewarp: Rethinking Timekeeping and Performance Monitoring Mechanisms to Mitigate Side-channel Attacks. In *International Symposium on Computer Architecture (ISCA)*, 2012.

[36] Clémentine Maurice, Nicolas Le Scouarnec, Christoph Neumann, Olivier Heen, and Aurélien Francillon. Reverse Engineering Intel Last-Level Cache Complex Addressing Using Performance Counters. In *Research in Attacks, Intrusions, and Defenses (RAID)*, 2015.

[37] Clémentine Maurice, Manuel Weber, Michael Schwarz, Lukas Giner, Daniel Gruss, Carlo Alberto Boano, Stefan Mangard, and Kay Römer. Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud. In *Network and Distributed System Security Symposium (NDSS)*, 2017.

[38] David Mulnix. Intel® Xeon® Processor Scalable Family Technical Overview. http://web.archive.org/web/20080207010024/http://www.808multimedia.com/winnt/kernel.htm, 2017. Accessed: 2020-08-13.

[39] Onur Mutlu. The RowHammer Problem and Other Issues we may Face as Memory Becomes Denser. In *Design, Automation & Test in Europe (DATE)*, 2017.

[40] Yossef Oren, Vasileios P. Kemerlis, Simha Sethumadhavan, and Angelos D. Keromytis. The Spy in the Sandbox: Practical Cache Attacks in JavaScript and Their Implications. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2015.

[41] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache Attacks and Countermeasures: The Case of AES. In *Cryptographers' Track at the RSA Conference on Topics in Cryptology (CT-RSA)*, 2006.

[42] Riccardo Paccagnella, Licheng Luo, and Christopher W. Fletcher. Lord of the Ring(s): Side Channel Attacks on the CPU On-Chip Ring Interconnect Are Practical. In *USENIX Security Symposium*, 2021.

[43] Dan Page. Theoretical Use of Cache Memory as a Cryptanalytic Side-Channel. *IACR Cryptol. ePrint Arch. 2002/169*, 2002.

[44] Colin Percival. Cache Missing for Fun and Profit. In *BSDCan*, 2005.

[45] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. DRAMA: Exploiting DRAM Addressing for Cross-cpu Attacks. In *USENIX Security Symposium*, 2016.

[46] Antoon Purnal, Lukas Giner, Daniel Gruss, and Ingrid Verbauwhede. Systematic Analysis of Randomization-based Protected Cache Architectures. In *IEEE Symposium on Security and Privacy (S&P)*, 2021.

[47] Antoon Purnal, Furkan Turan, and Ingrid Verbauwhede. Prime+Scope: Overcoming the Observer Effect for High-Precision Cache Contention Attacks. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2021.

[48] Moinuddin K. Qureshi. CEASER: Mitigating Conflict-based Cache Attacks via Encrypted-address and Remapping. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018.

[49] Oscar Reparaz, Josep Balasch, and Ingrid Verbauwhede. Dude, is my code constant time? In *Design, Automation & Test in Europe (DATE)*, 2017.

[50] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, You, Get off of My Cloud: Exploring Information Leakage in Third-party Compute Clouds. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2009.

[51] Eyal Ronen, Robert Gillham, Daniel Genkin, Adi Shamir, David Wong, and Yuval Yarom. The 9 Lives of Bleichenbacher's CAT: New Cache ATtacks on TLS Implementations. In *IEEE Symposium on Security and Privacy (S&P)*, 2019.

[52] Gururaj Saileshwar and Moinuddin Qureshi. MIRAGE: Mitigating Conflict-Based Cache Attacks with a Practical Fully-Associative Design. In *USENIX Security Symposium*, 2021.

[53] Mark Seaborn and Thomas Dullien. Exploiting the DRAM Rowhammer Bug to Gain Kernel Privileges. *Black Hat*, 2015.

[54] Anatoly Shusterman, Lachlan Kang, Yarden Haskal, Yosef Meltser, Prateek Mittal, Yossi Oren, and Yuval Yarom. Robust Website Fingerprinting Through the Cache Occupancy Channel. In *USENIX Security Symposium*, 2019.

[55] Mohammadkazem Taram, Ashish Venkat, and Dean Tullsen. Packet Chasing: Spying on Network Packets over a Cache Side-channel. In *International Symposium on Computer Architecture (ISCA)*, 2020.

[56] Eran Tromer, Dag Arne Osvik, and Adi Shamir. Efficient Cache Attacks on AES, and Countermeasures. *Journal of Cryptology*, 2010.

[57] Furkan Turan and Ingrid Verbauwhede. Trust in FPGA-Accelerated Cloud Computing. *ACM Computing Surveys*, 2020.

[58] Victor Van Der Veen, Yanick Fratantonio, Martina Lindorfer, Daniel Gruss, Clémentine Maurice, Giovanni Vigna, Herbert Bos, Kaveh Razavi, and Cristiano Giuffrida. Drammer: Deterministic Rowhammer attacks on mobile platforms. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016.

[59] Bhanu C Vattikonda, Sambit Das, and Hovav Shacham. Eliminating Fine Grained Timers in Xen. In *ACM Workshop on Cloud Computing Security (CCSW)*, 2011.

[60] Pepe Vila, Boris Köpf, and José F. Morales. Theory and Practice of Finding Eviction Sets. In *IEEE Symposium on Security and Privacy (S&P)*, 2019.

[61] Zhenghong Wang and Ruby B. Lee. New Cache Designs for Thwarting Software Cache-based Side Channel Attacks. In *International Symposium on Computer Architecture (ISCA)*, 2007.

[62] Zane Weissman, Thore Tiemann, Daniel Moghimi, Evan Custodio, Thomas Eisenbarth, and Berk Sunar. JackHammer: Efficient Rowhammer on Heterogeneous FPGA-CPU Platforms. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2020.

[63] Mario Werner, Thomas Unterluggauer, Lukas Giner, Michael Schwarz, Daniel Gruss, and Stefan Mangard. SCATTERCACHE: Thwarting Cache Attacks via Cache Set Randomization. In *USENIX Security Symposium*, 2019.

[64] Jan Wichelmann, Ahmad Moghimi, Thomas Eisenbarth, and Berk Sunar. MicroWalk: A Framework for Finding Side Channels in Binaries. In *Annual Computer Security Applications Conference (ACSAC)*, 2018.

[65] Mengjia Yan, Christopher Fletcher, and Josep Torrellas. Cache Telepathy: Leveraging Shared Resource Attacks to Learn DNN Architectures. In *USENIX Security Symposium*, 2020.

[66] Mengjia Yan, Read Sprabery, Bhargava Gopireddy, Christopher W. Fletcher, Roy H. Campbell, and Josep Torrellas. Attack Directories, Not Caches: Side Channel Attacks in a Non-Inclusive World. In *IEEE Symposium on Security and Privacy (S&P)*, 2019.

[67] Mengjia Yan, Jen-Yang Wen, Christopher W Fletcher, and Josep Torrellas. SecDir: a secure directory to defeat directory side-channel attacks. In *International Symposium on Computer Architecture (ISCA)*, 2019.

[68] Fan Yao, Milos Doroslovacki, and Guru Venkataramani. Are Coherence Protocol States Vulnerable to Information Leakage? In *IEEE Symposium on High Performance Computer Architecture (HPCA)*, 2018.

[69] Fan Yao, Guru Venkataramani, and Miloš Doroslovački. Covert timing channels exploiting non-uniform memory access based architectures. In *Proceedings of the on Great Lakes Symposium on VLSI*, 2017.

[70] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-channel Attack. In *USENIX Security Symposium*, 2014.

[71] Tianwei Zhang, Yinqian Zhang, and Ruby B Lee. Cloudradar: A Real-time Side-channel Attack Detection System in Clouds. In *Research in Attacks, Intrusions, and Defenses (RAID)*, 2016.

[72] Li Zhao, Ravi R. Iyer, Srihari Makineni, Don Newell, and Liqun Cheng. NCID: a Non-inclusive cache, Inclusive Directory Architecture for Flexible and Efficient Cache Hierarchies. In *Conference on Computing Frontiers*, 2010.

Appendix

A Access Time

The access times of various memory access operations are given in Figure 9 (for 1000 measurements). As the bins show, the access times easily allow distinguishing the operations. In the case of secondary device reads from L2 and LLC, the histograms do show a small overlap. However, during the experiments, we rarely need to determine L2 accesses, but often need to differentiate LLC and RAM accesses.

B Cache Manipulation Details

This section provides more insight into the migration between different cache levels in the non-inclusive hierarchies of our test platforms (cf. Section 2.6). We also relate this to statements and observations of related work. Please refer to our repository for supporting experiments.

When a line in memory is referenced by one of the cores, it is installed in the L2 of that core, and tracked in the inclusive coherence directory (CD) [66]. If the line is then referenced by another core (cf. Figure 4d), it is installed in that core's L2 as well. In addition, we observe that the line now has a copy in the LLC, likely as a performance optimization. We find this to happen immediately, i.e., already after one shared access, in contrast to earlier studies that state it to happen only after several accesses by processes on both cores [66], or not at all [67]. Note that this does not contradict the non-inclusive invariant of the LLC; lines in L2 *may* reside in the LLC simultaneously, but do not *have to* as they would in inclusive hierarchies. Lines for which there is a copy in the LLC are no longer tracked by the CD, i.e., their tag migrates to the LLC along with the data. Thus, contention on the CD (Figure 4b) no longer evicts such lines, as they no longer have a CD entry.

On Skylake-X, Yan et al. [66] find the associativity of the structure that carries shared lines to be 11, matching with our hypothesis that shared lines move to the LLC ($W = 11$).

Our experiments confirm earlier statements [66, 67] that lines evicted from the LLC are evicted from everywhere, i.e., they do not move back to L2 (with corresponding tracking in CD), although it would be theoretically possible.

In summary, lines *only in L2 caches* are not evicted by LLC contention as they do not have an entry there, in contrast to inclusive LLCs. However, they can be moved to the LLC with CD contention ([66], Figure 4b) or, as we discover, with a shared access (Figure 4d) or writes by the secondary device (Figure 4e). In contrast, lines with *at least one copy in the LLC* are not evicted by CD contention, but can be evicted from the entire hierarchy with contention on the LLC.

Clarifying Attacks. For lines that are only present in the LLC, e.g., after L2 contention (Figure 4a) or CD contention (Figure 4b), what happens when they are referenced again by a CPU core? We observe that *unshared* lines move back to L2 (and become tracked in CD), and that *shared* lines remain in the LLC and are copied to L2 (still tracked in LLC).

The behavior for shared lines is essential to understand EVICT+RELOAD (both ours and [66]). The fact that shared lines remain to have an LLC copy is the reason why our EVICT+RELOAD attacks need to evict the shared line from the entire cache hierarchy. If we would only move it from L2 to LLC, later accesses by the victim core would not be observable. The line will be served to the attacker from the faster LLC, regardless of a victim access.

The behavior for unshared lines is essential to understand PRIME+PROBE [66]. When an unshared line is read from LLC, it is moved to L2, tracking it in the CD again (and hence, evicting another entry from there). Although this transfer of tracking from LLC to CD was not mentioned explicitly in [66], we believe it to be consistent with (and necessary for) their PRIME+PROBE attack, where unshared lines migrate back and forth between the LLC and the CD.

C Replacement Policy

Table 10 covers access patterns using three congruent lines {A, B, C} and compares the observed (2-way associative) DDIO region contents against the ones expected with various replacement policies, e.g., LRU, variants of Quad-Age LRU, and RRIP. To reset the replacement policy, we first fill the region of interest with congruent lines {D, E, F} and flush them. The patterns are placed to the DDIO region with secondary writes, where every measurement is repeated 1000 times.

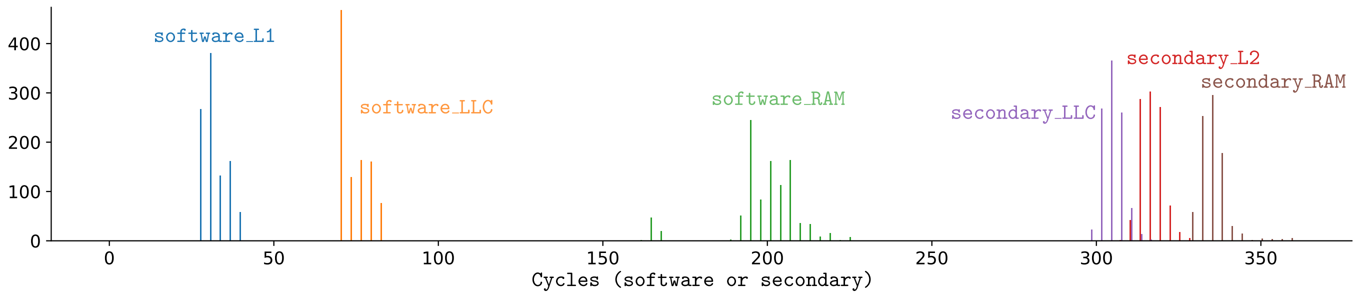


Figure 9: A histogram showing the timing of indicated memory operations.

Table 10: Replacement Policy Experiment (DDIO). Discrepancies between expected and observed states are indicated.

Access	Cache	Expected Contents with				
Patterns	Cont.	LRU	RRIP	QUAD Variants		
CBBAC	BC	AC	CB	AC	AC	CA
CBBBA	AB	AB	AB	AB	AB	BA
CBBCA	AC	CA	CA	CA	CA	BA
CBCAA	AC	CA	CA	CA	CA	AC
CBCAB	AB	BA	CB	CB	BA	AB

To our knowledge, the replacement policy for non-inclusive Skylake and Cascade Lake CPUs has not been reverse-engineered. Though we do not reverse-engineer the replacement policy in this work, the knowledge that the replacement policy is not plain LRU is already useful for our experiments.

We conducted similar tests for DDIO$^+$ lines, and observed an LRU-like policy. However, an exact reverse engineering of that policy is complicated, as consecutive DDIO$^+$ writes move a line from LLC to L2, and then back to LLC, where it may be considered to be a new address, instead of a cache hit. We leave it as future work.

D Revisiting Eviction (Inverse Variant)

Algorithm 5 Eviction without CD - Inverse

On the right, the LLC and L2 states of victim ($L2_V$) and attacker ($L2_A$) are shown for each operation on the target address (T).

```
                          L2_V  L2_A  LLC      L2_V  L2_A  LLC
1: ? vic_CpuRd                   ✓                    ✗
                          [T]   [ ]   [ ]      [ ]   [ ]   [ ]
2: att_CpuTime               Remote L2                RAM
                          [T]   [T]   [T]      [ ]   [T]   [ ]
3: hlp_CpuRead            [T]   [T]   [T]      [ ]   [T]   [T]
4: att_CpuEvict_LLC       [ ]   [ ]   [ ]      [ ]   [ ]   [ ]
```

Algorithm 5 exposes the inverse version of Algorithm 3. The helper process (hlp) is running on another core as the main attacker process. It is added to ensure that the line moves to LLC even if the victim does not access it, in order to remove it from the cache hierarchy. Similarly, Algorithm 6 is the inverse version of Algorithm 4. Both inverse algorithms are slightly more complex, but seem to work across CPU sockets.

Algorithm 6 Eviction with Reduced Eviction Set - Inverse

On the right, the attacker's L2 and LLC (with DDIO$^+$ and DDIO) are shown, with lines 0-3, LLC magnet ways (M), and target (T) .

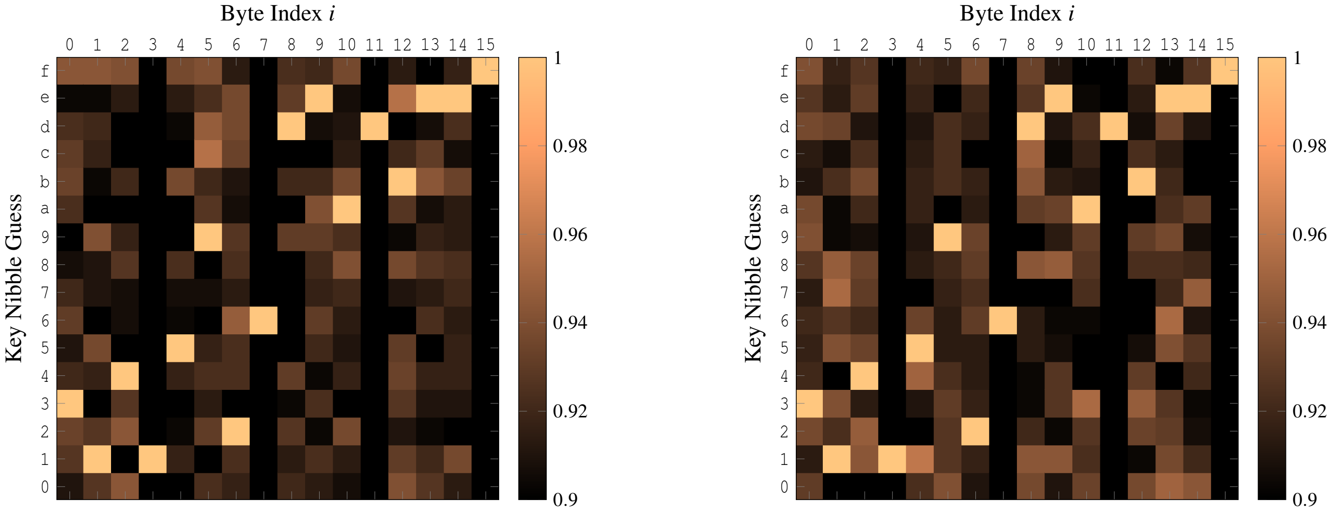
```
Prepare:                          L2       LLC
  1: att_SecWr (0,1)              [   ]    [    |1 0]
  2: att_CpuWr (0,1)              [1 0]    [    |M M]
  3: att_SecWr (0,1)              [   ]    [1 0 |M M]
Wait:
  4: ? vic_CpuRd (T)              ✓                      ✗

Measure & Reinstate:
  5: att_CpuTime (T)                  Remote L2                RAM
                                  [   ]  [1 0|M T]       [T  ]  [1 0|M M]
  6: hlp_CpuRd (T)                [   ]  [1 0|M T]       [   ]  [1 0|M T]
  7: att_SecWr (2,3)              [   ]  [1 0|3 2]       [   ]  [1 0|3 2]
  8: att_CpuWr (2,3)              [3 2]  [1 0|M M]       [3 2]  [1 0|M M]
  9: att_SecWr (2,3)              [   ]  [3 2|M M]       [   ]  [3 2|M M]
Reiterate from line 4, swapping the roles of 0-1 and 2-3.
```

E AES T-Tables

To speed up the computation of the AES-128 block cipher, table-based implementations consolidate a large part of the round function into precomputed tables. The secret key k and (known) plaintext p consist of 16 bytes (resp, k_i and p_i, for $0 \leq i < 16$). In every round, four tables are referenced, and for the majority of the encryption rounds, these are tables Te_{0-3}.

The proof-of-concept implements the first-round attack by Osvik et al. [41], monitoring accesses to the first cache line of tables Te_{0-3} for known plaintexts. The cache line corresponding to $Te_j[p_i \oplus k_i]$ is *always* accessed in the first round, where $j = i \bmod 4$. All other lines in Te_j are *often* accessed, yielding a statistical difference between the cache line corresponding to $p_i \oplus k_i$, and the other lines of the table. By



(a) Cross-Core ($\mathcal{V}_{\mathtt{CC}}$)

(b) Cross-Socket ($\mathcal{V}_{\mathtt{CS}}$)

Figure 10: Reduced eviction ($\mathrm{EV} = 4$) pattern for AES T-tables on the ACE1 platform (cf. Section 2.6). Upper key nibbles (`0x31415926deadbeef`) can be read from the columns (full key used: `0x3010401050902060d0e0a0d0b0e0e0f0`)

Table 11: The FPGA Resource Utilisation of Our Accelerator

Design	PAC Type	ALMs (%)	REGs	RAM
Ours	A10	31,993 (7%)	46,382	119
Hello FPGA	A10	30,759 (7%)	42,398	101
Ours	S10	53,497 (6%)	67,266	157
Hello FPGA	S10	53,095 (6%)	63,323	139

identifying the plaintext byte p_i for which the first cache line of $Te_{(i \bmod 4)}$ is accessed always, the attacker learns the upper $\log_2(\frac{256}{\mathtt{CLsize}})$ bits of k_i. On modern platforms, with cache lines of $\mathtt{CLsize} = 64$ bytes, this amounts to the upper four bits (i.e., half-byte or nibble) of k_i. To learn which p_i leads to the first line always being accessed, the attacker submits plaintexts with fixed p_i and all other bytes random. This can be repeated for all 16 key bytes k_i (always fixing plaintext byte p_i).

Figure 10 visually shows the described attack on AES T-Tables using reduced eviction sets ($\mathrm{EV} = 4$), resp. for the cross-core and cross-socket reduced eviction. For each combination (byte index i, key nibble hypothesis), the example performs 300 encryptions. The columns clearly reveal the upper nibble of every key byte. For the cross-core setting, 300 encryptions per combination are sufficient to recover the full key in all of 100 runs. For the cross-socket setting, the same amount of successful encryptions is also sufficient, but the target line may get stuck in the other socket (cf. Section 8.3). This happens once per 12750 encryptions (average over 10 million encryptions). When it does, it can be detected, and our proof-of-concept flushes the line from the cache. Hence, in practice, an attacker may prefer to use full eviction in the cross-socket case to increase the robustness against this event.

Many works have extended the original first-round attack to full key recovery, as well as significantly reducing the number of required encryptions. Such optimizations, while useful, are out of scope for this attack example.

F FPGA Utilisation

Table 11 shows the resource utilization of our FPGA hardware accelerator on two different Intel Programmable Acceleration Cards (PAC), with the most important column being the Adaptive Logic Modules (ALM), which consist of both programmable logic and registers (REGs). It is built based on the *hello fpga* sample project provided by Intel[3]. Hence, it consists of Intel-provided modules for interfacing with it from software. The increase in utilization of our accelerator compared to the sample project reflects its actual utilization. The accelerator is very compact, as it barely increases the amount of total FPGA resources (ALMs).

[3] The `03a_hello_world_mpf` sample from the commit `#ec7e78f` of `https://github.com/OPAE/intel-fpga-bbb`