Reverse-Engineering the Address Translation Caches

Philipp Ertmer, Robert Dumitru(✉), and Yuval Yarom

Ruhr University Bochum, Bochum, Germany
{philipp.ertmer,robert.dumitru,yuval.yarom}@rub.de

Abstract. The address translation process and the responsible memory management unit (MMU) in modern CPUs have been the subject of multiple recent microarchitectural side-channel attacks. A precondition to many of these attacks is familiarity with the intimate details of the microarchitectural implementation of the process. However, because vendors do not typically publish extensive information on this, attackers must resort to reverse engineering techniques. Indeed, past works have investigated such techniques, providing insights and novel understanding on the implementation of components used in the address translation process.

In this work, we improve this understanding. We extend the cache desynchronization technique of Tatar et al., and apply it to the page translation caches, which store partial address translation information. We develop automated tooling for investigating five generations of Intel processors, ranging from Haswell to Alder Lake. Our investigations correct mistakes in prior publications, identify a cache level that was missed so far, and discover two hitherto unknown replacement policies. This new understanding of address translation can increase attack precision and facilitate better address-translation-based attacks.

1 Introduction

Microarchitectural side-channel attacks have emerged as a significant threat in the field of cyber security. Such attacks have been demonstrated to leak cryptographic keys [11,12,24,34,38], break address space layout randomization (ASLR) [10,13,15,17,41], and even compromise entire systems [7,19,20,29,33,39]. As many of these techniques exploit the CPU data and instruction caches to infer sensitive information, many software- and hardware-based countermeasures have been proposed to protect these components from side-channel attacks [8,14,21,31,42]. In turn, several follow-up works have demonstrated limitations in these countermeasures, because they fail to consider microarchitectural components other than memory caches [3,12,25,27,34,37,40].

To stay ahead in this ongoing challenge, attackers and defenders need to inform themselves about the underlying microarchitecture. One possibility is to consult the official resources provided by CPU vendors [6,18]. However, these are often vague or incomplete due to their proprietary nature. Thus, researchers

M. Egele et al. (Eds.): DIMVA 2025, LNCS 15747, pp. 149–168, 2025.
https://doi.org/10.1007/978-3-031-97620-9_9

often must resort to reverse engineering in order to recover the implementation details of targeted components.

One of the components that has recently garnered attention is the memory management unit (MMU), which is responsible for translating virtual memory addresses to the corresponding physical memory addresses. Two notable works in this direction are those of Gras et al. [12] and Tatar et al. [32], which reverse-engineer the translation lookaside buffer (TLB), a cache implemented by the MMU. Their insights enable attackers to bypass countermeasures against microarchitectural attacks [12,34,39] and efficiently execute attacks targeting the MMU [32].

However, several other components that implement address translation in modern CPUs have attracted much less attention. In particular, the MMU includes *translation caches*, which cache partial results of address translations. While these play a critical role in the address translation process and for many attacks targeting the MMU [13,34,41], their pertinent implementation details are still largely undocumented. Van Schaik et al. [35] investigated these structures, but the information they provide is partial, and has not been corroborated by independent studies. Understanding the exact structure and behavior of these components is crucial for finding potential vulnerabilities as well as for implementing effective defenses.

Contributions

In this work we close the gap and improve the overall understanding of the MMU. We adapt the TLB desynchronization approach of Tatar et al. [32] and apply it to reverse-engineer translation caches. We build an automated tool called *Talbot*, for reverse engineering translation caches,[1] which we use to evaluate six different Intel microarchitectures. Talbot recovers multiple properties of translation caches, including their size, hash function, replacement policy, and more.

Our adapted desynchronization approach allows us to reduce noise and produce more precise and consistent results than were previously possible. With this increased precision, we correct some minor mistakes in the information published in past works and identify that, contrary to past publications, the Skylake microarchitecture does feature four levels of translation caches.

Additionally, our reverse-engineering effort identifies two unpublished replacement policies used on Intel processors. The first, which we term *hit-updated PLRU*, is a variant of tree-based PLRU where the tree structure is only updated on cache hits but not on cache misses. Consequently, entries that are not used tend to be quickly replaced. The second, which we term *most-recently hit*, is a variant of the most-recently used policy where the entry that experienced the most recent hit is the first candidate for replacement.

In summary, this work makes the following contributions:

- We adapt TLB desynchronization for the purpose of reverse-engineering translation caches (Sect. 3).

[1] Talbot is open-source, available at https://github.com/0xADE1A1DE/Talbot.

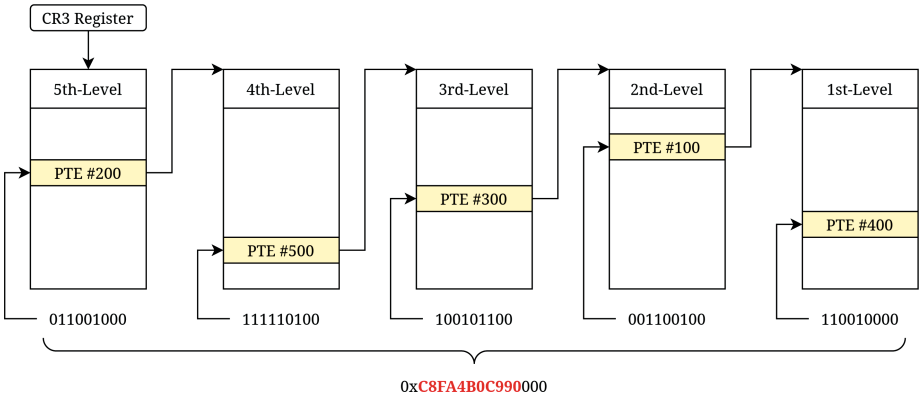

Fig. 1. The page table walk performed by a MMU to translate virtual address `0xC8FA4B0C990000` to its corresponding physical page. The most-significant 9-bit slice is used to index PTE number 200 in the fifth-level page table, which points to the fourth-level page table. The next 9-bit slice is then used to index entry 500 in the fourth-level page table, and so forth.

- We design and implement Talbot, an automated tool for investigating translation caches (Sect. 5).
- Using Talbot, we reverse-engineer the translation caches of six different Intel microarchitectures, identifying new structures and replacement policies (Sect. 4).

2 Background

In this section, we provide background on address translation, cache-memory desynchronization, and cache replacement policies.

2.1 Address Translation

Modern processors use memory virtualization to simplify and optimize system memory management. This abstraction presents each process running on a machine with its own dedicated, contiguous blocks of virtual address space to operate on. Such functionality is made possible by dynamic translation of virtual addresses to physical addresses. The memory management unit (MMU) is the dedicated hardware component responsible for managing these translations.

The high level translation process is generally well documented by CPU manufacturers. Translation information is stored in main memory in the form of *page tables*, which are structured as a multilevel directed tree. Each page table is indexed by a specific portion of the virtual address. A page table entry (PTE) points to either the next (lower) level page table or the requested physical page.

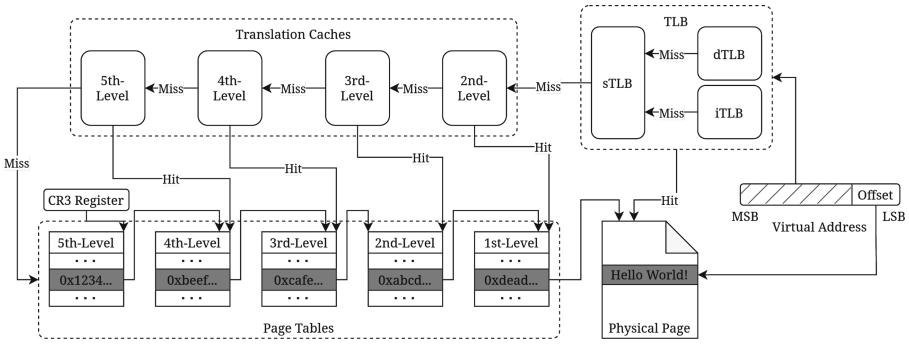

Fig. 2. Schema of the address translation process, showing the interaction of translation caches and the page table hierarchy.

To translate an address, the MMU traverses these page table structures in what is known as a *page table walk*. Figure 1 depicts a page table walk for a system that uses five-level paging, as implemented in the x86_64 architecture. First, the MMU finds the physical address of the root page table by reading the CR3 register. The prefix of the virtual address is split into five 9-bit slices, which index the page tables of successive lower levels. The MMU begins the page table walk by using the most significant 9-bit slice to index the root (or fifth-level) page table, selecting a PTE that points to the next lower-level page table. The process continues iteratively until the page with the requested data is found. The remaining address bits are used as an offset within that page.

2.2 MMU Caches

Regularly performing multi-level page table walks that retrieve each entry from physical memory can severely degrade system performance. To avoid this where possible, CPUs cache recently resolved addresses in a dedicated structure called TLB. That is, the TLB stores the PTEs of recently translated addresses, allowing the MMU to avoid the page table walk for these addresses. While this avoids most walks, TLB misses can still add significant overhead [6,23].

To reduce the overhead of TLB misses, processors cache partial address translation information. One microarchitectural design for implementing this, and the focus of this paper, is with translation caches [5], which store higher level PTEs that match prefixes of recently resolved addresses. The MMU uses a prefix of the virtual address to index these caches [18], allowing it to avoid the initial part of the page table walk and resume the walk from the cached location. As depicted in Fig. 2, the MMU first looks up the virtual address in the TLB. In the case of a hit, it uses the address to bypass the whole page table walk. Otherwise, the MMU proceeds to progressively higher levels of translation caches, until a match is found from where it then starts the page table walk. In practice, the MMU typically queries the translation caches for all levels in parallel, and uses the result of the most specific (lowest level) hit found.

Alternative designs for caching partial address translation information also exist, in AMD Opteron processors for example, the page walk cache is a fully associative cache which stores the PTE [6], indexed by its physical address.

2.3 Desynchronization

To reverse-engineer the TLB, Tatar et al. [32] use desynchronization, which is an approach adapted from cache storage channels [16] for address translation. The core idea of cache storage channels is to break cache coherency, creating a scenario wherein a cached value differs from its corresponding memory contents. Reading the desynchronized data then signals a cache hit or miss, depending on whether the cached value or stored memory content is read. This provides a mechanism to observe whether certain entries have been evicted from the TLB.

Persistent desynchronization is possible because the TLB does not enforce memory coherency. Therefore, when Tatar et al. overwrite a PTE in one of the first-level page tables stored in memory, the corresponding TLB entries are not subsequently invalidated or changed. With this, they directly change the physical address to which the virtual address translates to. If the desynchronized entry is present in the TLB, translating an address using the corresponding PTE results in its original physical address. However, if the entry has been evicted from the TLB, the same virtual address translates to a different physical address. By selecting physical addresses with different contents, TLB hits and TLB misses can be distinguished.

2.4 Replacement Policies

When placing new data into a full cache, the CPU selects which entry to replace based on a given *replacement policy*. Ideally, the replacement policy chooses the entry for eviction that will be used most distantly in the future [9]. Replacement policies use past access behavior to take a best guess at this.

Permutation policies [1] are a popular class of replacement policies that maintain an eviction order for all of the blocks stored in a cache set. This order can be represented using *permutation vectors*, where the right-most elements are chosen for eviction first. A permutation vector π_i indicates how each position in the original order is updated upon an access to the entry at position i.

Pseudo least-recently used (PLRU) is a commonly used example of such a permutation policy. PLRU aims to approximate the *least-recently used* (LRU) policy, while reducing the amount of resources needed to track the least recently used entry. In the frequently used tree-based PLRU policy, the entries are organized in a binary tree, where each node keeps track of which of its subtrees were used least recently. Figure 3 depicts a tree-based PLRU and the corresponding permutation vectors. As illustrated in the figure, the permutation vector $\pi_3[1]$ is 0. This means that upon a hit to the entry at position 3 (d), the entry at position 0 (b), is updated to position 1 (according to its index). Along with the other positional updates and the other permutation vectors, this permutation behavior defines the tree-based PLRU replacement policy.

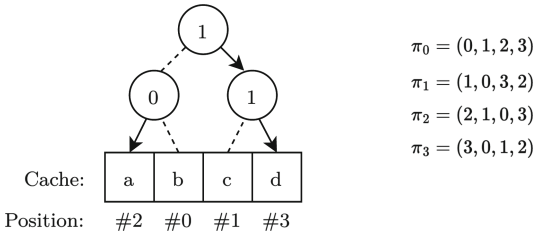

Fig. 3. A sample state of the PLRU replacement policy and the identifying permutation vectors [32].

3 Strategy

We present our approach for reverse-engineering translation caches. We observe that TLB desynchronization [32] operates on fundamental properties of caching and can therefore reliably distinguish between cache hits and misses. We build on the technique, applying it to translation caches implemented within Intel CPUs, which we choose for their prevalence and dominance in the CPU market [2,26, 30].

We first detail how to apply TLB desynchronization to translation caches, and achieve translation cache desynchronization. We discuss optimizations that we extend on previous approaches with to further reduce the noise of measuring cache hits and misses in translation caches.

3.1 Translation Cache Desynchronization

The TLB desynchronization strategy of Tatar et al. [32] provides a means of reverse-engineering the TLB. Unlike previous approaches based on timing, TLB desynchronization operates on fundamental cache properties making it significantly more precise and robust.

To apply desynchronization to translation caches we overwrite an intermediate-level PTE in memory instead of a first-level, as would be done for TLBs. Address translations that use the newly-desynchronized PTE consequently use a new page table for the next-lower level which has been desynchronized. We prepare this page table such that it provides mapping for the same address ranges as the original one, but these mappings point to different physical addresses. Following Tatar et al., we ensure that the content of the physical pages of the new mapping differs from that of the original mapping. This allows us to differentiate between a translation cache hit and a translation cache miss with a single read from the address range:

- If the affected PTE is in the translation cache after desynchonization, translating an address that uses it accesses the original physical address.
- If this entry is evicted, translating the same virtual address results in a different physical address (the one we overwrite with in desynchronization).

In order for these follow-up observations to reliably distinguish translation cache hits and misses, we must ensure that the translation process (page table walk) indeed reaches the target page table level. This will not occur in scenarios where address translation is served from the TLB or from lower-level translation caches, which we do not desynchronize. In such scenarios the page walk process skips the target page translation level which we target, ignoring the desynchronized translation cache.

3.2 Prefix Alignment

We now discuss how we can use a property that we call *prefix alignment* to ensure that the address translation process always touches the PTE of interest.

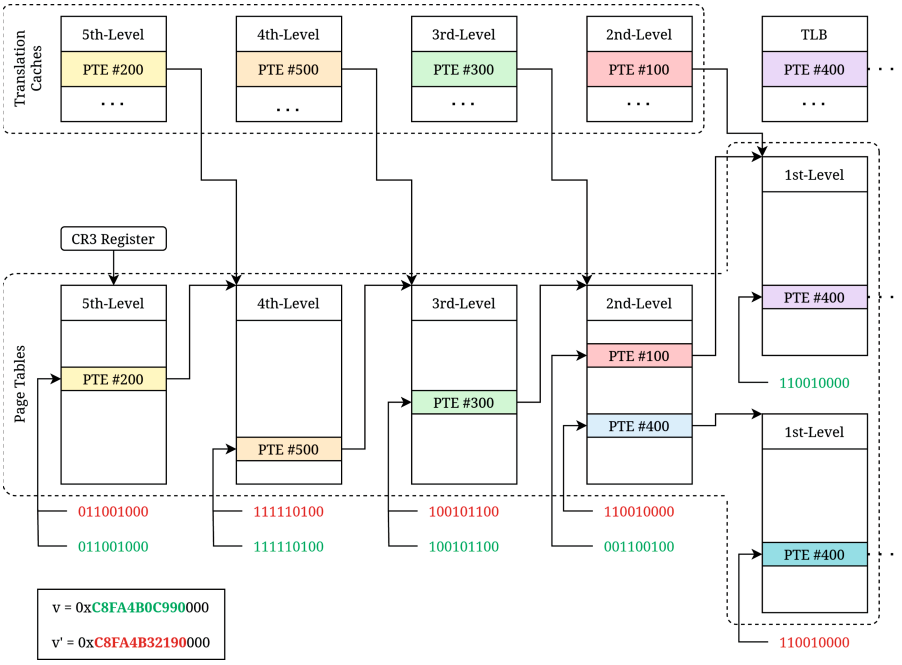

Fig. 4. Page table layout for two virtual addresses, v and v', that are prefix aligned for down to the third-level page table, with separate second-level translation cache and TLB entries.

Suppose we want to measure whether the third-level PTE of a virtual address v is cached. Assume that the lower-level PTEs of v are present in either the TLB or the second-level translation cache. After having desynchronized the corresponding translation cache entry, we need to trigger a translation process that touches the third-level PTE of v to measure a MMU cache hit or miss. Simply

re-translating the same address v is not an option, because the translation process will take the shortcut offered by the TLB or otherwise by the second-level translation cache.

A naïve way to avoid this problem would be to evict the first- and second-level PTEs from the TLB and second-level translation cache, respectively. While this approach can work, it introduces system noise and increases the complexity of the technique.

We opt for an alternative approach, which instead circumvents re-hitting the TLB and PTEs at lower levels than our target. Instead of re-translating v, we translate a *prefix-aligned* address v' that shares all upper-level PTEs with v, down to and including the third-level PTE. Figure 4 depicts the page tables corresponding to two such example addresses. In order for the lower-level translation cache and TLB entries for v and v' to not interfere with one another, the second-level PTEs of v and v' must differ. Hence, in a system with five-level paging, the two virtual addresses v and v' share the three top-most significant 9-bit slices, but are different in the following 9-bit slice, which is used to index the second-level page table. Because the MMU selects a translation cache or TLB entry based on the longest matching prefix of the translated address, the MMU cannot use the lower-level translation cache and TLB entries of v when translating v'. With this approach we force the subsequent page table walk to include the target-level PTE and are able to check whether it is cached or not.

This method is applicable for any target-level PTE by ensuring the target- and all upper-level PTEs are shared, and the next lower-level PTEs of two or more addresses are different. Since page tables contain 512 PTEs, for each target level we can construct up to 512 prefix-aligned addresses that have unique PTEs on the next lower-level.

3.3 Limitations

As is the case in past work [35], desynchronization cannot differentiate between translation cache hits and page walk cache hits. To the best of our knowledge, Intel does not implement page walk caches. Earlier AMD processors did implement it [6], but AMD's documentation is not clear about the current implementation [4].

4 Reverse Engineering

We reverse-engineer translation caches by using translation cache desynchronization. We design and evaluate experiments to determine their behavior and implementation details. We extend the approach of Tatar et al. [32] with additional properties relevant to translation caches. With this we discern the existence of certain cache structures, their hierarchy, and relationships.

All experiments focus on reverse engineering the named properties on a single translation cache level at a time, but they are applicable to all levels.

Table 1. System configurations of our test environment.

Processor	Alder Lake i7-1260P	Rocket Lake i7-11700KF	Kaby Lake i5-7500	Skylake i7-6700	Haswell 4415U
Hardware					
Cores	4/8[a]	8	4	4	2
Hyperthreading	✓/✗[b]	✓	✗	✓	✓
Memory size	32 GB	64 GB	16 GB	32 GB	4 GB
Memory type	DDR4	DDR4	DDR4	DDR4	DDR4
Software					
OS	Debian 12	Ubuntu 22.04.3 LTS	Arch Linux	Debian 12	Ubuntu 22.04.4 LTS
Kernel version	6.1.0-22-amd64	5.15.0-116-generic	6.9.10-arch1-1	6.1.0-18-amd64	6.5.0-41-generic
Architecture	x86-64	x86-64	x86-64	x86-64	x86-64

[a] Alder Lake implements four performance and eight efficiency cores.
[b] Only performance cores support hyperthreading.

We experiment with five processor models, summarized in Table 1. On the Alder Lake system we also differentiate between *performance* and *efficiency* cores, often referred to as *P-Cores* and *E-Cores*. They differ in cycle frequency and hyperthreading support and implement different microarchitectures. Thus, our experiments cover six microarchitectures. We note that none of these systems support five-level paging. In Table 2 we summarize all of our results.

4.1 Cache Existence

We first verify the existence of translation caches at various target levels. According to the Intel developer manual, the MMU can implement any or all level translation caches [18].

To confirm the existence of a translation cache at a given target level, we trigger translation for an address. We then desynchronize the target level PTE and finally access a prefix aligned address to check whether the address is cached. If the target-level translation cache exists, reading from a prefix aligned address still results in its original physical address. However, when the translation cache does not exist, reading from the prefix aligned address results in a different physical address, as the MMU resorts to accessing the in-memory PTE to translate the address.

Our experiments show that the MMU typically implements all levels of translation caches. If translation caches are omitted, this concerns the uppermost levels.

4.2 Cache Hierarchy

Our next aim is to find out whether the translation cache consists of multiple layers, similar to the TLB or the CPU instruction and data caches. These typically implement multiple layers where the first layer is split between instructions and data, whereas lower layers are shared between both.

Shared Layer. First, we determine whether the target translation cache implements a shared layer. Using an instruction fetch, we trigger translation for a

target address and desynchronize its target-level PTE. Then, we use a data read to trigger translation of a prefix aligned address. If read loads from the original physical address associated to the prefix aligned address, the translation cache implements a shared layer. Otherwise, we conclude that the translation cache is split between data and instructions. We find that all translation caches implement a shared layer, except for the second-level translation caches of Alder Lake's efficiency cores.

Split Layer. Additionally, we implement an experiment to determine the existence of a split layer. First, we bring a target PTE to the translation cache using one access type and desynchronize it. Afterwards, we evict the related translation cache entry from a potential shared layer by triggering many translations for different addresses using the other access type. Finally, we use the original access type again, to trigger translation of a prefix aligned address. If accessing this prefix aligned address still results in the originally associated data, the translation cache must implement a split layer. If not, we conclude that the translation cache does not implement a split layer. The experiment results indicate that none of the microarchitectures we examine implement a split layer.

Our observations suggest that Alder Lake's E-cores do not implement a second-level translation cache, because it implements neither a shared nor a split layer. However, this is a contradiction to the translation cache existing. Analyzing the case, we conclude that the second level translation cache must be implemented as a semi-split cache. That is, entries that are introduced by one access type can be *evicted* using the other access type, but a cache entry introduced by one access type cannot be *used* by the other access type.

4.3 Sets, Ways, and Hash Functions

To reverse-engineer the number of sets and ways, as well as the hash function implemented by translation caches, we first assume, without loss of generality, that the target translation cache is split into S sets of W ways. We further assume that the cache implements a hash function $H : \text{address} \rightarrow \{0, 1, \ldots, S-1\}$ that maps a given address to a set. We guess multiple combinations of S, W, and H, which are similar to results from past research.

We then use the guessed hash function H to generate multiple random groups of $W + 1$ addresses whose target-level PTEs map to the same translation cache set and use these to validate the guess. Specifically, we trigger an address translation for each of the addresses in the group, loading them to the respective translation cache. We then desynchronize all of the translation cache entries of these addresses. Finally, we attempt address translation again to detect if any address prefix was evicted from the translation cache, indicating contention on a cache set. We then look for the combination of S, W, and H that minimizes S and W, with priority to minimize W.

Table 2 summarizes the results. Based on the suspicion that the hash functions are similar to those used for TLBs, we test the *LIN* and *XOR* hash functions [12,32]. However, we find that these are not implemented in translation

Table 2. Summary of the reverse-engineered properties of translation caches on different Intel processors.

Property	Alder Lake i7-1260P (P-Cores)	Alder Lake i7-1260P (E-Cores)	Rocket Lake i7-11700KF	Kaby Lake i5-7500	Skylake i7-6700	Haswell 4415U
Second-level translation cache						
Exists	✓	✓	✓	✓	✓	✓
Split Layer	✗	✗	✗	✗	✗	✗
Shared Layer	✓	✗	✓	✓	✓	✓
Sets	8	1	8	8	8	8
Ways	4	28–30	4	4	4	4
Hash function	LIN» 1	n/a[a]	LIN» 1	LIN»1	LIN» 1	LIN» 1
Replacement policy	PLRU	LRU type	HUPLRU	HUPLRU	HUPLRU	HUPLRU
Nested	✗	✗	✗	✗	✗	✗
Unified huge TLB	✗	✗	✗	✗	✗	✗
Supported PCIDs	0[c]	0[c]	0	0	0	0
Third-level translation cache						
Exists	✓	✓	✓	✓	✓	✓
Split Layer	✗	✗	✗	✗	✗	✗
Shared Layer	✓	✓	✓	✓	✓	✓
Sets	1	1	1	1	1	1
Ways	3–4	12–14	3–4	2–4	4	1–3
Hash function	n/a[a]	n/a[a]	n/a[a]	n/a[a]	n/a[a]	n/a[a]
Replacement policy	PLRU	LRU type	PLRU	(P)LRU	PLRU	(P)LRU
Nested	✗	✗	✗	✗	✗	✗
Unified huge TLB	✗[d]	✗	✗[d]	✗[d]	✗	n/a[e]
Supported PCIDs	0[c]	0[c]	0	0	0	0
Fourth-level translation cache						
Exists	✓	✗	✓	✗	✓	✗
Split Layer	✗	–	✗	–	✗	–
Shared Layer	✓	–	✓	–	✓	–
Sets	1	–	1	–	1	–
Ways	2	–	2	–	1	–
Hash function	n/a[a]	–	n/a[a]	–	n/a[a]	–
Replacement policy	MRH	–	MRH	–	n/a[b]	–
Supported PCIDs	0[c]	–	0	–	0	–

[a]Defining a hash function is not useful when the cache is directly mapped.
[b]Cannot explicitly define a replacement policy.
[c]Cannot test PCID support with NOFLUSH bit set to one.
[d]The huge TLB is not wiped by our eviction set.
[e]The machine does not have enough free memory to execute the experiment.

caches. Instead, we find that translation caches implement an undocumented hash function, which we call *LIN» 1* hash.

To calculate the target set t of a virtual address VA, *LIN» 1* computes $t = (tag_{VA} >> 1) \bmod S$. Thus, the translation cache index tag_{VA} of VA is shifted one bit to the right, essentially ignoring the least significant bit of the cache index on set selection. Hence, pairs of adjacent PTEs in a page table map to the same translation cache set.

4.4 Replacement Policy

To reverse-engineer the replacement policy of a target translation cache set, we follow the approach of Abel and Reineke [1]. We use knowledge of the number of ways W and the hash function H attained in our prior experiments to establish a known state in a target set s. We can achieve this by triggering translation for W addresses that, according to H, are known to map to s. Then, we need to determine the replacement order of the known state. We repeat the following steps for all $i \in \{1, \ldots, W-1\}$:

1. desynchronize all entries contained in the target translation cache set
2. trigger translation for i independent addresses which also map to s
3. iteratively observe which entry is evicted by the i-th access

The position of the evicted entry in the replacement order corresponds to $W - i$. Afterwards, we can determine the permutation vectors with the following steps:

1. establish a known state with a known order
2. touch one of the PTEs in the target set to trigger a permutation
3. repeat the process presented before to determine the replacement order of the entries

Repeating this process for all possible W permutations, we obtain the permutation vectors that define the target set's replacement policy.

The results of this experiment are displayed in Table 2. On most examined platforms the algorithm cannot determine the permutation vectors. It only works for the second-level translation cache of Alder Lake's performance cores and for most of the third-level translation caches, except those of Alder Lake's efficiency cores. From manual analysis of more complex access patterns we find that these translation caches do not implement permutation policies. We identify two novel replacement policies, which we present in the following two sections.

HUPLRU. *Hit-updated PLRU* (HUPLRU) is the first replacement policy we discover and it is implemented on most second-level translation caches. Just like tree-based PLRU, it implements a binary tree to keep track of the recency of the entries in the cache. Each node in the tree serves as a decision point, pointing in the direction of the least recently used entry. In PLRU, each node in the tree is updated upon hitting an existing or inserting a new entry to point in the opposite direction of that entry. For HUPLRU this is different. The tree is only updated upon hitting an existing entry, but not upon inserting a new one.

We hypothesize that Intel has chosen to implement this variation of PLRU as an optimization strategy to prevent cache pollution. HUPLRU ensures that one-hit wonders do not pollute the cache, as a second-level PTE must be accessed at least twice to be retained. Such PTEs have high reuse probability because it indicates that memory is being iterated and more hits are to follow. Accessing a second-level PTE only once suggests a 2 MiB jump in the virtual addresses utilized. These are rare due to the principal of spatial locality and should not be retained in the cache.

MRH. On the fourth level translation caches of Rocket Lake and Alder Lake's P-cores, we identify another novel replacement policy, which we call *most-recently hit* (MRH). This is a variant of the more common *most-recently used* (MRU) replacement policy. As the name suggests, MRU chooses the most-recently used entry for eviction. The most-recently used entry is either the one that was most-recently inserted, or the one that was most-recently hit. For MRH, this is different: only the most-recently hit entry can be chosen for eviction. Inserting a new entry does not change the order of replacement.

In general, it seems counterintuitive to replace the entries that were just recently used. However, in our specific case this strategy aligns with observed translation cache behavior and access likelihoods. The third-level PTEs map memory regions of 1 GiB. In Sect. 4.5, we discover that regardless of fourth-level translation cache evictions, the corresponding third-level translation cache entries are retained. Hence, even after a fourth-level eviction, the third-level cache ensures that an access to the same region incurs minimal additional cost. Furthermore, we suspect that Intel chooses MRH over MRU because it is less prone to pollution through unused entries, which otherwise would never leave the cache.

4.5 Nesting

Next, we examine the relationships between the different levels. To this end, we introduce the notion of *nesting*. We say that translation caches are nested when it is guaranteed that all indices of entries present in a lower-level translation cache are prefixed by an index of an entry present in the next upper-level.

To investigate nesting, we trigger translation for a target address and desynchronize its target level PTE. Then we try to evict a higher-level translation cache entry associated with the target address. Finally, we read from a prefix aligned address, sharing the target level PTE of our target address. Measuring a cache miss indicates nesting, whereas a cache hit indicates that there is no such relationship between the different level translation caches. Our evaluation demonstrates that the translation caches we investigate are not nested.

4.6 Huge TLBs

In addition to the *normal* TLB that stores translations of 4 KiB pages, the MMU implements additional TLBs for 2 MiB and 1 GiB huge pages, which we call *huge* TLBs. These TLBs cache PTEs of the second- and third-level page tables mapping the corresponding huge pages. We now check whether these TLBs are unified with their corresponding translation caches [35].

To this end, we introduce a target PTE to the target level translation cache and desynchronize it. We then evict all entries from the 2 MiB or 1 GiB huge TLB, depending on whether we are investigating the second- or third-level translation cache respectively. Finally, we trigger translation for a prefix aligned address, sharing the target PTE. If this translation indicates a translation cache miss, it confirms that huge TLBs and translation caches are unified.

We find that translation caches and huge TLBs are not unified. Considering our refined findings for the structure of translation caches, this is not too surprising. While the 2 MiB and 1 GiB dTLB's structure aligns to some degree with the structure of the second- and third-level translation caches, we already found that translation caches are shared between instructions and data. Hence, the only conceivable relation is to have them unified with the huge sTLBs. However, as we can see in Fig. 5, their structure does not really align with the structure we have observed for translation caches.

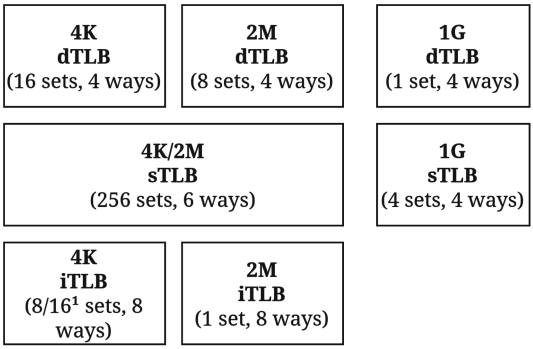

Fig. 5. TLB sizes and layout reported by `cpuid` for all test systems, except Alder Lake and Rocket Lake.

4.7 PCID Support

Finally, we investigate how translation caches enforce process isolation. Intel CPUs manage this using *process context identifiers* (PCIDs). Tatar et al. [32] observe that the TLB supports only four of the 4096 possible PCIDs simultaneously. As soon as an additional PCID is used, the MMU invalidates all TLB entries related to the *least-recently used* PCID. They conclude that the MMU implements a PCID cache with four entries, and that the TLB entries also store an identifier relating them to one of the entries in that cache.

We investigate whether translation caches operate according to a similar principle. To switch PCIDs, we write to the CR3 control register. According to the Intel developer manual [18], setting bit 63 of the CR3 register hints the processor to not flush the translation caches when switching PCIDs. However, we find that whether or not we set bit 63, translation caches do not support PCIDs and are always flushed upon context switches.

If this were not the case, the number of supported PCIDs could be discerned with the following experimental procedure. First, trigger translation for a target

address from a fixed PCID. Next, desynchronize its target level PTE and then switch to $n + 1$ different PCIDs. Finally, switch back to the fixed PCID and trigger translation for a prefix aligned address. The lowest value of n, for which this translation still indicates a cache miss, is exactly equal to the maximum amount of supported PCIDs.

4.8 Cross-Hyperthread PCID Support

One remaining question that subsequently arises is how process isolation is enforced between different hyperthreads running on the same core. As translation caches are implemented per core, and they do not make use of PCIDs, process isolation needs to be enforced in a different way.

To investigate this, we first introduce a new entry to the target translation cache and desynchronize it. Afterwards, we switch to the co-resident hyperthread, and switch to a different PCID from there. Then, in the original hyperthread, we trigger translation of a prefix aligned address, in order to determine whether our target entry is still present in the translation cache. A translation cache hit indicates that process isolation is properly enforced, because the PCID switch only invalidates the translation cache entries related to one hyperthread.

Our experiments show, that all translation caches do implement proper process isolation between hyperthreads. We assume that they make use of the hyperthread ID to relate entries to their corresponding hyperthreads [32], though a different design is conceivable. We leave it to future work to further investigate upon this. The behavior is quite interesting, because the processor needs to find and invalidate the specific entries related to the current hyperthread upon every context-switch.

4.9 Discussion

In this section, we discuss our reverse engineering results. In particular, we compare our results to the previously observed properties of translation caches by van Schaik et al. [35].

Previous Results. Two of the microarchitectures we examine, Haswell and Skylake, were also examined by van Schaik et al. [35]. The Skylake models examined are identical, but the Haswell models differ. We use the *Intel Pentium 4415U*, whereas they examine the *Intel Core i7-4500U*.

On both microarchitectures, we find different sizes for the second-level translation caches. Van Schaik et al. observe that each of these consists of 24 entries. However, with our experiment to determine the hash function, we instead observe that they consist of 32 entries in total. Additionally, contrary to van Schaik et al., we find that Skylake implements a fourth-level translation cache, and Haswell's third-level translation cache consists of only 1–3 entries instead of 3–4.

There are several explanations for these different findings. First, van Schaik et al. [35] rely on timing measurements to differentiate between translation cache hits and misses. Desynchronization differentiates translation cache hits and

misses based on fundamental properties of the address translation process [32]. Because of this, desynchronization offers far more robust classification than timing measurements, which are susceptible to noise [32].

Amplifying this, van Schaik et al. have to consistently evict the PTEs from the CPU data caches, the TLB, and lower-level translation caches. Desynchronization does not require such techniques, as the CPU data caches do not interfere with desynchronization, and must run in kernel mode, allowing us to easily flush the MMU caches. Additionally, in Sect. 3.2, we propose prefix alignment, which allows us to touch a target-level PTE without flushing the lower-level MMU caches at all.

Furthermore, as we discover in Sect. 4.4, translation caches may implement complex replacement policies. As we cannot find any hints that van Schaik et al. [35] consider that, this might be another explanation for the differing results.

5 Talbot

We present Talbot, a tool that automates the processes for reverse engineering translation caches in Intel processors. Talbot implements all of the experiments we describe in Sect. 4. In this section, we discuss implementation challenges and limitations, and we evaluate the tool.

5.1 Challenges

Several non-trivial technical challenges arise in the effort of automating various reverse engineering processes.

Out-of-order execution, along with scheduled and asynchronous events, complicate running our experiments as they introduce interference and system noise. Where possible we mitigate these effects, for example by implementing strict pointer chasing for the relevant memory accesses, and disabling preemption and interrupts.

Additionally, due to the large memory areas mapped by upper-level page tables, we also have to optimize the required memory. We make use of overlapping memory areas and page table entries to implement memory management. Finally, we encounter some trouble with flushing the translation caches in between experiments, making the experiment results very unstable. We suspect that this is related to the replacement policy state that the translation cache is left in after flushing. This results in certain entries being evicted early, before the corresponding cache set is completely filled. Consulting with the Intel developer manual [18], we find that using `mov` instructions targeting the CR3 register is the most stable approach across our different test platforms.

5.2 Limitations

Talbot only supports four-level paging because none of the machines we tested feature five-level paging. We leave extending the tool for five-level paging to future work.

The set of hypothesized hash functions that Talbot supports is limited to *LIN*, *XOR*, and *LIN» 1*. When Talbot cannot attribute any of these hash functions as the tested system's correct candidate, it produces minimal eviction sets, using the single holdout method [22,28]. Similarly, Talbot only provides the permutation vectors when reverse engineering the replacement policy. Manual work is required to draw conclusions about the implemented policy. Analysts running the tool must have some understanding of the experiments it performs. All experiments are run, regardless of whether the translation cache exists or not. Furthermore, Talbot even prints out a hash function, if the translation cache is directly mapped.

All of the experiments our tool implements do not account for multi-layer caches. We did not encounter any test platforms that implement a multi-layer hierarchy. Regarding replacement policies, in its current state Talbot is limited to reverse-engineering permutation-based policies.

5.3 Evaluation

For our experiments, Talbot is particularly effective in identifying cache existence, hierarchies, and hash functions. Across multiple processor architectures, it reliably detects different architectures with high accuracy, though sometimes there is a need to run the tool multiple times and average the results. Talbot also consistently identifies features such as PCID support, nesting, and the relationship to huge TLBs. However, our test environment does not challenge these experiments, as these properties are consistent across the different microarchitectures tested.

Despite this, Talbot also has shortcomings, mostly related to the limitations we mentioned. Most prominently, the automated reverse engineering of replacement policies is not completely reliable. This is because the assumption that translation caches implement permutation policies rarely holds. Future work may try to adapt the approach of Vila et al. [36] for reverse-engineering replacement policies to the case of translation caches. As a more limited solution, we provide scripts that identify the HUPLRU and MRH replacement policies on the affected systems.

In summary, Talbot offers a robust means of quickly reverse engineering translation caches, with careful and knowledgeable interpretation of its results. The tool is a basis that can be further developed for reverse engineering diverse replacement policies and its inter-experiment relationships.

6 Conclusion

In this work, we adopt the TLB desynchronization strategy [32] to obtain translation cache desynchronization and reverse-engineer translation caches. We obtain a refined description of the implementation details and largely extend the understanding of translation caches, opposed to previous reverse-engineering efforts.

We go beyond the sizes of translation caches, and investigate more specific implementation details, including cache structure, replacement policy, process isolation capability, and relationships.

Our results show that the potential of attacks utilizing the address translation process, and especially translation caches, is not fully utilized yet. Future work can build upon the properties we expose and construct novel attacks or improve the performance of existing attacks.

Additionally, we implement Talbot, a tool to automatically reverse-engineer specific properties of translation caches on arbitrary Intel processors. Future work can use Talbot to evaluate the implementation of translation caches on other existing and future microarchitectures. Enhancing the tool's reverse engineering capabilities would be especially useful, in particular concerning more complex replacement policies.

Acknowledgments. This work was supported by an ARC Discovery Project number DP210102670; and the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy - EXC 2092 CASA - 390781972.

References

1. Abel, A., Reineke, J.: Measurement-based modeling of the cache replacement policy. In: RTAS, pp. 65–74 (2013)
2. Alcorn, P.: AMD and Intel CPU Market Share Report: Recovery on the Horizon (2023). https://www.tomshardware.com/news/amd-and-intel-cpu-market-share-report-recovery-looms-on-the-horizon
3. Aldaya, A.C., Brumley, B.B., ul Hassan, S., García, C.P., Tuveri, N.: Port contention for fun and profit. In: IEEE SP, pp. 870–887 (2019)
4. AMD: AMD64 Architecture Programmer's Manual Volume 2: System Programming (2024)
5. Barr, T.W., Cox, A.L., Rixner, S.: Translation caching: skip, don't walk (the page table). In: ISCA, pp. 48–59 (2010)
6. Bhargava, R., Serebrin, B., Spadini, F., Manne, S.: accelerating two-dimensional page walks for virtualized systems. In: ASPLOS, pp. 26–35 (2008)
7. Bosman, E., Razavi, K., Bos, H., Giuffrida, C.: Dedup Est Machina: memory deduplication as an advanced exploitation vector. In: IEEE SP, pp. 987–1004 (2016)
8. Braun, B.A., Jana, S., Boneh, D.: Robust and efficient elimination of cache and timing side channels, arXiv:1506.00189 (2015)
9. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 4th edn. MIT Press (2022)
10. Evtyushkin, D., Ponomarev, D., Abu-Ghazaleh, N.: Jump over ASLR: attacking branch predictors to bypass ASLR. In: MICRO, pp. 1–13 (2016)
11. Ge, Q., Yarom, Y., Cock, D., Heiser, G.: A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. J. Cryptogr. Eng. **8**(1), 1–27 (2018)
12. Gras, B., Razavi, K., Bos, H., Giuffrida, C.: Translation leakaside buffer: defeating cache side-channel protections with TLB attacks. In: USENIX Security, pp. 955–972 (2018)

13. Gras, B., Razavi, K., Bosman, E., Bos, H., Giuffrida, C.: ASLR on the line: practical cache attacks on the MMU. In: NDSS (2017)
14. Gruss, D., Lettner, J., Schuster, F., Ohrimenko, O., Haller, I., Costa, M.: Strong and efficient cache side-channel protection using hardware transactional memory. In: USENIX Security, pp. 217–233 (2017)
15. Gruss, D., Maurice, C., Fogh, A., Lipp, M., Mangard, S.: Prefetch side-channel attacks: bypassing SMAP and kernel ASLR. In: CCS, pp. 368–379 (2016)
16. Guanciale, R., Nemati, H., Baumann, C., Dam, M.: Cache storage channels: alias-driven attacks and verified countermeasures. In: IEEE SP, pp. 38–55 (2016)
17. Hund, R., Willems, C., Holz, T.: Practical timing side channel attacks against kernel space ASLR. In: IEEE SP, pp. 191–205 (2013)
18. Intel Inc.: Intel 64 and IA-32 Architectures Software Developer Manuals
19. Kocher, P., et al.: Spectre attacks: exploiting speculative execution. In: IEEE SP, pp. 1–19 (2019)
20. Lipp, M., et al.: Meltdown: reading kernel memory from user space. In: USENIX Security, pp. 973–990 (2018)
21. Liu, F., et al.: CATalyst: defeating last-level cache side channel attacks in cloud computing. In: HPCA, pp. 406–418 (2016)
22. Liu, F., Yarom, Y., Ge, Q., Heiser, G., Lee, R.B.: Last-level cache side-channel attacks are practical. In: IEEE SP, pp. 605–622 (2015)
23. McCurdy, C., Cox, A.L., Vetter, J.S.: Investigating the TLB behavior of high-end scientific applications on commodity microprocessors. In: ISPASS, pp. 95–104 (2008)
24. Osvik, D.A., Shamir, A., Tromer, E.: Cache attacks and countermeasures: the case of AES. In: CT-RSA, pp. 1–20 (2006)
25. Paccagnella, R., Luo, L., Fletcher, C.W.: Lord of the ring(s): side channel attacks on the CPU on-chip ring interconnect are practical. In: USENIX Security, pp. 645–662 (2021)
26. PassMark Software: PassMark CPU Benchmarks - AMD vs Intel Market Share (2024). https://www.cpubenchmark.net/market_share.html
27. Pessl, P., Gruss, D., Maurice, C., Schwarz, M., Mangard, S.: DRAMA: exploiting DRAM addressing for cross-CPU attacks. In: USENIX Security, pp. 565–581 (2016)
28. Qureshi, M.K.: New attacks and defense for encrypted-address cache. In: ISCA, pp. 360–371 (2019)
29. Razavi, K., Gras, B., Bosman, E., Preneel, B., Giuffrida, C., Bos, H.: Flip Feng Shui: hammering a needle in the software stack. In: USENIX Security, pp. 1–18 (2016)
30. Shilov, A.: Arm-Based CPUs Could Double Notebook PC Market Share by 2027 (2023). https://www.tomshardware.com/news/arm-based-cpus-set-to-double-notebook-pc-market-share-by-2027
31. Sprabery, R., Evchenko, K., Raj, A., Bobba, R.B., Mohan, S., Campbell, R.H.: A novel scheduling framework leveraging hardware cache partitioning for cache-side-channel elimination in clouds, arXiv:1708.09538 (2017)
32. Tatar, A., Trujillo, D., Giuffrida, C., Bos, H.: TLB;DR: enhancing TLB-based attacks with TLB desynchronized reverse engineering. In: USENIX Security, pp. 989–1007 (2022)
33. van der Veen, V., et al.: Drammer: deterministic Rowhammer attacks on mobile platforms. In: CCS, pp. 1675–1689 (2016)
34. van Schaik, S., Giuffrida, C., Bos, H., Razavi, K.: Malicious management unit: why stopping cache attacks in software is harder than you think. In: USENIX Security, pp. 937–954 (2018)

35. van Schaik, S., Razavi, K., Gras, B., Bos, H., Giuffrida, C.: RevAnC: a framework for reverse engineering hardware page table caches. In: EuroSec, pp. 1–6 (2017)
36. Vila, P., Ganty, P., Guarnieri, M., Köpf, B.: CacheQuery: learning replacement policies from hardware caches. In: PLDI, pp. 519–532 (2020)
37. Wang, Y., Paccagnella, R., He, E.T., Shacham, H., Fletcher, C.W., Kohlbrenner, D.: Hertzbleed: turning power side-channel attacks into remote timing attacks on x86. In: USENIX Security, pp. 679–697 (2022)
38. Yarom, Y., Falkner, K.: Flush+Reload: a high resolution, low noise, L3 cache side-channelattack. In: USENIX Security, pp. 719–732 (2014)
39. Zhang, Z., Cheng, Y., Liu, D., Nepal, S., Wang, Z., Yarom, Y.: PThammer: cross-user-kernel-boundary Rowhammer through implicit accesses. In: MICRO, pp. 28–41 (2020)
40. Zhang, Z., Tao, M., O'Connell, S., Chuengsatiansup, C., Genkin, D., Yarom, Y.: BunnyHop: exploiting the instruction prefetcher. In: USENIX Security, pp. 7321–7337 (2023)
41. Zhao, Z.N., Morrison, A., Fletcher, C.W., Torrellas, J.: Binoculars: contention-based side-channel attacks exploiting the page walker. In: USENIX Security, pp. 699–716 (2022)
42. Zhou, Z., Reiter, M.K., Zhang, Y.: A software approach to defeating side channels in last-level caches. In: CCS, pp. 871–882 (2016)