TLB;DR: Enhancing TLB-based Attacks with TLB Desynchronized Reverse Engineering

Andrei Tatar[†*] Daniël Trujillo[†‡*] Cristiano Giuffrida[†] Herbert Bos[†]

† Vrije Universiteit, Amsterdam ‡ ETH Zurich

[*] *Equal contribution joint first authors*

Abstract

Translation Lookaside Buffers, or TLBs, play a vital role in recent microarchitectural attacks. However, unlike CPU caches, we know very little about the exact operation of these essential microarchitectural components. In this paper, we introduce TLB desynchronization as a novel technique for reverse engineering TLB behavior from software. Unlike previous efforts that rely on timing or performance counters, our technique relies on fundamental properties of TLBs, enabling precise and fine-grained experiments. We use desynchronization to shed new light on TLB behavior, examining previously undocumented features such as replacement policies and handling of PCIDs on commodity Intel processors. We also show that such knowledge allows for more and better attacks.

Our results reveal a novel replacement policy on the L2 TLB of modern Intel CPUs as well as behavior indicative of a PCID cache. We use our new insights to design adversarial access patterns that massage the TLB state into evicting a target entry in the minimum number of steps, then examine their impact on several classes of prior TLB-based attacks. Our findings enable practical side channels à la TLBleed over L2, with much finer spatial discrimination and at a sampling rate comparable to L1, as well as an even finer-grained variant that targets both levels. We also show substantial speed gains for other classes of attacks that rely on TLB eviction.

1 Introduction

Deeper knowledge of microarchitectural components frequently leads to significant advances in low-level attacks and defenses. For instance, reverse engineering CPU cache replacement policies [1, 2, 34] allowed researchers on the defensive side to ensure bounds on execution time and to protect against side channels [1], and on the offensive side, to leak information through the replacement policy state [36] or optimize cache eviction for more effective Rowhammer attacks [4]. Likewise, knowledge of the interplay between the memory management unit, caches, and virtual-to-physical address translation structures enabled new information leakage attacks through trusted CPU components [14, 32]. However, most reverse engineering efforts to date have focused exclusively on the data and instruction CPU caches, with little attention devoted to the translation lookaside buffer (TLB), the other vital cache component involved in memory accesses. This is somewhat surprising, as TLBs play a pivotal role in a growing number of attacks [14, 32, 39]. Aside from crude examinations (e.g., by Gras et al. [14]), researchers have not looked at the detailed behavior such as allocation and replacement policies of TLBs at all.

In this paper, we show that our lack of knowledge of the operation of the TLB hinders the exploration of new attack vectors and limits the efficiency of attacks proposed so far. For example, the absence of optimal eviction strategies limited the TLBleed covert channel [14] to coarse-grained information over the L1 data TLB, rather than more fine-grained channels over the larger and shared L2 TLB. In this paper, we reverse engineer modern TLBs and show how knowledge of replacement policies not only improves existing attacks but also makes new attack variants practical.

In contrast to previous research, we do not leverage performance counters or timing side channels for reverse engineering, but rather use a precise technique based on *TLB desynchronization*. In particular, we desynchronize the TLB and the page table structure in memory by altering the page table entry (PTE) for a page in the TLB. Doing so allows us to determine very precisely when a TLB entry is evicted, because that is when the processor starts using the new mapping from the page tables in memory.

Using TLB desynchronization as a side channel, we examine TLB features that previous work has omitted, such as replacement policies and PCID handling, and in doing so are the first to uncover several new undocumented properties. Specifically, we describe a new replacement policy in use by the 12-way L2 TLBs of recent Intel microarchitectures, as well as behavior indicative of a previously undocumented PCID cache in all the tested CPUs.

Finally, by creating a more complete model of hardware behavior, we show how we can gain net speed benefits for prior

attacks that rely on the TLB such as TLB side channel [14], as well as translation-based cache [32] and Rowhammer [39] attacks. Our reverse engineering efforts also make new attack variants practical, such as an efficient TLBleed over the L2 TLB. Such a variant provides 8 times the spatial discrimination of the original TLBleed, while our optimized eviction strategy provides a sampling rate comparable to the L1 sampling rates of the original TLBleed. Further, we introduce an even more fine-grained variant targeting a particular (L1, L2) set pair, improving spatial discrimination by two orders of magnitude, and show how optimized eviction sets are necessary for the practical implementation of such an attack.

Contributions

- We introduce TLB desynchronization as a reliable side channel for TLB reverse engineering and use it to (a) reproduce existing results, and (b) uncover entirely novel TLB properties and behavior.
- We discover previously undocumented TLB behavior on Intel CPUs, including a novel replacement policy, and behavior congruent with an undocumented PCID cache.
- We use this knowledge to introduce optimized eviction sets and show how they improve TLB-based attacks.

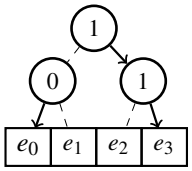
Our code to build optimized eviction sets and implement the considered case studies is available as open source at https://github.com/vusec/tlbdr.

2 Background

In this section we briefly describe TLBs and some of their fundamental properties. We then lay out previous reverse engineering work and attacks involving the TLB.

2.1 Virtual Memory and TLBs

Abstracting a machine's available RAM through the use of virtual memory is a nearly ubiquitous technique used in modern computer architectures. A common implementation of this abstraction is paging, where memory is divided into physical page frames—equally-sized contiguous blocks. The virtual address space is similarly divided into virtual pages. Translating between virtual and physical address spaces is handled by the Memory Management Unit (MMU) inside the CPU, which uses hierarchical in-memory data structures called page tables to accomplish this task. Page tables contain page table entries (PTEs) which point to a page frame containing either the requested data or to another, lower-level, page table. This translation performed by the MMU is called a page table walk, owing to the traversal over page table structures it involves.



(a) A possible ordering.

$$\begin{aligned}\pi_0 &= (0,1,2,3)\\ \pi_1 &= (1,0,3,2)\\ \pi_2 &= (2,1,0,3)\\ \pi_3 &= (3,0,1,2)\end{aligned}$$

(b) Permutation vectors.

Figure 1: 4-way tree-PLRU

Translation Lookaside Buffer As multiple memory accesses can be expensive, recently resolved translations are stored in a cache called the Translation Lookaside Buffer (TLB) in order to speed up subsequent accesses. The TLB, like data caches, is a set-associative cache partitioned into S sets, each consisting of a number of ways W, with a so-called hash function determining the set in which a translation is cached. Modern processors typically use a two-level TLB hierarchy. The first level (L1) is split in two components: the dTLB is used to cache translations triggered by data loads while the iTLB stores those triggered by instruction fetches. A second level (L2) TLB is consulted after an L1 miss and can be either shared (sTLB), storing translations from either memory access type, or itself split, handling data loads and instruction fetches independently. Similarly to other multi-level caches, the concepts of inclusivity and exclusivity apply to TLBs as well. An inclusive TLB is one where entries in L1 are necessarily also present in L2, whereas an exclusive TLB has entries present in at most one level at a time.

Replacement Policies Caches of all kinds quickly face the issue of storing a new translation into a full set, thus having to first pick an entry to evict. The method by which a cache selects this eviction victim is called a *replacement policy*. Optimally, a cache should evict the entry that will be used most distantly in the future [6]. Implementing such a replacement policy, however, is impossible, as the future is unknowable.

Fortunately, the past is often a reasonable approximation of the future. This forms the basis for the Least-Recently Used (LRU) replacement policy which keeps track of the access history of entries, always evicting the least-recently used. As the number of ways in a set increases, though, LRU scales very poorly and becomes infeasible. To address this, approximations of LRU, called Pseudo LRU (PLRU), are commonly implemented instead. One family of such policies is tree-PLRU, which use $W-1$ bits to represent a binary tree that approximates access history ordering. An example tree-PLRU ordering for a 4-way set is shown in Figure 1a. After an access, the tree is traversed and all the nodes encountered are updated to point in the opposite direction. On insertion, the entry currently pointed to is evicted and replaced, after which the nodes traversed in order to reach it change direction. More generally we can view a replacement policy as maintaining an ordering of ways in a set, forming a queue of eviction victims.

Permutation Vectors Although descriptions and visualizations of replacement policies are helpful, thorough reverse engineering efforts require the rigor of a mathematical model. Abel and Reineke [1] introduce *permutation vectors* as a model for a broad class of replacement policies where the state of a TLB set can be exhaustively described by a permutation (i.e., ordering) of its W ways that represents access history, with the rightmost element being chosen for eviction next. Permutations are also used to describe updates to cache state following hits or misses, with the new state formed by composing the initial state with the update permutation. A permutation vector π_i shows how each entry's position changes after a hit on the entry at position i. For example, if we access entry x at position 3 and $\pi_3(0) = 3$, x will move to position 0. A miss vector π_m shows what happens on a TLB miss, and is conventionally fixed to $(W-1, 0, 1, ..., W-2)$, i.e. the new entry is inserted at position 0 and other entries have their positions increased by one. Permutation vectors allow us to model many common replacement policies like LRU and tree-PLRU. In Figure 1b we see an example for 4-way tree-PLRU. Permutation vectors do have limitations, and are inadequate for describing non-deterministic policies, as well as policies whose decision is not solely based the access history (e.g., bit-PLRU [3], which replaces the left-most element that was not accessed recently).

Address Space Identifiers Since each process is offered its own virtual address space, translation structures are process-specific. Hence, upon context switch, any existing TLB entries become stale and should no longer be used for translation. A naive way around this is to flush the entire TLB on context switch, which although simple, has the newly scheduled task start off with an empty TLB, forcing the MMU to perform page walks for the first memory accesses. To avoid this so-called cold start, modern processors tag TLB entries with an address space identifier. This eliminates the need for a TLB flush on every context switch, as it allows the processor to distinguish between valid and stale entries. As recent transient execution attacks proved the need for a separate kernel address space [23], such identifiers became even more important for performance, avoiding the need for a TLB flush on every privilege switch. Intel processors refer to this identifier as Process-Context Identifier (PCID).

2.2 Exploiting the TLB

Gras et al. [15] introduce AnC, a cache attack on the MMU that can be used to break ASLR. Their insight is that the MMU needs to access multiple levels of page tables in order to complete a translation, and each accessed entry is cached. By using an established `EVICT+TIME` attack on these cached entries they can leak the value of any data pointer. RevAnC, introduced by van Schaik et al. [33], uses repeated AnC attacks to reverse engineer the sizes of the TLB and translation caches on multiple architectures. XLATE [32] similarly uses the translation process as a cross-process side channel.

In TLBleed [14], Gras et al. show that the TLB itself can be abused to spy on a co-resident hyperthread. Specifically, they use the L1 dTLB as a side channel to leak the key bit being processed by the libgcrypt EdDSA key scalar-multiplication function on a victim thread. They successfully recover the secret key using a limited number of brute force attempts. In the process, they reverse engineer several properties of modern Intel TLBs, showing that the L1 dTLB and the L2 sTLB are competitively shared between hyperthreads, while the L1 iTLB is statically partitioned. Next to that, they determine set sizes, set selection hash functions, and also show that the TLB is non-inclusive.

Zhang et al. [39] show that page walks can be leveraged to perform Rowhammer, in an attack they call PTHammer. By flushing the TLB and cache entries for the last-level PTE they ensure this entry is fetched from DRAM on each page walk. These memory accesses are used to hammer DRAM on a row that is not directly accessible to the attacker, invalidating the threat model of many software-based Rowhammer defenses.

All TLB-based attacks described here indispensably rely on evicting one particular entry out of the TLB. Currently they achieve this using a blunt approach: fill the target set with W or more new entries to *force* the target entry out. In §5 we propose a finer-grained and more performant alternative.

3 TLB Desynchronization

In this section, we introduce TLB desynchronization as a novel primitive for reverse engineering the TLB that, unlike previous work, relies neither on timing [1, 33], nor on hardware performance counters [1, 2, 14].

Our starting insight is that unlike data caches, which feature intricate coherency protocols, TLBs do not enforce coherence with the in-memory page tables. The task of invalidating any potentially stale TLB entries is explicitly left to the operating system [19] and failing to do so would be a serious bug in normal circumstances. For our purposes of reverse engineering, however, we can leverage stale entries to accurately determine whether a given memory access incurred a hit in the TLB.

To do so, we first select a victim address and trigger a TLB fill by accessing it. If we alter the corresponding PTE directly afterward without explicitly invalidating the TLB, the victim entry becomes *desynchronized* with the PTE in memory. Any subsequent memory access to the victim address that hits the TLB will use the stale entry, whereas any TLB miss will use the in-memory PTE. If we can then distinguish which PTE is used for translation, we enable reliable TLB hit detection with the finest possible granularity—individual accesses.

Desynchronization poses several attractive properties when compared to alternatives. As opposed to performance counters, it does not rely on sampling and thus precisely measures

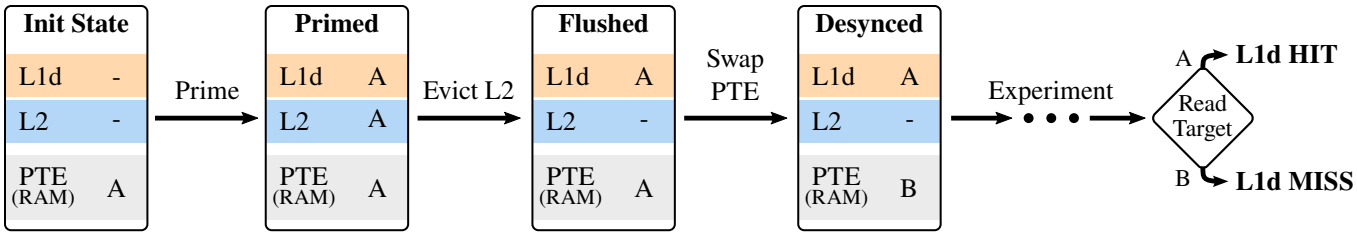


Figure 2: Overview of a TLB desynchronization experiment on L1d with pages A and B of different contents.

specific memory accesses independently. In addition, desynchronization classifies hits and misses based on fundamental properties of the TLB, offering a much clearer and more robust classification than timing measurements, which require calibration and are more prone to noise.

Before we can leverage desynchronization to reverse engineer the TLB we must address several challenges.

Distinguishing Memory Accesses A first issue we face is that we must distinguish between memory accesses going through a stale (in-TLB) entry versus a fresh (in-memory) PTE. A straight-forward solution would be to mark the PTE invalid by setting a reserved bit or clearing the present bit. A subsequent TLB miss would then result in a page fault, which we can easily measure. However, this method would execute the page fault handler on every TLB miss, potentially thrashing the TLB state and adversely affecting measurements.

We instead opt for *PTE swapping*, by which we interchange the contents of our victim PTE with another, also valid, PTE. As a result of desynchronizing and PTE swapping, the victim page maps to different page frames when accessed via a TLB hit or page table walk. By ensuring that these physical pages contain different data we can easily and precisely classify TLB hits and misses by interpreting these contents.

Component classification Although we can distinguish between a TLB hit and a TLB miss, for non-exclusive TLBs we still cannot easily determine which component caused a hit (e.g., whether a data load was served from L1d or L2). In certain cases when testing the sTLB we can use a data load for priming and an instruction fetch for testing presence in the TLB, or vice versa, to eliminate influence from L1. To solve this issue more generally, we selectively flush an entry from a single TLB component without affecting the others (e.g., flush an entry from L2 while preserving it in L1). Selectively flushing L2 can be done naively by issuing a large number of memory accesses of the opposite type to the split TLB we want to preserve, assuming a shared L2. For example, after initiating enough instruction fetches on unique pages, we are certain that any cached entries are evicted from both L1i and L2, while leaving L1d unaffected. Figure 2 shows an overview of such a TLB desynchronization experiment.

More precise eviction can be achieved by targeting a single specific set, requiring fewer, albeit more carefully chosen memory accesses. This, for example, enables us to evict an entry from L1d without evicting any entries that are relevant for our experiment from L2. For this to work, however, we need to know the hash functions used for set indexing.

Implementation To experiment with desynchronized TLBs, we need to handle several issues. First, userspace programs cannot directly access or modify page tables, forcing us to implement PTE swapping in kernel mode. Second, scheduling events such as context switches can thrash the TLB state and contaminate our results, prompting an implementation that minimizes task switching. Third, we want to keep the amount of out-of-order execution to a minimum to avoid polluting our results. Finally, asynchronous events such as interrupts can also affect the TLB state and should be minimized. Taking all these considerations into account, our reverse engineering experiments run entirely in kernel mode with preemption and interrupts disabled, and use a strict form of pointer chasing.

4 Reverse Engineering

In this section we discuss how to use the TLB desynchronization primitive to design and run experiments for probing TLB behavior. In designing the experiments presented in this section, we assume a shared L2 TLB, an assumption we found not to hold for two AMD microarchitectures examined. For such systems, we lay out an alternative set of experiments in Appendix A. The results for the two AMD microarchitectures are summarized later in this section, in Table 2.

We first examine a range of basic TLB properties of Intel CPUs that confirm and expand the insight into TLB behavior of previous work [14]. From §4.4 onward, we focus on features that were thus far unexplored, yet no less fundamental. In the process, we uncover a previously undocumented replacement policy, as well as evidence of a PCID cache.

4.1 Inclusivity and Exclusivity

Before we pursue more advanced experiments we must first confirm several fundamental properties, and we start with inclusivity. To do so, we prime the TLB with a data load of our target address T, swap its PTE to desynchronize the TLB

from the page tables, then perform unique instruction fetches until the L2 sTLB is completely flushed. If a subsequent data load of T is a TLB hit, the TLB must have been non-inclusive, as L1d retains T independently of L2. A miss implies either strict inclusivity, or that L1 is not populated by page walks. On all tested microarchitectures, we obtain proof of a non-inclusive TLB, confirming the results of Gras et al. [14].

Knowing this, we expand upon their work by performing a similar experiment to check for *exclusivity*. Specifically, we prime the TLB with a data load, desynchronize, then access T via an instruction fetch. A TLB hit will occur if and only if T was also loaded into L2 in addition to L1d, and thus a non-exclusive TLB. Our results show that all tested microarchitectures are non-exclusive, as the second access incurs an sTLB hit. We conclude that all our tested systems implement non-inclusive, non-exclusive TLBs. In addition, we know that entries are inserted into both TLB levels by a page walk.

4.2 Set Size and Mapping

Now that we know where entries may reside in the TLB and how they are inserted by the MMU, we use TLB desynchronization to measure the number of ways and the set hash functions, again extending earlier results [14].

We consider a TLB component with S sets and W ways that uses one of the two types of hash functions found before on TLBs [14]. Assuming we start with an empty TLB and no evictions occur before a TLB set is full, visiting $W+1$ virtual pages belonging to said set should cause evictions for correct values of W and S. Evictions should also occur if we pick W greater than the true number of ways, or if we pick S a multiple of the true number of sets. Therefore, the smallest values for both W and S that trigger TLB evictions denote the correct number of ways and sets. To find this minimal pair we run our experiment for all sensible combinations of W, S, and candidate hash function. Directly after accessing our pages, we swap their PTEs to desynchronize the TLB and page tables. We next access the pages again, measuring if any access incurred a miss. If so, we know that the first round of accesses resulted in a TLB eviction. To eliminate the influence of TLB components that we are not testing (e.g., false hits from stale entries in L2 when testing L1), we use the techniques described in §3.

Our results, shown in Table 1, agree for the sTLB and dTLB with the ones described by Gras et al. [14]. On the iTLB, however, the measured number of sets depends on the activity of the co-resident hyperthread for all tested microarchitectures except the oldest one, Westmere-EP. With the hyperthread active, our results are consistent with previous work and with static partitioning of L1i. But when idle, the number of sets *doubles* (i.e., our experiment had access to both "halves" of the iTLB), suggesting that partitioning occurs *dynamically*, with the iTLB switching between partition states at runtime.

4.3 Reinsertion Behavior

Now that we know how the TLB populates on a miss, we next examine reinsertion on hits. In particular, we want to know whether L2 serves as a victim cache, whether L2 hits propagate upward, and whether L1 hits propagate downward.

We start by looking at victim caches, specifically whether L2 sTLB is populated by evictions from L1d or L1i. To do so, we prime either L1d or L1i with our target T, desynchronize the TLB, evict L2, and finally evict our stale entry from L1. A subsequent hit on T means that evicting L1 populated L2, implying a victim cache, whereas a miss suggests otherwise. Our results show that entries are never reinserted into L2 after either L1d or L1i eviction. None of the L2 sTLBs tested behave as a victim cache.

We next investigate whether an L2 hit is reinserted into L1. For this, we access T, desynchronize the TLB, then access T using the opposite type of memory access, which should cause an L2 hit. Finally, we evict the L2 set and access T again using the latter access type. If L2 hits populate L1 we should see a hit, while a miss indicates otherwise. For all tested microarchitectures our results show that the last memory access results in a TLB hit. Hence, entries are reinserted into the dTLB or the iTLB upon an sTLB hit.

As a last experiment, we test if the TLB reinserts the other way around—into the sTLB upon L1 hit. For this, we access a target page T, desynchronize the TLB, and evict its sTLB set, leaving the entry only in L1. We access the target again, using the same access type. If this causes sTLB reinsertion, an access of the opposite type results in a TLB hit. Likewise, a TLB miss proves that entries are not reinserted. Our results show no evidence of L2 reinsertion in any of the TLBs.

4.4 Replacement Policies

With the insights gained thus far we have an overview of where and how entries are inserted into the TLB. We use this knowledge to explore replacement policies, using permutation vectors as a model for reverse engineering. We implement the algorithm presented by Abel and Reineke for regular caches [1], but for the TLB components, using TLB desynchronization.

4.4.1 Experiment Design

We first test whether we can fill all W ways of a TLB set by initiating W accesses to unique virtual pages mapping to the same set. After each access, we swap the corresponding PTE and finally access the W pages again to see if any of them is a TLB miss. Although similar to the experiment in §4.2, the fundamental difference here is that we start with a full set instead of an empty set. The results confirm that after visiting W pages, all their entries are present in the TLB set, on all tested microarchitectures and TLB components. This is in line with the conventional miss vector used in previous work.

$$
\begin{array}{lcl}
\pi_0 & = & (0,\ 1,\ 2,\ 3,\ 4,\ 5,\ 6,\ 7,\ 8,\ 9,\ 10,\ 11) \\
\pi_1 & = & (1,\ 2,\ 0,\ 4,\ 5,\ 3,\ 7,\ 8,\ 6,\ 10,\ 11,\ 9) \\
\pi_2 & = & (2,\ 0,\ 1,\ 5,\ 3,\ 4,\ 8,\ 6,\ 7,\ 11,\ 9,\ 10) \\
\pi_3 & = & (3,\ 1,\ 2,\ 0,\ 4,\ 5,\ 9,\ 7,\ 8,\ 6,\ 10,\ 11) \\
\pi_4 & = & (4,\ 2,\ 0,\ 1,\ 5,\ 3,\ 10,\ 8,\ 6,\ 7,\ 11,\ 9) \\
\pi_5 & = & (5,\ 0,\ 1,\ 2,\ 3,\ 4,\ 11,\ 6,\ 7,\ 8,\ 9,\ 10) \\
\pi_6 & = & (6,\ 1,\ 2,\ 3,\ 4,\ 5,\ 0,\ 7,\ 8,\ 9,\ 10,\ 11) \\
\pi_7 & = & (7,\ 2,\ 0,\ 4,\ 5,\ 3,\ 1,\ 8,\ 6,\ 10,\ 11,\ 9) \\
\pi_8 & = & (8,\ 0,\ 1,\ 5,\ 3,\ 4,\ 2,\ 6,\ 7,\ 11,\ 9,\ 10) \\
\pi_9 & = & (9,\ 1,\ 2,\ 0,\ 4,\ 5,\ 3,\ 7,\ 8,\ 6,\ 10,\ 11) \\
\pi_{10} & = & (10,\ 2,\ 0,\ 1,\ 5,\ 3,\ 4,\ 8,\ 6,\ 7,\ 11,\ 9) \\
\pi_{11} & = & (11,\ 0,\ 1,\ 2,\ 3,\ 4,\ 5,\ 6,\ 7,\ 8,\ 9,\ 10)
\end{array}
$$

Figure 3: Permutation vectors for the sTLB replacement policy of Skylake-SP, Kaby Lake and Coffee Lake(-S) CPUs.

The main intuition of the algorithm presented by Abel and Reineke [1] is that we can determine the position of an entry in the ordering by finding the minimum number of TLB fills required to evict the entry from the set. In general, the position of the entry is $W - n_{min}$, where n_{min} is the minimum number of TLB fills required to evict the entry and $1 \leq n_{min} \leq W$.

To find the permutation vectors, we first establish a known state in the TLB set by priming it with W pages. After that, we trigger any of the W permutations by causing a TLB hit on the entry with the corresponding position in the ordering. To determine what permutation vector we have triggered with the TLB hit, we find $n_{min}(x)$ for each entry x present in the set. We do this by desynchronizing x immediately after accessing it when priming, then determining the minimum number of TLB fills required to cause a TLB miss when accessing x afterwards. The new position of the entry is then $W - n_{min}(x)$. Hence, for each position i we determine the corresponding permutation vector π_i by deciding $W - n_{min}(x)$ for each entry x and placing each x_{pos} on this index in π_i. That is, $\pi_i(W - n_{min}(x)) = x_{pos}$ for each entry x and position i.

Before our experiment we first warm up the respective set by initiating a number of memory accesses on unique virtual pages to ensure the set is full. We do this because we wish to measure steady-state behavior, and replacement policies may behave differently when the set is not yet full [1, 2]. As in the other experiments, we eliminate the influence of TLB components not under test by selectively evicting that level.

To determine the new position of an entry we test a number of iterations, each using randomly selected sets. This results in a 2D matrix of $S \times W$ for each entry and permutation vector, where rows represent the different TLB sets and columns represent the possible positions in the ordering. A cell in this matrix shows the number of votes for a position, obtained by testing in a certain set. To find the correct new position for each entry, we take the column with the highest number of total votes, those in other columns being deviations, which we refer to as errors. The error rate is defined as the percentage of errors given the total number of votes. Essentially, we select the deterministic permutation vector model closest to our observations.

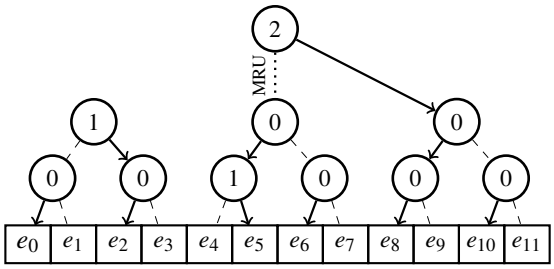


Figure 4: A possible sTLB ordering on Skylake-SP, Kaby Lake and Coffee Lake(-S) after an access to entry 7. Entry 8 is the next victim.

4.4.2 Results

Our results reveal a previously unknown replacement policy on the sTLB of Skylake-SP, Kaby Lake, and Coffee Lake(-S). Figure 3 shows the measured permutation vectors.

We recognize a possible implementation using a tree structure that behaves congruent with the permutation vectors. The 12 entries are divided into three groups of four entries each, with tree-PLRU keeping an ordering within each group. The root node indicates the current victim group, which is chosen based on the Most-Recently Used (MRU) group. Intuitively, if we number the groups 0, 1 and 2, we can say that the victim group is MRU+1 using modulo 3 arithmetic. A visual explanation of a possible ordering after an access to entry 7 can be seen in Figure 4. Here, the second group was most-recently used, and hence the root node is pointing to the third group. A subsequent hit on any entry in the third group (e_8, e_9, e_{10} or e_{11}) would make the first group the victim group, as $(2+1) \equiv 0 \pmod 3$. Finally, the nodes encountered while traversing to the entry within the third group are updated to point in the opposite direction, as with regular tree-PLRU. On TLB fill the victim entry is replaced and the state updated as if there had been a hit on the victim.

We are unaware of previous reverse engineering efforts that describe this policy. The most similar is $\text{LRU}_3\text{PLRU}_4$, found by Abel and Reineke [2] on the L1 data cache of Intel Ice Lake, where the 12 entries are also divided in three PLRU trees, with the victim group selected using LRU. Adopting their naming scheme, we refer to our found policy as $(\text{MRU+1})_{\%3}\text{PLRU}_4$. We ascertain that both of these policies belong to a family of tree-PLRU variants patented by Intel Corporation [5].

Table 1 summarizes the replacement policies we measured. On Westmere-EP, Ivy Bridge, and Haswell, we identified a tree-PLRU replacement policy on the sTLB. For the dTLB our results are consistent with tree-PLRU on all tested microarchitectures. The iTLB replacement policy on Westmere-EP

Table 1: Summary of reverse engineered properties on Intel.

TLB Property	Westmere-EP E5645	Ivy Bridge i3-3220	Haswell i7-4790	Skylake-SP Silver 4110	Kaby Lake i7-7700K	Coffee Lake(-S) i7-8750H, i9-9900K
Inclusive	✗	✗	✗	✗	✗	✗
Exclusive	✗	✗	✗	✗	✗	✗
L2 is victim cache	✗	✗	✗	✗	✗	✗
L2 hit inserts into L1	✓	✓	✓	✓	✓	✓
L1 hit inserts into L2	✗	✗	✗	✗	✗	✗
L1 dTLB						
Number of sets	16	16	16	16	16	16
Number of ways	4	4	4	4	4	4
Hash function	linear	linear	linear	linear	linear	linear
Replacement policy	tree-PLRU$_4$	tree-PLRU$_4$	tree-PLRU$_4$	tree-PLRU$_4$	tree-PLRU$_4$	tree-PLRU$_4$
Max PCIDs	N/A	N/A	N/A	N/A	N/A	N/A
L1 iTLB						
Number of sets	32	16 / 32[1]	8 / 16[1]	8 / 16[1]	8 / 16[1]	8 / 16[1]
Number of ways	4	4	8	8	8	8
Hash function	linear	linear	linear	linear	linear	linear
Replacement policy	tree-PLRU$_4$	LRU$_4$	tree-PLRU$_8$[2]	tree-PLRU$_8$[2]	tree-PLRU$_8$[2]	tree-PLRU$_8$[2]
Max PCIDs	1 / 4[3]	1 / 4[3]	1 / 4[3]	1 / 4[3]	1 / 4[3]	1 / 4[3]
L2 sTLB						
Number of sets	128	128	128	128	128	128
Number of ways	4	4	8	12	12	12
Hash function	linear	linear	linear	XOR	XOR	XOR
Replacement policy	tree-PLRU$_4$	tree-PLRU$_4$	tree-PLRU$_8$	(MRU+1)$_{\%3}$PLRU$_4$	(MRU+1)$_{\%3}$PLRU$_4$	(MRU+1)$_{\%3}$PLRU$_4$
Max PCIDs	4	4	4	4	4	4

[1] Depending on the activity of the co-resident hyperthread; see §4.2.
[2] Model closest to our observations, but very high error rate; see §4.4.2.
[3] Depending on whether the *NOFLUSH* bit is set when switching PCIDs; see §4.5.

corresponds to tree-PLRU, whereas Ivy Bridge uses perfect LRU. On Haswell, Skylake-SP, Kaby Lake, and Coffee Lake(-S), the closest model to our observations is tree-PLRU, albeit with a higher than usual error rate—where errors denote sets that do not behave consistently across iterations.

4.4.3 Error Rates

Our most stable results are on the sTLB of all microarchitectures, with error rates of less than 1%. The dTLB error rate on all microarchitectures is less than 8%, but since measurement code itself is active in at least one of the 16 sets, the majority of measurements in that set are off. The policies of the iTLB on Westmere-EP and Ivy Bridge—tree-PLRU and LRU—show an error rate of less than 5%, but, again, our code resides in at least one set.

The exact context in which we conduct the experiment influences the error rate. For example, writing to the CR3 register before the experiment starts typically increases the error rate significantly, while walking over the pointer chain twice, recording only the second outcome, decreases it somewhat. In all cases the measured permutation vectors remain either unchanged or clearly erroneous.

Establishing the exact cause of this behavior is very difficult. We can exclude non-deterministic replacement policies, as we achieve very low error rates with deterministic models. Likewise, an adaptive policy would be unlikely for multiple reasons. First, we can exclude set dueling, as we did not find proof of difference in behavior among specific sets. Second, any variant policy would itself be measurable as a different, coherent replacement policy some of the time, and we have found no evidence of such. Therefore, we speculate that the fluctuation in error rate is caused by unexpected memory accesses during the experiment. Possible causes could be System Management Mode (SMM), non-maskable interrupts, and TLB prefetching, which we know is used by AMD [8, 9].

4.5 PCID Support

The currently active PCID corresponds to the 12 least significant bits of the CR3 register. To investigate the role PCIDs play in TLB behavior, we attempt to isolate an entry in the L1 dTLB similarly to previous experiments. This time, however, we prime and check the TLB using a different PCID than the one used for eviction. The results were surprising: in none of the cases did we witness a TLB hit.

One possible explanation we considered was the TLB getting flushed on switching PCIDs. The Intel architecture manual [19] offers no strong guarantees about how the TLB behaves in this scenario, claiming it may or may not retain entries. In addition, when switching from one PCID to another, software needs to explicitly hint to the CPU not to flush cached entries corresponding to the new PCID, a hint which the CPU is free to ignore. To provide the hint, it sets the 63rd

(*NOFLUSH*) bit of the CR3 register on a context switch.

Due to the lack of information on this topic we conduct a systematic study of the interplay between PCIDs and TLB behavior. We later identified a patent by Intel describing implementations that behave in line with some of our findings [11].

We first investigate the maximum number n of concurrent PCIDs supported by a TLB. We start by accessing a randomly chosen page and desynchronize its TLB entry. We then successively switch to n different PCIDs, where $0 \leq n \leq 4095$. We do this both with and without setting the *NOFLUSH* bit. Lastly, we switch back to the original PCID while setting the *NOFLUSH* bit and measure an access to the target page. If we witness a TLB hit, we have not reached a maximum yet, and we increase n. If we see a TLB miss on every iteration, we have found a limit on the PCID support. As done previously, other TLB components are evicted as described in §3.

As a follow-up experiment, we investigate whether a hyperthread can evict entries of the other hyperthread by switching to new PCIDs. For this purpose, we create two kernel threads scheduled on the same physical core. In thread 1 we visit a randomly selected target page, desynchronize its TLB entry, then temporarily switch to another PCID. Next, on thread 2, we switch between all possible PCIDs in order. Lastly, in thread 1 we switch back to its original PCID with the *NOFLUSH* bit set, and measure our target.

We present our results for each TLB component, as summarized in Table 1. The results show that the dTLB does not support PCIDs at all, as even when $n = 0$ the second access results in a TLB miss. Hence, the dTLB is not aware of the currently active PCID and flushes all entries upon a CR3 load. The immediate security implication of this result is that the recent TagBleed attack [22] cannot abuse the dTLB. The second experiment shows that one hyperthread's PCID switch does not evict entries of the other hyperthread. This suggests that entries are tagged with their owning hyperthread.

In contrast, we found that the sTLB does support PCIDs on all tested microarchitectures, with no difference in behavior between having the *NOFLUSH* bit set or not. When $0 \leq n < 4$ the last access results in a TLB hit, while as soon as $n = 4$ we witness TLB misses, without exception. We conclude that the sTLB supports a maximum of 4 PCIDs. As with the dTLB, entries belonging to one hyperthread are not evicted by PCID switches on the other hyperthread.

Our experiments show that the iTLB is also aware of PCIDs. When not setting the *NOFLUSH* bit our entry is consistently evicted when $n = 1$. However, when the *NOFLUSH* bit is set, we measure evictions for unstable values of n, ranging from 1 to 4. We were not able to find an explanation for this behavior. As with the dTLB and sTLB, PCID switches on one hyperthread do not evict entries belonging to the other hyperthread, supporting the hypothesis of an iTLB that is wholly partitioned between active hyperthreads.

4.6 PCID Cache

The fact that there is a limit on the number of distinct PCIDs for which a TLB can hold entries suggests the existence of a cache for holding PCIDs, which we refer to as a *PCID cache*. Only PCIDs present in these caches may have their entries in a TLB component. Having such a cache would mean TLB entries need only store the bits to index into a small cache denoting their assigned address space, as opposed to storing all 12 bits of a PCID. As the sTLB supports 4 PCIDs, its PCID cache has 4 entries and because PCID switches on one hyperthread do not evict entries of the other hyperthread, it follows that each hyperthread has its own instance.

As a cache with limited space, it must implement some form of replacement policy to decide which PCID to evict when full. We attempt to measure this behavior, modeling it using permutation vectors and using the same algorithm as in §4.4, but now using PCID switches instead of memory accesses. In other words, we determine how many PCID switches are required to evict a PCID from the cache, for each PCID position in each permutation vector.

Our results show that the sTLB PCID cache uses a perfect LRU replacement policy. The observations described in §4.4.3 also pertain to the permutation vectors of the PCID cache, although to a lesser extent. As mentioned, we observed inconsistent results on the iTLB, but have enough information to surmise one of two possibilities: either (a) the iTLB and sTLB share a PCID cache, with the iTLB invalidating the entries more eagerly (e.g., when not setting the *NOFLUSH* bit) or (b) the iTLB has its own PCID cache of size 4 with a more complex replacement policy.

4.7 iTLB Partitioning

Following our findings regarding iTLB partitioning between hyperthreads from §4.2 we look into what happens at the moment of transition between non-partitioned and partitioned states. Because partitioning is done set-wise there must be a discrete moment in time when the set hash functions transition from one to the other, evicting entries. We test several hypotheses to how eviction is handled. First and most obvious is to flush the entire iTLB and have subsequent accesses repopulate from sTLB or RAM using the new hash. A slight optimization is to only flush the set indexes that exist solely in the unpartitioned state, and which the new hash function cannot address. A third strategy is to evict enough of the less recently used ways of all sets to make room for the partition.

To test our hypotheses we design an experiment that monitors iTLB behavior and measures what happens on transition. For an iTLB with S sets of W ways we set up two lots, A and B, of $S \times W$ pages each, mapped to fill the entire iTLB. We populate these pages with code that amounts to a jump chain that walks across all sets W times. Additionally, before the jump, we set a bit in a known register identifying the set,

Table 2: Summary of reverse engineered properties on AMD.

TLB Property	Zen+ Ryzen 7 2700X	Zen 3 Ryzen 5 5600X
Inclusive	✗	✗
L1 dTLB		
Number of sets	1	1
Number of ways	64[1]	64[1]
L1 iTLB		
Number of sets	1	1
Number of ways	64[1]	64[1]
L2 dTLB		
Number of sets	256/192[2]	256
Number of ways	8	8
Set selection bits	12–18, 21	12–18, 21
L2 iTLB		
Number of sets	128	128
Number of ways	4[2]	4
Set selection bits	12–17, 21	12–17, 21

[1] Reported by cpuid, but consistent with our results.
[2] Results inconclusive; see §A.2.

way number and chain (A or B) associated with that page. We also add code to the page at the end of the chain that handles recording and looping. This produces a self-contained code blob that fills the iTLB and remains resident.

Running either of these chains by itself will not yield any new information. Thus we prime with chain A, desynchronize to have the page tables point at chain B, flush all entries out of L2, then run chain A again. At the end the measurement registers show, for every entry, whether it has survived unevicted thus far—chain A entries are from the initial priming, chain B entries were reloaded from RAM. We loop over the chain for as long as unevicted entries remain, recording new measurements and the number of iterations between them. This offers us a deep view into how the iTLB behaves over time, allowing us to test our hypotheses.

Results show a high level of interference within the first iterations, with (all) entries rarely remaining in the iTLB for long, even with a completely idle co-resident hyperthread. After the first few evictions the iTLB stabilizes and we see entries with long lifetimes. Activating the co-resident hyperthread yields results not conclusive enough to completely determine the transition strategy, however we can draw some conclusions: first, the iTLB *does not* flush sets in their entirety on transition, rather it seems to flush some of the less recently used ways. Second, regardless of the co-resident hyperthread's activity, a sufficiently often accessed entry can be kept in the iTLB indefinitely, barring any descheduling or interrupts.

4.8 AMD

As pointed out before, our experiments described so far rely on the assumption that the L2 TLB is a shared component for data loads and instruction fetches. When running our experiments on two AMD microarchitectures—Zen+ and Zen 3—we found that this assumption does not hold.

We therefore design an alternative set of experiments. The lack of a shared TLB component has proven challenging, however, and several experiments do not work without this feature. Foremost, we are unable to selectively flush a single TLB level as we describe in §3, which prevents us from distinguishing between L1 and L2 entries. A first consequence of this is that we are unable to design experiments that test for reinsertion behavior and exclusivity. In addition, this limitation, coupled with a fully associative L1 TLB, further prevented us from reliably testing for replacement policies.

However, in spite of these difficulties, we design experiments to determine set size, set mapping and inclusivity. These experiments are explained in Appendix A. Our results are summarized in Table 2.

5 Massaging TLB State

In this section we apply the knowledge about TLB replacement policies gained in §4.4 to show how an attacker can purposefully manipulate the TLB state to their advantage. Specifically, we show how to create access patterns that evict a target entry in fewer accesses than the state of the art.

Previous attacks that construct and use TLB eviction sets [14, 15, 32, 39] do so under the assumption that an optimal eviction set is of size W—the number of ways in a set. Intuitively this makes sense: inserting W new entries clobbers all existing W ways for a large class of replacement policies, thus evicting our target regardless of its position in the set's replacement queue. This strategy is albeit only optimal if one wishes to evict *all* entries from a set and no assumptions can be made about its existing state. In practice, however, both these assumptions are overly restrictive, as an attacker has considerable power to prime the TLB state, and often targets a singular entry in a set.

We show how relaxing these restrictions combined with an accurate model of the replacement policies in use allows us to construct access patterns that evict a particular target entry out of an adequately primed TLB in fewer than W accesses. First we introduce the theory behind constructing these patterns, which we call optimized eviction sets. We then examine two replacement policies, tree-PLRU$_4$ and (MRU+1)$_{\%3}$PLRU$_4$, used on the L1d and L2s of modern Intel CPUs, and construct optimized eviction sets for a most-recently-used target.

5.1 Theoretical Considerations

In order to benefit from knowledge about replacement policies when constructing eviction sets we must first model the TLB state and its evolution through time. Our key insight is treating the possible states of a TLB set as nodes of a graph, with directed edges that represent lookups changing the state.

Table 3: Optimized eviction runs for Tree-PLRU$_4$.

(a) Unknown Starting State

	START	(**T** X X X)
0	π_m	(0 **T** X X)
1	π_m	(1 0 **T** X)
0	π_1	(0 1 X **T**)
2	π_m	(2 0 1 X)

(b) Primed Starting State

	START	(3 2 1 0)
T	π_m	(**T** 3 2 1)
2	π_2	(2 3 **T** 1)
3	π_1	(3 2 1 **T**)
0	π_m	(0 3 2 1)

(c) Eviction Loop Self-Synchronization

	START	(3 2 1 0)	START	(3 2 1 0)
T	π_m	(**T** 3 2 1)	–	(3 2 1 0)
2	π_2	(2 3 **T** 1)	π_1	(2 3 0 1)
3	π_1	(3 2 1 **T**)	π_1	(3 2 1 0)
0	π_m	(0 3 2 1)	π_3	(0 3 2 1)

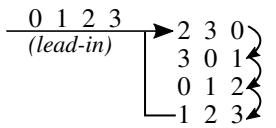


Figure 5: Optimized eviction loop for Tree-PLRU$_4$.

For each TLB set of W ways and replacement policy P we define σ_i—a data structure that exhaustively describes the state of a TLB set, namely the present entries as well as their relative replacement order. We also define Σ_P the set of all possible σ_i states and refer to it as the *state space* of replacement policy P. Σ_P will form the nodes of our graph. Analogously, for each $\sigma_i \in \Sigma_P$ we define outgoing edges to all possible successor states as expressed by P. We remark that outgoing edges imply either a *hit*, of which there are exactly W distinct possibilities, or a *miss*, of which there can be arbitrarily many, bounded only by available memory.

Now that we have a model of TLB behavior, we apply it to the problem of constructing eviction sets. In broad terms, we pick a starting state that contains the eviction target and then apply a graph search algorithm to find the shortest path to our desired end state—one with the target entry evicted. As for the choice of algorithm, we first remark that the raw state graph has a huge number of nodes, each with a similarly huge indegree and outdegree, leading to rapid memory exhaustion for a naive breadth-first-search (BFS) approach.

Fortunately we can make several optimizations to reduce the problem size. For our purpose of constructing an eviction set, we observe that all entries can be categorized in one of three distinct ways: **T**, a singular distinguishable target entry, inaccessible to the attacker; **X**, "don't care", non-distinguishable, inaccessible entries; and $\mathbf{i} \in \mathbb{N}$, numbered, distinguishable entries accessible to the attacker. By only considering these types of entries we greatly reduce the size of our state space. Similarly, if we only consider outgoing edges under the attacker's control (i.e., hits or misses of i-type entries) we further limit the outdegree of our graph. In addition, we remark that when considering outgoing edges that represent a miss, all i-type entries are functionally identical, and can thus be reduced to a single edge. By convention we pick the lowest-numbered entry not currently in the TLB for this purpose. These changes make it practical to apply BFS on the reduced graph, providing insight into massaging TLB states.

5.2 Tree-PLRU$_4$

We apply our previously discussed model to the tree-PLRU$_4$ replacement policy. Thus we have $\sigma_i = (e_1, e_2, e_3, e_4)$ and outgoing edges according to the permutation vectors shown in Figure 1b. At first we consider the case where the attacker has no apriori information about the state of the TLB, other than that the target is the most recently accessed entry. To model this we choose our starting state $\sigma_S = (T, X, X, X)$ and then apply the search algorithm to find the shortest path evicting T. Table 3a shows our results. While our sequence has the same number of lookups as the state of the art, it incurs one fewer TLB miss. According to previous work [14], this translates into marginally fewer CPU cycles.

We can do better, however, if we allow the attacker to prime the TLB set ahead of time. Considering the primed state $\sigma_P = (3, 2, 1, 0)$, a TLB lookup of T will miss and lead to $\sigma_S = (T, 3, 2, 1)$ which we use as starting state. Looking at the resulting minimal path in Table 3b we see a clear improvement over the previous case. Not only are we evicting T using one fewer lookup, we are incurring a TLB miss only when absolutely necessary—when evicting and replacing T. This grants an attacker both faster eviction as well as stealthier operation when compared with traditional eviction sets. In latency terms, we expect a theoretical reduction of CPU cycles of more than 25% [14]. We however observe that the final state differs from the primed state we started with, leading to a future repeat of the eviction run triggering other permutation vectors than intended, ultimately failing to evict T.

To address this issue we extend our eviction sequence with analogous runs, taking note of any repeating patterns. We do this by adding a TLB lookup of T to the previous final state, then we apply the search algorithm again to obtain the next eviction run to add to our sequence and repeat the process. After four such iterations we observe that we have reached a previously visited σ_S, leading to a loop in our sequence and an end to our algorithm. We now have a minimal, repeatable sequence of TLB lookups which will evict any single most-recently-used target, shown in Figure 5 along with the lookups necessary to initially prime the set, which we call the *lead-in*.

There is one more issue we must contend with, namely synchronization. We have thus far constructed our eviction loop relying on a lookup of T to *always* occur in-between eviction

Table 4: Optimized evictions for (MRU+1)$_{\%3}$PLRU$_4$.

(a) Unknown Starting State

	START	(**T** X X X X X X X X X X)
0	π_m	(0 **T** X X X X X X X X X X)
1	π_m	(1 0 **T** X X X X X X X X X)
2	π_m	(2 1 0 **T** X X X X X X X X)
3	π_m	(3 2 1 0 **T** X X X X X X X)
4	π_m	(4 3 2 1 0 **T** X X X X X X)
5	π_m	(5 4 3 2 1 0 **T** X X X X X)
6	π_m	(6 5 4 3 2 1 0 **T** X X X X)
2	π_4	(2 4 6 5 1 3 X X 0 **T** X X)
7	π_m	(7 2 4 6 5 1 3 X X 0 **T** X)
8	π_m	(8 7 2 4 6 5 1 3 X X 0 **T**)
9	π_m	(9 8 7 2 4 6 5 1 3 X X 0)

(b) Primed Starting State (a = 10, b = 11)

	START	(b a 9 8 7 6 5 4 3 2 1 0)
T	π_m	(**T** b a 9 8 7 6 5 4 3 2 1)
6	π_6	(6 a b 9 8 7 **T** 5 4 3 2 1)
9	π_3	(9 b a 6 8 7 3 5 4 **T** 2 1)
8	π_4	(8 a 9 b 7 6 2 4 3 5 1 **T**)
0	π_m	(0 8 a 9 b 7 6 2 4 3 5 1)

runs. That assumption evidently does not hold in a real-world scenario, and starting from a different state will trigger different permutation vectors than expected. We therefore need to investigate how the TLB state evolves when running our loop with and without target lookups. To achieve this we replay the lookups of an eviction run over both the primed state σ_P and our original starting state σ_S and compare resulting states. Examining the results for the first eviction run in Table 3c we observe the very convenient property of self-synchronization. Regardless if T is present in the starting state we see identical final states—the TLB "re-synchronizes" to a properly primed state on every run. This property holds similarly for the other three eviction runs in the loop, meaning that each run will prime the set for the next, regardless of lookups to T.

5.3 (MRU+1)$_{\%3}$PLRU$_4$

Similarly to before, we apply our eviction run construction model on (MRU+1)$_{\%3}$PLRU$_4$. We have $len(\sigma_i) = W = 12$ and outgoing edges correspond to the permutation vectors presented in Figure 3. In addition, because (MRU+1)$_{\%3}$PLRU$_4$ is implemented by an L2 TLB we must either model it together with a replacement policy for L1 or always ensure L1 misses.

We start by looking at the case with no apriori information about the TLB state and obtain the eviction run shown in Table 4a. Similar to our previous results for tree-PLRU$_4$, we obtain a marginally more efficient eviction run—one fewer memory access and two fewer TLB misses, in theory 10% fewer CPU cycles. Once again, allowing the attacker to prime the TLB set dramatically improves the situation, as we can see in Table 4b. Our optimized eviction run requires only one third as many memory accesses, and incurs a single TLB miss, necessary to evict T. This, as before, grants an attacker both greater speed and more difficult detection, with a theoretical latency of less than a third of the state of the art [14]. We also remark that these 4 lookups, by virtue of being L1 misses, also entirely flush a 4-way L1 dTLB.

We notice that not all primed entries are used for eviction, only the ones corresponding to π_6, π_3, and π_4. This means our priming can be limited to just the entries required to trigger these vectors, while also ensuring these positions remain primed for a second run. Taking this idea further, we can extend several π_6-π_3-π_4 eviction runs one after another until we see starting states repeat. This occurs after 6 iterations, giving us the complete eviction loop we see in Figure 6. Analogously to tree-PLRU$_4$, each π_6-π_3-π_4 eviction run following the lead-in is also self-synchronizing, i.e., it reaches identical final states regardless of whether there has been a target lookup.

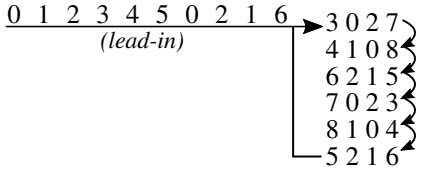


Figure 6: Optimized eviction loop for (MRU+1)$_{\%3}$PLRU$_4$.

6 Case Studies

We now evaluate the impact of optimized TLB evictions on several classes of existing attacks, as well as new attack variants made practical by optimized eviction sets. Specifically, we examine side channels that target the TLB directly [14], as well as various attacks that rely on TLB eviction [15, 32, 33, 39]. We ran our case studies on an Intel i7-7700K (Kaby Lake) CPU with microcode version `0xea` at 4.20 GHz, with frequency scaling disabled, 32 GB DDR4 RAM, and running Linux kernel v5.14.

6.1 Speed Benefits for Existing Attacks

We first examine the performance improvements that optimized eviction sets can bring to existing attacks.

Page Translation Side Channels In our first case study, we look at the class of side channels that rely on measuring the side effects of page translation and feature TLB eviction as a supporting mechanism for triggering page table walks. Here the TLB evictions are not at the heart of the attack, but still play an important role in the overall overhead. Examples in-

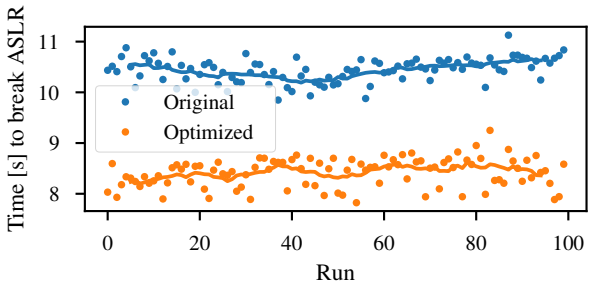


Figure 7: Comparison of time required to break ASLR.

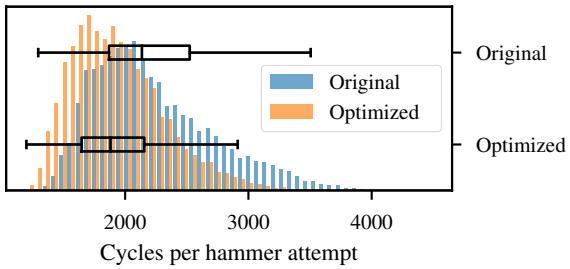


Figure 8: Comparison of time cost per double-sided hammer.

clude AnC [15], RevAnC [33], and XLATE [32]. We examine AnC as a representative of this class.

Starting with the native code implementation [1] as a baseline we patched it to use optimized TLB eviction sets. Because the purpose is to evict a target entry from the TLB entirely, we used the L2 eviction loop from §5.3. Figure 7 plots the time needed to break ASLR for both the original AnC implementation and our patched version. We see a clear improvement for the latter, taking on average 20% less time to break ASLR than naive eviction sets. These results show that optimized TLB evictions provide tangible improvements even to side channel attacks in which they serve a supporting role.

Page Translation Rowhammer In our second case study, we look at Rowhammer-based page translation attacks which, again, feature TLB eviction as a supporting mechanism for triggering page table walks. The recent PTHammer [39] technique exemplifies this class of attack. Rowhammer is a vulnerability in modern DRAM manifesting as bits in one row flipping due to the leakage of electrical charge, caused by rapid repeated accesses—hammering—of neighboring rows. The hammer rate must be sufficiently high to allow for enough charge to leak before the row is refreshed. Furthermore, the PTHammer authors show that the relationship between hammer rate and bit flip frequency is strongly non-linear, making even marginal gains potentially very significant.

[1] https://github.com/vusec/revanc

Table 5: Raw sample rates (M/sec) of TLB-based channels.

		Naive	Optimized	diff
L1d	clear	50.5	51.4	+1.8%
	noise	34.4	41.3	+20%
L2	clear	14.1	33.4	2.37×
	noise	13.5	25.5	1.89×
L1d+L2	clear	3.5	18.18	5.19×
	noise	3.07	14.28	4.65×

We initially attempted to reimplement PTHammer [39], reproduce their results, and show the improvements of optimized eviction sets on concrete bit flips. This attempt proved unsuccessful on modern hardware, and we were unable to acquire a hardware setup similar to what the authors used due to its age (circa 2011). As an alternative, we implemented a benchmark using the same techniques as the authors, which we used to compare the effect of naive and optimized eviction sets on the hammer rate.

Looking at the results, the hammer time falls into three distributions, with only the fastest evidencing differences between naive and optimized evictions. We hypothesize this to be caused by stalls on the ring interconnect and surmise that the slower rates are insufficient to cause bit flips. Thus, we examine the faster distribution in detail in Figure 8. We see a roughly 12% shorter median hammer time for optimized vs. naive eviction sets. Again, these results show that optimized TLB evictions provide tangible improvements even to Rowhammer attacks in which they serve a supporting role.

6.2 Direct TLB Attacks

We now examine attacks that directly feature TLB lookups as a side channel, as exemplified by the TLBleed attack introduced by Gras et al. [14]. Specifically, we highlight how optimized eviction sets improve the sampling performance of TLBleed-style attacks, both enhancing the original variant, as well as practically enabling additional variants. We discuss the benefits of these new variants and why they are relevant for potential attackers. Finally, we conclude by quantifying the benefit of optimized evictions in covert channel scenarios.

TLBleed Sampling Performance Using similar techniques as described by Gras et al., we implement a TLBleed-style side channel using both naive and optimized eviction sets and measure its sample rate. We focus on two scenarios: *clear*, when the other logical core is idle, and *noise*, when a co-resident thread randomly accesses memory. Our results are summarized in Table 5.

Compared to the original attack on L1d, we implement the eviction loop discussed in §5.2. While the results show only a small improvement on a clear channel, the difference is much

more pronounced when the channel is actively used, where we see the original implementation requiring as many as 4 misses, compared to just one for our optimized eviction sets.

We adapt our experiment from before to target the L2 sTLB, implementing the eviction loops discussed in §5.3. We note that the original TLBleed paper did not consider attacks based on L2 sTLB due to the fairly poor performance, as evident from Table 5. The improvements due to optimized eviction sets for L2 are more evident than for L1, showing roughly 2× improvement in the sample rate for both clear and noisy channels. Indeed, the absolute rates measured here for L2 approach those presented for L1d in TLBleed [14], providing sufficiently fine temporal granularity to enable similar attacks, but now on L2, yielding a practical, finer-grained variant of the original TLBleed attack.

Set-pair TLBleed We observe in Table 1 that newer Intel microarchitectures exhibit co-prime set selection functions for L1d and L2. Under these circumstances, a further possible TLB side channel can be constructed over a particular pair of L1d and L2 sets by timing the eviction of each level independently. Doing so naively is laborious, as we lay out in Appendix B, but we can use our knowledge of replacement policies to construct more practical eviction sets. We remark that each step of our eviction loop from §5.3 entirely fills L1d, therefore repeating it will always hit L1. Any new access will evict an entry, causing misses that hit L2 with a discernible delay. Conveniently, repeating a loop step is self-synchronizing, as we show in detail in Appendix C. Having independently measured L1 usage, we time the next step of the eviction loop to measure L2, similar to the previous paragraph. We consider our victim set pair to have been accessed only when both measurements agree, considering it noise otherwise.

The performance improvements of optimized eviction sets are most evident for this technique, with up to 5× the rate of naive, as shown in Table 5.

Spatial Discrimination and Set Count When running TLBleed attacks over L2 or a pair of L1 and L2 sets, we gain a finer target selection due to the larger number of sets, or of combinations of sets. This better spatial discrimination gives an attacker greater confidence that a positive measurement has actually been caused by a target access, as random noise is less likely to perturb the TLB state in the same way.

Existing exploits that target either the LLC [13] or the TLB [14] report higher levels of noise on some sets that is due to aliasing—undesired (secret-independent) victim accesses that map to the same set as the target secret. Better spatial discrimination reduces aliasing, and therefore aliasing noise, in a way that is directly proportional to the number of sets. Monitoring L2 would thus provide roughly an order of magnitude less aliasing noise (8 times less), with L1+L2 reducing it further by an additional order of magnitude, to 128 times less than original L1d TLBleed.

Furthermore, having more sets to work with has an additional benefit. Optimized covert channel implementations built on top of cache contention require a significant number of auxiliary sets for unidirectional communication. The PHY implementation proposed by Maurice et al. [26], for example, requires 26 sets, with 14 sets for sequencing and synchronization alone, and would not fit within the number of L1 TLB sets. While serial designs, such as the one by Liu et al. [24], address synchronization differently and thus use a minimal number of sets, they do so at the expense of performance, requiring multiple set evictions for every bit. As such, an eviction primitive offering similar sample rates for a much larger number of sets enables state-of-the-art covert channel implementations for the TLB.

Covert Channel In order to quantify the benefits of optimized eviction sets beyond raw sample rates, we implemented a proof-of-concept covert channel over L2 in a sandbox setting, based on the original TLBleed implementation [14]. Compared to the original implementation, which used eviction to both send and receive a bit, we send by using a single memory access to each set of interest. This not only significantly improves baseline performance, but also enables us to use the eviction loops discussed in §5.3 as a receive primitive.

We measured the bandwidth of our channel while transmitting 100 000 words of 32 bits—16 for payload, 8 for sequencing, and 8 for error detection. Using naive evictions, we observe an average effective bandwidth of 5.6 MBit/s, peaking at 6.2 MBit/s. Meanwhile, optimized evictions measured 11 MBit/s on average, with a maximum of 11.7 MBit/s, for a relative improvement of nearly 2× on average and 89% peak, which mirrors the sample rate improvements seen in Table 5. The undetected error rate is below $2 \cdot 10^{-5}$ for both naive and optimized eviction sets, with at most one word being erroneous—in line with the original TLBleed implementation.

7 Related Work

Having laid out attacks that target the TLB in §2.2 and §6, we now focus on related work that focuses on manipulating the TLB state for benign purposes, as well as more general work on reverse engineering and exploiting the cache subsystem. We also address the implications of detailed knowledge of replacement policies on secure TLB designs.

Manipulating TLB State Since address translation represents a considerable proportion of memory access latency, monitoring and controlling the TLB state has been the focus of much prior work concerned with real-time systems [20,28], where preserving bounded latency guarantees is crucial. Furthermore, the systems security community has previously explored using TLB entries desynchronized from their in-memory PTEs to implement memory protections such as non-

executable [31] and execute-only memory [12] in software. In addition, TLB desynchronization has been previously used in adversarial settings to hide rootkits from detection [30] or defeat checksumming-based software tamperproofing [35].

Reverse Engineering Cache Subsystems While cache and TLB reverse engineering efforts often go hand in hand with exploit development [14, 15, 17, 22, 32, 33] or are prompted by the need to improve existing attacks [25], there is literature on more benign applications. Abel and Reineke [1, 2] introduced permutation vectors and reverse engineered many CPU cache properties, among which replacement policies, for the purpose of developing better software optimizations and cycle-accurate simulators. A master thesis by Zhu [40] reported hash functions and inclusion policy for the Intel Skylake TLB, but mistakenly reported the replacement policies of the dTLB and sTLB as LRU.

Exploiting Data Caches Cache attacks have a long tradition in the systems security literature, using cache set contention as a side channel and often targeting cryptographic algorithms [10, 18, 24, 29, 37]. Guanciale et al. [16] showed that caches can also be purposely desynchronized to create a so-called cache storage channel, serving as an alternative to timing-based attacks. Additionally, more recent attacks [27, 38] have explored other microarchitectural side channels in order to increase granularity beyond cache line size. Finally, cache attacks such as FLUSH+RELOAD [37] have been recently repurposed as convenient covert channels to leak information in the context of Spectre and other transient execution attacks [21].

Secure TLBs To address the various attacks on the TLB, Deng et al. [7] have proposed secure designs that either partition the TLB (SP-TLB) or introduce randomness in TLB loads (RF-TLB). SP-TLB would retain the same security properties whether or not the attacker knows the replacement policies in use. Indeed, any (statically or dynamically) partitioned TLB will successfully prevent an attacker from adversarially evicting a victim entry, as we have seen with some Intel iTLBs.

The property offered by RF-TLB of decorrelating TLB states from access patterns in the secure region remains unaffected by the attacker's knowledge of replacement policies. However, as noted by the authors, practical implementations should not randomize all TLB accesses to limit the performance impact. If the randomization frequency chosen by a practical implementation can be overcome by an attacker with a sufficient number of repetitions, then our eviction sets can, in principle, make the attack more efficient. However, since randomized replacement entries can fall outside of the targeted set, an attacker would have a harder time implementing optimized eviction sets on RF-TLB.

8 Conclusion

In this paper, we introduced *TLB desynchronization* as a novel way of reverse engineering previously inscrutable TLB properties. This allowed us to expose behavior previously undocumented and we subsequently used these new insights to better understand—and thus manipulate—the TLB, constructing optimized eviction sets. We showed how this new eviction primitive improves existing TLB-based attacks. Finally, we posit that this work is a stepping stone towards gaining a more complete model of the dynamic behavior of the memory subsystem of modern CPUs, knowledge that is crucial not only to developing more effective attacks, but for constructing more informed defenses as well.

Acknowledgements

We thank our shepherd, Michael Schwarz, and the anonymous reviewers for their comments. We also thank Ben Gras for helping with the TLBleed covert channel implementation. This work was supported by the EU's Horizon 2020 research and innovation programme under grant agreement No. 825377 (UNICORE), Intel Corporation through the Side Channel Vulnerability ISRA, and NWO through projects "TROPICS", "Theseus", and "Intersect".

References

[1] Andreas Abel and Jan Reineke. Measurement-based modeling of the cache replacement policy. In *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 65–74. IEEE, 2013.

[2] Andreas Abel and Jan Reineke. nanobench: A low-overhead tool for running microbenchmarks on x86 systems. In *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 34–46. IEEE, 2020.

[3] Hussein Al-Zoubi, Aleksandar Milenkovic, and Milena Milenkovic. Performance evaluation of cache replacement policies for the spec cpu2000 benchmark suite. In *Proceedings of the 42nd annual Southeast regional conference*, pages 267–272, 2004.

[4] Zelalem Birhanu Aweke, Salessawi Ferede Yitbarek, Rui Qiao, Reetuparna Das, Matthew Hicks, Yossi Oren, and Todd Austin. Anvil: Software-based protection against next-generation rowhammer attacks. *ACM SIGPLAN Notices*, 51(4):743–755, 2016.

[5] Adi Basel, Gur Hildesheim, Shlomo Raikin, Robert Chappell, Ho-Seop Kim, and Rohit Bhatia. Balanced

P-LRU tree for a "multiple of 3" number of ways cache, 2016. US Patent 9,348,766.

[6] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[7] Shuwen Deng, Wenjie Xiong, and Jakub Szefer. Secure tlbs. In *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, pages 346–359. IEEE, 2019.

[8] Advanced Micro Devices. *Software Optimization Guide for AMD EPYCTM 7003 Processors*. AMD, November 2020.

[9] Advanced Micro Devices. *Software Optimization Guide for AMD Family 17h Models 30h and Greater Processors*. AMD, March 2020.

[10] Daniel Genkin, Lev Pachmanov, Eran Tromer, and Yuval Yarom. Drive-by key-extraction cache attacks from portable code. In *International Conference on Applied Cryptography and Network Security*, pages 83–102. Springer, 2018.

[11] Robert T George, Jason W Brandt, Jonathan D Combs, Peter J Ruscito, and Sanjoy K Mondal. Associating address space identifiers with active contexts, June 23 2009. US Patent 7,552,254.

[12] Jason Gionta, William Enck, and Peng Ning. Hidem: Protecting the contents of userspace memory in the face of disclosure vulnerabilities. In *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*, pages 325–336, 2015.

[13] Enes Goktas, Kaveh Razavi, Georgios Portokalidis, Herbert Bos, and Cristiano Giuffrida. Speculative Probing: Hacking Blind in the Spectre Era. In *CCS*, November 2020. Pwnie Award for Most Innovative Research.

[14] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks. In *USENIX Security*, August 2018. Pwnie Award Nomination for Most Innovative Research.

[15] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. ASLR on the Line: Practical Cache Attacks on the MMU. In *NDSS*, February 2017. Pwnie Award for Most Innovative Research, DCSR Paper Award.

[16] Roberto Guanciale, Hamed Nemati, Christoph Baumann, and Mads Dam. Cache storage channels: Alias-driven attacks and verified countermeasures. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 38–55. IEEE, 2016.

[17] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical timing side channel attacks against kernel space aslr. In *2013 IEEE Symposium on Security and Privacy*, pages 191–205. IEEE, 2013.

[18] Mehmet Sinan Inci, Berk Gulmezoglu, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Cache attacks enable bulk key recovery on the cloud. In *International Conference on Cryptographic Hardware and Embedded Systems*, pages 368–388. Springer, 2016.

[19] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4*. Intel Corporation, June 2021.

[20] Takuya Ishikawa, Toshikazu Kato, Shinya Honda, and Hiroaki Takada. Investigation and improvement on the impact of tlb misses in real-time systems. *Proc. of OSPERT*, 2013.

[21] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1–19. IEEE, 2019.

[22] Jakob Koschel, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. TagBleed: Breaking KASLR on the Isolated Kernel Address Space Using Tagged TLBs. In *EuroS&P*, September 2020.

[23] 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 *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[24] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. Last-level cache side-channel attacks are practical. In *2015 IEEE symposium on security and privacy*, pages 605–622. IEEE, 2015.

[25] Clémentine Maurice, Nicolas Le Scouarnec, Christoph Neumann, Olivier Heen, and Aurélien Francillon. Reverse engineering intel last-level cache complex addressing using performance counters. In *International Symposium on Recent Advances in Intrusion Detection*, pages 48–65. Springer, 2015.

[26] 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 *NDSS*, volume 17, pages 8–11, 2017.

[27] Ahmad Moghimi, Jan Wichelmann, Thomas Eisenbarth, and Berk Sunar. Memjam: A false dependency attack against constant-time crypto implementations. *International Journal of Parallel Programming*, 47(4):538–570, 2019.

[28] Shrinivas Anand Panchamukhi and Frank Mueller. Providing task isolation via tlb coloring. In *21st IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 3–13. IEEE, 2015.

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

[30] Sherri Sparks and Jamie Butler. Shadow walker: Raising the bar for rootkit detection. *Black Hat Japan*, 11(63):504–533, 2005.

[31] The PaX Team. paging based non-executable pages. https://pax.grsecurity.net/docs/pageexec.txt, 2003. Accessed 2020-12-03.

[32] Stephan van Schaik, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Malicious Management Unit: Why Stopping Cache Attacks in Software is Harder Than You Think. In *USENIX Security*, August 2018.

[33] Stephan van Schaik, Kaveh Razavi, Ben Gras, Herbert Bos, and Cristiano Giuffrida. RevAnC: A Framework for Reverse Engineering Hardware Page Table Caches. In *EuroSec*, April 2017.

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

[35] Glenn Wurster, Paul C Van Oorschot, and Anil Somayaji. A generic attack on checksumming-based software tamper resistance. In *2005 IEEE Symposium on Security and Privacy (S&P'05)*, pages 127–138. IEEE, 2005.

[36] Wenjie Xiong and Jakub Szefer. Leaking information through cache LRU states. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 139–152. IEEE, 2020.

[37] Yuval Yarom and Katrina Falkner. Flush+ reload: A high resolution, low noise, l3 cache side-channel attack. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 719–732, 2014.

[38] Yuval Yarom, Daniel Genkin, and Nadia Heninger. CacheBleed: a timing attack on OpenSSL constant-time RSA. *Journal of Cryptographic Engineering*, 7(2):99–112, 2017.

[39] Zhi Zhang, Yueqiang Cheng, Dongxi Liu, Surya Nepal, Zhi Wang, and Yuval Yarom. Pthammer: Cross-user-kernel-boundary rowhammer through implicit accesses. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 28–41. IEEE, 2020.

[40] Weixi Zhu. Exploring superpage promotion policies for efficient address translation. *Master's Thesis, Rice University*, 2019.

A Reverse Engineering AMD

In this section we lay out a set of experiments that do not rely on a shared L2 TLB. Specifically, we discuss testing for the existence of a shared TLB component, determining set size and mapping, as well as testing for inclusivity.

Neither tested microarchitectures support PCIDs, therefore we cannot run our PCID experiments discussed in §4.5.

A.1 Shared TLB

To verify whether a TLB has a shared component at all, we check whether an entry primed by a data load can eventually be used to translate an instruction fetch, or the other way around. Specifically, we access an arbitrarily chosen page A using one access type, followed by accessing n other pages using the same access type. We then de-synchronize the TLB with the in-memory page tables, and we probe A again using the opposite access. We perform this experiment in multiple iterations, for different values of $0 \leq n < 5000$. In case of a shared TLB component, we would expect a hit for some values of n. In case of full seperation between the iTLB and dTLB, we would not expect any hits.

Our results show almost always misses, with hits occurring in less than 0.1% of cases. We however do not consider this proof of a shared TLB component, due to several observations that exclude this possibility:

- Hits only occur when an instruction fetch is followed by a data load, never the other way around.
- There is no correlation between the occurrence of hits and the value of n.
- The amount of hits reduces when we insert *cpuid* instructions before and after each memory access, leading us to suspect speculative execution.

We hypothesise that any spurious hits are caused by (speculative) TLB prefetching. According to the AMD architecture manual [8, 9], the hardware page table walker can perform speculative translation, supporting our hypothesis. Thus we conclude that the TLB is fully separated on both levels.

A.2 Set Mapping and Set Size

We now design an experiment for determining the set size and mapping that does not rely on a shared L2 TLB. We assume that the bits used for set indexing are a subset of the 20 least

significant bits of a virtual page number (bits 12–31 of the virtual address). This is reasonable because we expect the TLB to enumerate all sets even with only 4 GiB of virtual memory (e.g., when running 32-bit code).

We start by picking m—a guess for the number of ways—and an arbitrary 20 bit number X, then assemble a lot of m pages with bits 12 to 31 set to this value. In addition, we pick a bit b, $12 \leq b \leq 31$, that we flip in X, obtaining Y, and assemble an eviction lot of M pages with bits 12 to 31 set to Y, with M chosen to be significantly larger than the sum of any reasonable L1 and L2 set capacity. If bit b is used by the set hash function, the eviction lot would end up in a different set. However, if bit b is irrelevant for set selection, we expect both lots of pages to land in the same set, with the second one evicting (some of) the first. Therefore, we prime the TLB with the first lot of m pages then desynchronize their PTEs. Next we access the eviction lot 10 times, then test if the m pages are still in the TLB. We repeat this for a large number of iterations for various values of m and b. If we do not consistently witness evictions of any of the m pages, we conclude that bit b is used for set selection. If we consistently see misses, we can deduce that bit b is irrelevant. Additionally, if we consistently see misses regardless of the value of b we can deduce that m is larger than the true number of ways.

Results appear inconclusive at first, as performing this experiment does not always yield consistent results. However, when filtering out the runs that are either noisy or exhibit extremely unlikely properties (e.g., odd number of ways or entries that are never evicted from the TLB) we are able to detect a pattern. We consistently witness dTLB misses when $m > 8$, regardless of the bit flipped. Analogously for the iTLB, we see misses when $m > 4$. For the dTLB, bits 12–18 and 21 of the virtual address are involved with set indexing, while the iTLB uses bits 12–17 and 21.

Furthermore, the fact that we consistently measure misses for $m > 8$ (dTLB) and $m > 4$ (iTLB), regardless of the chosen b, leads us to the conclusion that L1 has a single set, i.e., is fully associative. If that were not the case, we would find a value of b for which the M eviction pages would fall in a different L1 set, thus not evicting the original m entries.

Given fully-associate L1 TLBs, we deduce the L2 dTLB to have 256 sets, each 8-way associative. Similarly, we infer L2i to have 128 sets, each 4-way associative. In the case of Zen+ this contradicts the number of dTLB sets and the iTLB number of ways as reported by `cpuid`, which is 192 and 8 respectively. When looking closer at the results we do notice that flipping bit 21 only preserves all m entries when m is much smaller than W_{L2d}. This leads us to suspect the hash function to be more complex than linear bit slicing, although we have not been able to reverse engineer it.

Finally, we find the sizes of L1d and L1i as reported by `cpuid` to be consistent with our experiments, as picking M significantly lower than 64 does not show any misses.

The fact that the L1 TLBs are fully associative could be a factor in the unreliable results of the experiment to detect set selection bits. Keeping a perfect LRU replacement ordering with such a large number of ways is clearly infeasible in hardware, therefore L1 most likely implements a replacement policy with fewer states (e.g. tree-PLRU). We hypothesize that, due to unavoidable L1 hits induced by our experiment code, some of our m pages "jump ahead" in the replacement queue and are prevented from being evicted by the eviction set. This unavoidable amount of noise caused by a fully associative L1 also makes precisely measuring the number of ways or the replacement policy of L1 particularly challenging.

A.3 Inclusivity

As a final experiment, we test whether the TLB is inclusive. We remark that a strictly inclusive TLB would necessarily admit at most W_{L2} entries that map to the same L2 set. Hence, we can test inclusivity by first priming the TLB with $W_{L2}+1$ entries mapping to the same L2 set, desynchronizing after every access, then probing whether the entries are still present.

The results consistently show $W_{L2}+1$ TLB hits, proving both iTLB and dTLB to be non-inclusive.

B Performing Set-Pair TLBleed

Separately measuring an adversarial eviction of a target address in both levels of the TLB is not as straight-forward as single-level TLBleed attacks. In this section we will present one possible way of doing this for a two-level TLB of W_{L1} and W_{L2} ways, making minimal assumptions about its behavior.

We put no constraints on the replacement policies in use, except that W_{Lx} misses completely flushes the set and primes it with the new entries. Furthermore, the TLB in question must be non-inclusive and non-exclusive, have fewer ways in L1 than in L2, and have hash functions that are mutually prime, i.e., an address can be assigned all possible (L1, L2) set pairs. We assume an attacker knows the number of sets and ways for both levels of the TLB, as well as the hash functions.

We start by allocating two lots of W_{L2} distinct addresses each that will be used in alternation. At each step, we prime L1 and L2 with entries from one lot and wait for a possible victim access. Next, when we wish to probe, we first go through our last-accessed W_{L1} entries in the same order as before, timing the accesses. We expect all L1 hits if there was no intervening L1 access, and one or more misses otherwise.

In the second phase, we walk through all W_{L2} entries of our current lot in order. If there has been no intervening activity on any level we expect all hits once again, since we had primed the entire set and the L1 phase doesn't touch L2. Similarly, if there has only been activity on L1, the previous phase would not have evicted any L2 entry, by virtue of all L1 misses being already primed in L2. Finally, only when there has been an access to the L2 set do we expect *at least* one miss—required

to evict the new entry—with more possible depending on the actual replacement policy in use.

We now have two separate measurements of whether each level has been accessed. We consider there to have been an access to our target (L1, L2) set pair only if *both* phases report misses. In any other case we classify the measurement as either quiet or noise caused by accesses outside our interest.

When moving on to the next step, we use the other lot of addresses, to ensure fresh priming of the TLB.

C State Evolution of Set-Pair TLBleed

In Table 6 we see in detail how the TLB state evolves in one step of the optimized set-pair eviction loop introduced in §6.2. We treat four cases: (a) when no access occurs to either set, (b) and (c) when sporadic accesses happen on L1 and L2 respectively, and (d) when our actual target pair of sets has been accessed. Worth noting is that cases (a) and (b) have identical intermediate states, showing that the L1 phase does not affect the state of L2, regardless of L1 access. Additionally, we see that the final states are identical across all cases—the eviction loop is self-synchronizing, regardless of target accesses.

Table 6: Optimized set-pair TLBleed

(a) No access: 4 L_1 hits and 0 L_2 misses

	START	(8 0 1 4) (8 0 2 1 7 6 4 X X X 3 5)
4	$L_1\pi_3$	(4 8 0 1) (8 0 2 1 7 6 4 X X X 3 5)
1	$L_1\pi_3$	(1 4 8 0) (8 0 2 1 7 6 4 X X X 3 5)
0	$L_1\pi_3$	(0 1 4 8) (8 0 2 1 7 6 4 X X X 3 5)
8	$L_1\pi_3$	(8 0 1 4) (8 0 2 1 7 6 4 X X X 3 5)
6	$L_1\pi_m$ $L_2\pi_5$	(6 8 0 1) (6 8 0 2 1 7 5 4 X X X 3)
2	$L_1\pi_m$ $L_2\pi_3$	(2 6 8 0) (2 8 0 6 1 7 X 4 X 5 X 3)
1	$L_1\pi_m$ $L_2\pi_4$	(1 2 6 8) (1 0 2 8 7 6 X X X 4 3 5)
5	$L_1\pi_m$ $L_2\pi_{11}$	(5 1 2 6) (5 1 0 2 8 7 6 X X X 4 3)

(b) L1 access only: 4 L_1 misses and 0 L_2 misses

	START	(**T** 8 0 1) (8 0 2 1 7 6 4 X X X 3 5)
4	$L_1\pi_m$ $L_2\pi_6$	(4 **T** 8 0) (4 0 2 1 7 6 8 X X X 3 5)
1	$L_1\pi_m$ $L_2\pi_3$	(1 4 **T** 8) (1 0 2 4 7 6 X X X 8 3 5)
0	$L_1\pi_m$ $L_2\pi_1$	(0 1 4 **T**) (0 2 1 7 6 4 X X X 3 5 8)
8	$L_1\pi_m$ $L_2\pi_{11}$	(8 0 1 4) (8 0 2 1 7 6 4 X X X 3 5)
6	$L_1\pi_m$ $L_2\pi_5$	(6 8 0 1) (6 8 0 2 1 7 5 4 X X X 3)
2	$L_1\pi_m$ $L_2\pi_3$	(2 6 8 0) (2 8 0 6 1 7 X 4 X 5 X 3)
1	$L_1\pi_m$ $L_2\pi_4$	(1 2 6 8) (1 0 2 8 7 6 X X X 4 3 5)
5	$L_1\pi_m$ $L_2\pi_{11}$	(5 1 2 6) (5 1 0 2 8 7 6 X X X 4 3)

(c) L2 access only: 4 L_1 hits and 1 L_2 miss

	START	(8 0 1 4) (**T** 8 0 2 1 7 6 4 X X X 3)
4	$L_1\pi_3$	(4 8 0 1) (**T** 8 0 2 1 7 6 4 X X X 3)
1	$L_1\pi_3$	(1 4 8 0) (**T** 8 0 2 1 7 6 4 X X X 3)
0	$L_1\pi_3$	(0 1 4 8) (**T** 8 0 2 1 7 6 4 X X X 3)
8	$L_1\pi_3$	(8 0 1 4) (**T** 8 0 2 1 7 6 4 X X X 3)
6	$L_1\pi_m$ $L_2\pi_6$	(6 8 0 1) (6 8 0 2 1 7 **T** 4 X X X 3)
2	$L_1\pi_m$ $L_2\pi_3$	(2 6 8 0) (2 8 0 6 1 7 X 4 X **T** X 3)
1	$L_1\pi_m$ $L_2\pi_4$	(1 2 6 8) (1 0 2 8 7 6 X X X 4 3 **T**)
5	$L_1\pi_m$ $L_2\pi_m$	(5 1 2 6) (5 1 0 2 8 7 6 X X X 4 3)

(d) L1+L2 access: 4 L_1 misses and 1 L_2 miss

	START	(**T** 8 0 1) (**T** 8 0 2 1 7 6 4 X X X 3)
4	$L_1\pi_m$ $L_2\pi_7$	(4 **T** 8 0) (4 0 **T** 1 7 2 8 X 6 X 3 X)
1	$L_1\pi_m$ $L_2\pi_3$	(1 4 **T** 8) (1 0 **T** 1 4 2 X X 6 8 3 X)
0	$L_1\pi_m$ $L_2\pi_1$	(0 1 4 **T**) (0 **T** 1 7 2 4 X 6 X 3 X 8)
8	$L_1\pi_m$ $L_2\pi_{11}$	(8 0 1 4) (8 0 **T** 1 7 2 4 X 6 X 3 X)
6	$L_1\pi_m$ $L_2\pi_8$	(6 8 0 1) (6 8 0 2 1 7 **T** 4 X X X 3)
2	$L_1\pi_m$ $L_2\pi_3$	(2 6 8 0) (2 8 0 6 1 7 X 4 X **T** X 3)
1	$L_1\pi_m$ $L_2\pi_4$	(1 2 6 8) (1 0 2 8 7 6 X X X 4 3 **T**)
5	$L_1\pi_m$ $L_2\pi_m$	(5 1 2 6) (5 1 0 2 8 7 6 X X X 4 3)