





Risky Translations: Securing TLBs against Timing Side Channels

Florian Stolz*¹ , Jan Philipp Thoma*¹ , Pascal Sasdrich¹  and Tim Güneysu^{1,2} 

¹ Ruhr University Bochum, Horst Görtz Institute for IT Security, Bochum, Germany

² DFKI GmbH, Cyber-Physical Systems, Bremen, Germany

{florian.stolz,jan.thoma,pascal.sasdrich,tim.gueneysu}@rub.de

Abstract. Microarchitectural side-channel vulnerabilities in modern processors are known to be a powerful attack vector that can be utilized to bypass common security boundaries like memory isolation. As shown by recent variants of transient execution attacks related to Spectre and Meltdown, those side channels allow to leak data from the microarchitecture to the observable architectural state. The vast majority of attacks currently build on the cache-timing side channel, since it is easy to exploit and provides a reliable, fine-grained communication channel. Therefore, many proposals for side-channel secure cache architectures have been made. However, caches are not the only source of side-channel leakage in modern processors and mitigating the cache side channel will inevitably lead to attacks exploiting other side channels. In this work, we focus on defeating side-channel attacks based on page translations.

It has been shown that the Translation Lookaside Buffer (TLB) can be exploited in a very similar fashion to caches. Since the main caches and the TLB share many features in their architectural design, the question arises whether existing countermeasures against cache-timing attacks can be used to secure the TLB. We analyze state-of-the-art proposals for side-channel secure cache architectures and investigate their applicability to TLB side channels. We find that those cache countermeasures are not directly applicable to TLBs, and propose TLBCOAT, a new side-channel secure TLB architecture. We provide evidence of TLB side-channel leakage on RISC-V-based Linux systems, and demonstrate that TLBCOAT prevents this leakage. We implement TLBCOAT using the `gem5` simulator and evaluate its performance using the PARSEC benchmark suite.

Keywords: Microarchitecture · TLB · Side Channel · Randomization

1 Introduction

The ever-growing complexity of modern processors enables the ongoing acceleration of computing power. Today’s processors feature a variety of performance optimizations like forwarding mechanisms, speculation, and caches. The software stack that is built on top of this often treats the hardware as a trust anchor. Although the clash between performance-driven hardware with respect to secure software processes was demonstrated many times, e.g., by demonstrating that cache-timing can leak secret key material [Ber05], advances in this direction only recently gained momentum with the disclosure of transient execution attacks. These attacks include Spectre [KHF⁺19], Meltdown [LSG⁺18], and successors (see generally [CBS⁺19]) and utilize microarchitectural side-channel vulnerabilities to exploit leakage and extract secrets from transient CPU states while other attacks profile the

*The authors contributed equally to this work.

timing behavior of shared CPU components to leak cryptographic key material [Ber05, GRBG18, PGM⁺16, YF14].

Most of the recent microarchitectural attacks focus on the cache side channel since it is very reliable and fine-grained. Much effort has been put into preventing the exploitation of cache side channels [WUG⁺19, TNF⁺23, TZBR21, SQ21, Qur18]. When these countermeasures are integrated into a new generation of processors, attacks will inevitably evolve and exploit other vulnerable processor resources like the Translation Lookaside Buffer (TLB) which stores frequently used page mappings, translating virtual to physical addresses. First signs of this evolution are already visible — for example, the TLBLEED attack [GRBG18] was the first attack to demonstrate a timing side channel on TLBs by exploiting the observable TLB access patterns of cryptographic software to reconstruct the secret key. Tatar *et al.* [TTGB22] perform rigorous reverse engineering efforts on Intel TLBs and demonstrate enhanced attacks building on the improved understanding of the hardware. Due to the similarities in the architectural design of TLBs and caches, many cache attacks are transferable to TLBs, for example PRIME+PROBE [OST06, TOS10]. However, the resolution of the TLB side channel is much more coarse-grained since translations are stored per page (usually 4 kB) opposed to per cache line (64 Bytes). To bypass this limitation, the TLBLEED attack exploits the temporal TLB access patterns instead of the spatial information; i.e., the attacker observes *when* a page was accessed, not *if*. Follow up work presented further attack vectors on CPU TLBs [DXS19] and GPU TLBs [NPGB21]. The former develops a three-step model that discloses TLB access patterns that might lead to side-channel leakage. The latter reverse engineers the TLB hierarchy in Nvidia GPUs and finds that the device under test features a TLB that is shared between multiple Streaming Multiprocessors (SMs). This distinguishes TLBs in GPUs from those in CPUs, which are typically shared between hyperthreads at most.

The architectural similarities of caches and TLBs may suggest that some countermeasures against cache-timing side channels can be directly transferred to TLBs. However, until now, there has not been a thorough study on whether the adoption of existing cache countermeasures is sufficient to prevent side-channel attacks on TLBs. To the best of our knowledge, there is currently only one attempt at side-channel secure TLB designs. Deng *et al.* [DXS19] propose two alternative candidates for secure TLB architectures which are directly derived from corresponding cache countermeasures. The *SP TLB* design uses static partitioning between security domains to avoid leakage throughout these domains, while the second proposal, *RF TLB*, aims to leverage the spatial locality of accesses to populate the TLB and hide the accessed pages from an attacker. The static partitioning approach taken by *SP TLB* has limited flexibility and increases the TLB misses per kilo instructions by 207.5% [DXS19]. The *RF TLB* design, which is directly derived from the cache counterpart *RandomFillCache* [LL14], requires software developers to distinguish secure- from not secure pages. Moreover, it is still feasible for an attacker to observe TLB accesses, albeit with added spatial disturbance. Any TLB protection should not require exposure of TLB internals to the programmer. Security should be provided without the need to assign applications or separate pages to specific security domains. Otherwise this would shift the burden to the programmer or the Operating System (OS), which have to identify the pages that might be exploited by a side-channel attack. Furthermore, the OS should not be required to explicitly manage the TLB for all applications, except for small interactions such as initiating flushes, e.g., when access permissions are changed. Since TLBs are placed directly in the critical path of CPU caches (i.e., when caches are physically indexed), the performance overhead for the TLB access must be minimized.

Contributions. In this paper, we explore the applicability of state-of-the-art cache countermeasures to TLBs and their effectiveness against various attack vectors. We find that straightforward adoption of cache designs is not advisable to protect TLBs and present TLBCOAT (TLB Countermeasure against Attacks based on Timing), a novel TLB design

which enables strong protection against attacks while only marginally affecting the performance. We implement TLBCOAT in `gem5` [LPAA⁺20] and evaluate its security and performance. We show that classic set-associative TLBs can leak information even using a non-simultaneously shared setting in our simulation environment running a RISC-V Linux by exploiting the recently implemented Process-Context Identifiers (PCIDs). Crucially, we show that TLBCOAT prevents the leakage of such information. Finally, we provide a hardware implementation of TLBCOAT to demonstrate the feasibility in real-world systems.

2 Background

This section introduces background information on TLBs and virtual memory management. We further introduce common attack vectors on caches that are also applicable to TLBs.

2.1 Virtual Memory and the Translation Lookaside Buffer

Modern CPUs execute multiple processes simultaneously, each of which requires space in physical memory. Thus, each process has its own virtual address space, which acts as a memory abstraction and is mapped to the physical memory by the Memory Management Unit (MMU). This abstraction is usually implemented using paging. A *page* thereby maps a fixed size chunk of virtual memory to an equally sized chunk of physical memory, a so-called *page frame*. Typically, regular pages have a size of 4 kB although larger pages are allowed in several Instruction Set Architectures (ISAs) which are usually only used in special cases. Upon a memory access the MMU translates the provided virtual address to a physical address using the *page table*, which stores the mapping location as well as access rights for each page [Tan09].

Since searching the page table on every access is very slow, the most recently used translations are stored in the TLB. Their internal structure can vary depending on the CPU and the ISA; however, most TLBs are implemented as a set-associative structure. Few vendors implement fully-associative TLBs for the small first-level TLBs. Since full associative structures need to be searched entirely on every access, the performance does not scale for larger TLBs. After a translation is placed in the TLB, subsequent memory accesses will bypass the page table lookup to save performance. Modern CPUs often use a TLB hierarchy to improve performance even further. For example, x86 CPUs commonly separate the TLB into data (dTLB) and instruction TLBs (iTLB). Recent processors also employ multiple levels of TLBs, comparable to L1 and L2 caches used for the main CPU caches [Int16a, p. 2-6]. Intel refers to the second level TLB which combines data and instructions as sTLB. A similar structure is also used in many RISC-V cores, such as the *Rocket Chip* [AAB⁺16]. Architectures that support multiple page sizes may add further TLB levels. Compared to contemporary data caches, TLBs are limited in size, as it has been observed that in the general case, most programs only require a few pages simultaneously [Tan09]. An overview of typical TLB parameters is shown in Table 1.

Table 1: Overview of typical TLB parameters using the example of some recent processors. The Nvidia L2- and L3-TLB feature a victim slot where the least recently evicted entry is held (denoted as *+1*). ✓: Shared between cores, ○: Shared between hyperthreads, ✗: Not shared

	Intel Skylake [Int16a, Tab. 2-5]			AMD Zen2 [AMD17, p.25]				Nvidia Pascal [NPGB21]		
	iTLB	dTLB	sTLB	L1i	L2i	L1d	L2d	L1	L2	L3
Entries	128	64	1536	64	512	64	1536	16	64+1	1024+1
Assoc.	8	4	12	full	8	full	12	full	8	8
Shared?	✗	○	○	○	○	○	○	✗	✗	✓

A set-associative TLB can be imagined as a table-like structure with *ways* and *sets*, corresponding to columns and rows respectively. Since TLBs translate virtual to physical addresses, all TLBs are indexed using virtual addresses. A simple set addressing scheme may divide the virtual address into $\log_2(\text{page_size})$ offset bits, $\log_2(\#\text{sets})$ index bits, and the remaining Virtual Page Number (VPN) bits. When a process makes an access to a virtual address, the index bits of the address are used to determine a set of the TLB, i.e., the row in the table analogy. In a w -way set-associative TLB, each set thus contains w entries. Upon access, these entries are searched for the VPN which is part of the virtual address. If the VPN matches one of the entries from the set, the request results in a TLB *hit* and the translation information from that entry is returned. Otherwise, the request results in a TLB *miss* and the page table is searched for the translation data. Once the data is returned from the page table, the TLB selects one entry of the set and replaces it with the new data. Often, the replacement policy is a variant of Least-Recently-Used (LRU), i.e., the entry that has not been used for the longest time is replaced. In recent Intel CPUs, the replacement policy is still based on LRU but includes more complex aspects as reverse engineered in [TTGB22]. Next to the actual translation of the virtual to physical address, a TLB entry also stores the access rights to the respective page as well as some flags [Tan09].

Since the mapping of each program’s virtual memory is isolated, it is necessary to manage the TLB and invalidate entries that contain mappings for processes that are not currently scheduled. Therefore, on each context switch the OS has to change the MMU’s configuration accordingly and invalidate the TLB. A straightforward way of handling this task is a complete TLB flush. The invalidation of the whole TLB ensures that each process starts with an empty TLB. This method was, for example, used by Intel until the Westmere microarchitecture [Kan10]. However, this creates a high overhead which can be avoided if the next scheduled process only requires a few pages and does not utilize the entire TLB. Therefore, since Westmere and similarly on other architectures such as RISC-V [WLA⁺16], each translation can also be tagged with a PCID (or Address space Identifier (ASID) in case of RISC-V and ARM¹). By giving each process a unique ID (e.g., 12-bit ID on x86), the translation entries can be unambiguously assigned to a process. The TLB can stay intact and only remove entries based on its replacement policy. So-called *global pages* can be accessed by all processes and are often used for shared kernel functionalities. Even in case of a complete TLB invalidation it is possible to leave these entries valid [Int16b, Tab. 3-11].

2.2 Side-Channel Attacks on the TLB

The set-associative design of most TLBs is also very common in the main CPU caches. Hence many cache attacks can be transferred to the TLB. Unlike caches, TLBs are usually not shared between physical cores, but CPUs that use hyperthreading share the TLB between those hyperthreads. Furthermore, the CPU schedules different processes on each core which causes a time-division sharing of the TLB between multiple processes. With the introduction of PCIDs, the TLB is no longer flushed upon context switching which allows entries from descheduled processes to survive until the next time they are scheduled.

One of the most popular attacks on caches is PRIME+PROBE [OST06, TOS10]. Because of the set-associative design, a given address can only be mapped to a single TLB-set which consists of w entries where w is the associativity. By observing that the victim makes a secret dependent access to a given address, the attacker can construct w addresses from their address space that map to the same set. By accessing these addresses, the attacker can evict the victim translation from the TLB. Hence, these addresses form an *eviction set*. More importantly, as soon as the victim accesses the page again, one of the addresses from the eviction set will be removed. This can be observed by the attacker and

¹For the remainder of this paper we will use the term PCID to refer to any type of context identifiers.

hence, the secret can be recovered. The fact that TLBs are virtually indexed facilitates this attack since the attacker can simply mirror virtual addresses used by the victim and add a suitable offset to form an eviction set.

However, since the TLB stores entries with page granularity (e.g., 4 kB) opposed to cache line granularity (e.g., 64 Bytes), the attack on the TLB has a reduced *spatial* resolution; that is, the attacker can only observe accesses within a 4 kB range instead of a 64 Byte range. The work presented by Gras *et al.* [GRBG18] exploits the *temporal* access patterns instead. With this, the information used to recover the secret is *when* the page was accessed, not *if*. This is made possible by the likes of hyperthreading and similar technologies, as multiple processes can simultaneously manipulate and observe the state of the TLB. By design, the *temporal information* is very coarse on TLBs that are not simultaneously shared between processes or threads since the attacker can only examine the TLB state after a context switch. However, as shown by Deng *et al.* [DXS19], these configurations may also be vulnerable. In Section 6.2 we demonstrate *spatial information* leakage as part of our security evaluation. By reverse engineering the TLB on recent Intel CPUs, Tatar *et al.* [TTGB22] improved the performance of some of these attacks.

Another common attack against caches is FLUSH+RELOAD [YF14]. However, this attack requires an unprivileged instruction to flush data from the set-associative structure. While the `clflush` instruction allows this for caches on x86 processors, the TLB counterpart `invlpg` is privileged and therefore not useful for most real-world attacks. To the best of our knowledge, there exists no major ISA with an unprivileged TLB flush instruction.

Recent proposals in side-channel secure cache architectures increasingly build upon index randomization [WUG⁺19, TZBR21, WL07] which makes finding eviction sets much more complicated. However, the PRIME+PRUNE+PROBE attack [PGGV21] and derivatives [BDY⁺20] assemble generalized eviction sets which evict a target address with high probability. We cover these attacks later in this paper when we discuss index-randomization in TLBs.

3 Attacker Model

For our attacker, the ultimate goal is the extraction of page access information from the TLB which depend on a secret, thus allowing them to reconstruct it. For this, in accordance with prior research by Gras *et al.* [GRBG18], we assume that an attacker can execute unprivileged code on the target system and can employ an instruction to extract timing information, e.g., such as a cycle counter. The timing information can be used to measure the overhead created by TLB hits/misses, which in turn allows for reverse-engineering the mapping function of the TLB and creating an eviction set. Additionally, by analyzing the victim application, the virtual addresses may be retrieved, which allow for an attack. Specifically, if more than one core is present, the attacker is able to move the execution of code to the same physical core as the victim using system calls. Furthermore, we assume that the OS is trusted, as a malicious OS has full memory access and can trivially extract secrets from all running processes, thus removing the necessity for a TLB-based attack.

Since the implementation details of TLBs differ vastly for modern processor designs and ISAs, the threat model still needs to be tailored to the given circumstances. Properties which require increased attention during the security analysis are (1) the usage of PCIDs and (2) whether Simultaneous Multi-Threading (SMT) is available and enabled. We can thus, in general, differentiate between four scenarios:

Isolated TLBs. In this setting, each physical core has its own TLB and the CPU is not capable of SMT. Therefore, no logical cores exist which share processor resources. Furthermore, the TLB does not employ PCIDs. After each context switch the CPU has to flush all cached translations. In this scenario, only process-internal attacks are feasible. Such attacks can be useful for example in the context of web browsers, which may execute

(malicious) code delivered by the attacker’s website in the same process as the browser.

Non-simultaneously shared TLBs. This case describes most modern processors, which do not employ SMT. To avoid unnecessary slowdowns due to flushing, each translation is tagged with a PCID. Thus, processes can evict each other’s entries and therefore exploit spatial access information after context switches. We demonstrate the leakage in our evaluation environment in Section 6.2 on a simulated RISC-V Linux system. Since the TLB is not simultaneously shared, the attack cannot extract fine-grained *temporal information*.

Simultaneously shared TLBs with PCIDs. If SMT is enabled, logical cores can share physical resources such as the TLB. Translations are tagged with PCIDs, which make flushing upon a context switch unnecessary. Gras *et al.* [GRBG18] attacked this configuration on Intel CPUs to extract *temporal information* by observing page accesses of the victim from a simultaneously running attacker thread. It is also possible to extract *spatial information* in this setting.

Simultaneously shared TLBs without PCIDs. In this scenario, hardware threads share a TLB, but do not employ PCIDs. Therefore, after a context switch the whole or parts of the TLB have to be invalidated. Additionally, some form of hardware tagging is required to prevent multiple threads from accessing each other’s translations. Like the previous setting, this configuration allows leaking *temporal* and *spatial* access information from the co-located process, but not across context switches.

4 On the Effectiveness of Cache Countermeasures for TLBs

A vast variety of side-channel secure cache designs have been proposed in recent years. From the architectural similarities of caches and TLBs, one could conclude that simply applying the already proposed countermeasures is sufficient for a side-channel secure TLB. In this section we show how the unique properties of TLBs interfere with the security and performance of existing countermeasures.

4.1 Partitioning

Cache-timing side channels exploit the shared nature of caches in which a potential victim shares the resource with the attacker. Therefore, splitting the cache into two or more domains for secure and insecure processes and thus separating resources, can be an effective defense against such attacks. We differentiate between *static* and *dynamic* partitioning schemes.

Static Partitioning. Caches that employ static partitioning split the shared cache into multiple domains during design time. An identifier is required to select the appropriate cache partition for each access. If the attacker and the victim belong to different security domains, they are therefore strictly separated and cannot interfere with each other. As mentioned by He and Lee [HL17], security is achieved at the cost of cache performance, because the cache size is effectively smaller. Furthermore, the number of partitions is fixed during design time which causes problems when the number of mutually untrusted processes executed on the CPU exceeds the number of available partitions. HybCache [DFS20] aims to reduce the performance overhead of partitioning by instantiating a fully-associative subcache within the usual set-associative cache for code running in trusted environments like SGX.

Dynamic Partitioning. In dynamic partitioning schemes a monitor (e.g., the OS) can partition the cache at run-time. Unlike static partitioning, this allows the cache to adapt to various situations in which, for example, no secure partition is required. Like before, an identifier is required for selecting each partition. Page [Pag05] uses a directly-mapped cache with additional configuration data attached. The configuration stores the start and

size of each partition as well as other relevant information. The cache is exposed to the software via the ISA, which allows for the creation, deletion and flushing of partitions. Once a partition is created, the size is fixed and cannot be changed. Other schemes, for example, by Qureshi and Patt [QP06], use a monitoring circuit to gather information that is used by a partitioning algorithm to continuously adjust partition sizes. However, as pointed out by Wang *et al.* [WFZ⁺16] resizing capabilities can decrease the security of a cache, because of two side effects. Firstly, the size of a partition increases proportionally to the demand of the application. Via probing, attackers can learn the partition size and infer information about the confidential application. Secondly, schemes as shown by Qureshi and Patt [QP06] do not flush left-over cache lines after making a partition smaller. Other processes can use these left-over cache lines to perform classic timing side-channel attacks, because their content depends on the accesses of the confidential application. Therefore, careful design of the partitioning algorithm as well as partition enforcement are required to create a secure dynamically partitioned cache. Townley *et al.* propose a dynamic partitioning cache for secure enclaves in [TAL⁺22].

Partitioning in TLBs. Partitioning was explored by Deng *et al.* [DXS19] as a possible countermeasure against timing side-channel attacks in TLBs. Their design, *SP TLB*, is split at design time into a secure part, which is only accessible by one application that requires protection, and an insecure part, which is shared between all other processes. The PCID is used to differentiate between the two security domains. The division is way-based and done during design time, making it a static partitioning scheme. Thus, at any time a subset of ways is reserved for a secure application. *SP TLB* provides inherent security against attacks across security domains. However, its static nature is also a downside. Since 50% of the TLB are reserved for secure applications, the other processes have to compete for a half-sized TLB. This is even the case, when no secure application is running, leaving the allocated ways unused. Furthermore, it is not possible to execute multiple mutually untrusted secure applications at the same time as they would compete for the secure partition of the TLB in an insecure manner. Deng *et al.* report an increased TLB miss rate of up to a factor of 3 per kilo instruction for *SP TLB*. Hence, the design lacks scalability for many real-world applications since the workload on the TLB changes dynamically throughout the execution of various programs. Therefore, static partitioning is only an option if the designer knows specific execution characteristics at design time.

To the best of our knowledge no dynamic partitioning scheme exists for TLBs. In general, dynamic partitioning requires some kind of monitoring to balance the requirements of the secure and insecure domain. We assess that this would lead to a high overhead as the OS or a separate hardware component would have to constantly monitor the utilization. Additionally, reassigning ways between domains is not trivial as left-over information can in fact leak information as shown before by Wang *et al.* [WFZ⁺16].

4.2 Randomization

Next to partitioning, randomizing the mapping of addresses to cache entries is one of the most widespread measures to thwart cache-timing side channels. PRIME+PROBE attacks rely on the attacker’s ability to construct small eviction sets, that is, a set of addresses that map to the same cache entries. In traditional caches, this can easily be achieved by choosing the index bits in a certain way. By randomizing the address to cache mapping, the attacker can no longer deterministically construct such eviction sets.

Early index-randomization schemes [WL07, WL08] utilize lookup tables to store the randomized address to cache mapping. While this results in an effectively fully-associative cache, the complexity of a lookup is significantly increased. The CEASER [Qur18] design was the first to propose address encryption for efficient address to cache randomization. By using a trapdoor one-way-function, the design randomizes the mapping of addresses

to cache sets without the need to store the mapping for each address. Since the number of sets within the cache is limited, CEASER requires frequent re-randomization to avoid attacks that reverse engineer the mapping of certain addresses to the cache. Changing the key of the randomization function is not trivial as it effectively causes an implicit cache flush. That is, since after re-randomization a given address maps to a different set of entries with high probability and existing entries for that address are then no longer found upon lookup. Entries that have been modified (*dirty*) need to be written back to main memory prior to the re-randomization.

In the prospect of making re-randomization dispensable, CEASER-S [Qur19] and ScatterCache [WUG⁺19] improve the index-randomization by relaxing the concept of cache sets. In both designs, each cache way uses an independent addressing function, and hence, addresses that collide in one way no longer collide in all cache ways. The randomization function is either built from a low latency block cipher or hash-based. To further protect against attacks like FLUSH+RELOAD that exploit shared memory between processes, ScatterCache introduces domain separation by tweaking the randomization function with a security domain identifier. This results in separate cache entries of shared memory addresses for each process. Since FLUSH+RELOAD relies on the ability of the attacker to evict shared cache entries using the `clflush` instruction, duplicating the entries for each process mitigates the attack. A similar approach is taken by PhantomCache [TZBR21], which relaxes the traditional concept of cache sets even further. The addressing function allows entries to be mapped to multiple indices in each way. To limit the performance overhead, the number of positions is strictly limited. While the PhantomCache approach enlarges the randomization space, the increased search space also affects the lookup complexity.

Recent work [PGGV21] generalizes the index-randomization schemes and investigates the security properties against a profiling attacker. The authors present the PRIME+PRUNE+PROBE attack which constructs probabilistic eviction sets by observing conflicts between attacker controlled addresses and a target address. The attack challenges the assumption that index-randomization is sufficient to achieve side-channel security. The authors propose implementing re-randomization periods which depend on the cache size and the desired security level. Further analysis of index randomization [BDY⁺20] showed that complexity trade-offs between the profiling and the attack phase are feasible. By observing the probability distribution of cache hit and miss events, it is even possible to distribute the attack over multiple re-randomization periods, thus limiting the effectiveness of re-keying. To counter the new attack vectors, [TNF⁺23] proposes a combination of cache decay and index-randomization while [SQ21] adds a layer of indirection to the cache lookup and thereby emulates a fully associative cache without needing to search the entire cache upon lookup.

Another approach at randomization is taken by Random Fill Cache [LL14]. Liu and Lee utilize the spatial locality of cache accesses and instead of caching data upon access, data from adjacent memory addresses is cached. This effectively hides the accessed memory addresses and therefore prevents leakage of this very information.

Randomization in TLBs. While the concept of index-randomization is easily transferable to TLBs, the characteristics of TLBs make for some interesting possibilities and limitations. In [Sez04, PTSM15, Sel04], skewed-associative TLBs are used to support multiple page sizes in a single TLB and hence, gain performance advantages. However, since the mapping is not randomized, these designs do not affect the security against side-channel attacks. From a security perspective, the most important observation is that TLBs are much smaller than the main data caches. For example, the reasonably recent Intel Skylake microarchitecture features TLBs with 64 (4 ways) and 1,536 entries (12 ways) for the first level data TLB and the second level TLB respectively [Int16a, Tab. 2-5]. In comparison, the Last Level Caches (LLCs) for which the cache randomization schemes are intended, commonly feature

several hundred thousand entries which greatly increases the randomization space. Pure randomization schemes like ScatterCache and PhantomCache will not yield a sufficient security level for TLBs without very frequent re-randomization since very few addresses need to be accessed to observe the first collisions in the TLB. In addition, benign programs usually cause very few TLB collisions in a short time period due to spatial locality. However, for an attacker it is trivial to cause a large number of collisions in the same time frame, by accessing addresses from several pages. Hence, the re-randomization threshold would have to aim for the attacker capabilities rather than the characteristics of benign programs which renders this approach highly inefficient for TLBs.

The second proposed design for a side-channel secure TLB by Deng *et al.* [DXS19] transfers the idea of Random Fill Cache [LL14] to TLBs. However, since TLBs store page translations which usually refer to 4 kB memory chunks, an access to a certain page is not necessarily followed by an access to an adjacent page. In fact, the opposite behavior is true, i.e., an access to a certain page will likely be followed by more accesses to *the same* page. Hence, not storing that translation will result in degraded performance. The performance of *RF TLB* is therefore highly dependent on the range, random pages — which also includes the original page — are loaded from. A smaller range will lead to a higher probability that the requested page will be loaded, but degrades security. A larger range increases security, but decreases performance. Furthermore, modifications to the page table are necessary to provide dummy pages, if not enough real (e.g., user- or kernel-owned) pages exist to pick from. The authors measured a 9% performance degradation in their experiments compared to set-associative TLBs.

5 A Side-Channel Secure TLB Design

After finding that neither partitioning, nor current randomization schemes are a good fit for a side-channel secure TLB design, we present TLBCOAT (TLB Countermeasure against Attacks based on Timing), a novel timing side-channel resistant TLB architecture. TLBCOAT combines the strengths of the previously discussed measures while ironing out the TLB-specific weaknesses which are mostly due to the small size. For our security analysis, we consider recent attacks on randomized cache architectures including PRIME+PRUNE+PROBE [PGGV21]. Our design builds upon the well-established index-randomization approach and adds per-process domain separation. We modify the randomization mechanism to be well-suited for typical TLB access patterns and introduce per-process re-randomization thresholds that minimize the performance impact of re-keying while ensuring that an attacker cannot gather sufficient information to perform efficient attacks.

5.1 TLBcoat in a nutshell

To efficiently exploit timing side channels in TLBs, the attacker needs to be able to reliably evict entries from the TLB. While straightforward randomization of the address-to-set mapping prevents attackers from naively constructing eviction sets, the aforementioned attacks [PGGV21, BDY⁺20] create probabilistic eviction sets by profiling the address-to-cache mapping. Especially given the small size of TLBs, these attacks are even more relevant than in the traditional cache scenario.

TLBCOAT leverages the usually very high hit-rate of TLBs. The reason for the high hit-rate is the spatial locality of software, meaning that an access to an address will likely be followed by an access in close proximity which is mapped within the same page. This holds for data, e.g., when accessing arrays or stack variables, and also for instructions since instructions are subsequent in memory and branches often have only a small branch offset. Since TLB conflicts are the only potential opportunities for an attacker to learn secret information, TLBCOAT aims to limit conflicts below a threshold before changing the

mapping, such that the attacker cannot perform the necessary profiling of the TLB before re-randomization is initiated. Therefore, TLBCOAT implements a per-process miss-counter that triggers re-randomization solely for that process if the threshold is reached.

To maintain the traditional access performance, TLBCOAT uses a randomized set-associative design to store the address translations. We use a low latency randomization function that takes the virtual address as input and returns one index in each way at which the translation may be stored. The exact instantiation of the randomization function is arbitrary. Several related works [TNF⁺23, SQ21] use variants of the lightweight block cipher PRINCE [BCG⁺12]. The latency of the randomization function can likely be further reduced using a purpose-built function. To provide a unique mapping for each process, the randomization function in TLBCOAT is tweaked by the PCID and a randomization value *rid*. This allows efficient per-process re-randomization without affecting other processes as shown in Figure 1.

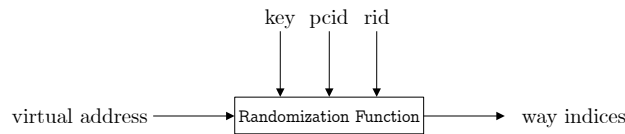


Figure 1: Randomization function of TLBCOAT which uses a global processor key and process-related information to generate possible positions inside the TLB.

Implementing complex replacement policies for randomized set-associative structures is not trivial since there are an exponential number of set-combinations for each entry. Hence, replacement policies like LRU would require a replacement-controller that knows the age relation for any entry in a way to any other entry in all the other ways. Since this is not practical in real-world hardware implementations, we approximate the behavior of LRU with a new replacement policy dubbed Randomized-Pseudo-Least-Recently-Used (RPLRU) described later in this paper.

The most important factor for the security of TLBCOAT is naturally the miss threshold used for re-randomization. This threshold depends on the size and associativity of the TLB and is further investigated in Section 6.3. The process specific miss counter (TLBCNT) is initialized to the threshold value. On every TLB miss, the counter is reduced by one. When the counter reaches zero, the *rid* is set to a random value and the TLBCNT register is reset to the threshold value. Since the mapping function changes, the previously stored translations are with high probability no longer found on a lookup which effectively flushes the entries from the current process in the TLB. A general overview of the whole system is shown in Figure 2.

TLBCOAT is minimally intrusive in software and only requires a few changes in the OS to function properly. The state of a process now also includes the TLBCNT register and a register holding the *rid*, which must be saved in the Process Control Block (PCB) upon a context switch. Both must only be read-write accessible from machine mode to prevent unprivileged code from changing them or extracting information. Therefore, neither the programmer nor OS have to identify pages or processes that require protection and tag them. Unlike *RF TLB* and *SP TLB* [DXS19], TLBCOAT does not require a differentiation between secure and insecure processes and is therefore capable of executing mutually untrusted code.

5.2 Design Rationale

In the following, the design of TLBCOAT is presented in more detail and we provide reasoning behind the choices made in the design process. First, we describe the randomization of the mapping function and then we discuss the counter-based re-randomization mechanism.

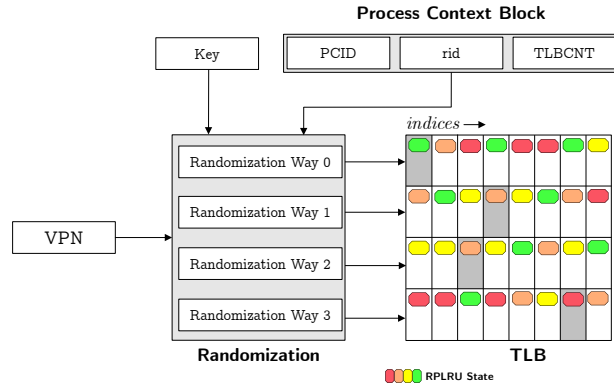


Figure 2: An example of TLBCOAT showing the added randomization module. The randomization is done for every way, which may require multiple parallel instantiations of the randomization function.

5.2.1 Randomization

Randomizing the address mapping is a common design feature of side-channel secure cache architectures; see [WUG⁺19, TZBR21, WL07]. Recent attacks [PGGV21, BDY⁺20] have challenged the assumption that index-randomization alone is sufficient to prevent timing side-channel attacks. However, index-randomization still increases the attack complexity. There are more recent randomization-based architectures that maintain security in the face of the new attacks; see for example [TNF⁺23, SQ21]. However, the dynamic Time to live (TTL) scheduling mechanism proposed by Thoma *et al.* [TNF⁺23] does not scale for very small designs like TLBs since a few accesses are sufficient to fill the entire structure. On the other hand, the work presented by Saileshwar and Qureshi [SQ21] adds complexity on the critical path of the lookup which does not suit the access characteristics of TLBs which need to be accessed very frequently.

TLBCOAT uses a keyed randomization function that takes the virtual address minus the page offset bits² as input and returns an index in each way of the set-associative TLB. The randomization function is further tweaked by a PCID and *rid*, the details on which we will cover later in this paper. Formally, the function can be described as $f_k : (a, t) \rightarrow \{i_0, \dots, i_{w-1}\}$ where a is the address and $t = (PCID, rid)$ is the tweak in a w -way set-associative TLB. We further require that the indices in each way are independent, i.e., given an address mapping $f(a) = \{i_0, \dots, i_{w-1}\}$, there is no polynomial attacker A that can predict the mapping of an address $b \neq a$ with $Pr[(x, i_x) \leftarrow A(a, i_0, \dots, i_{w-1}) : f(b)[x] = i_x] \geq \frac{w}{N} + negl(n)$. We propose using a lightweight block cipher for the randomization function. Since the attacker cannot observe the output of the randomization function, a full block cipher is not required from a security perspective allowing the use of round reduced variants as proposed in [TNF⁺23]. Purpose-built randomization functions like Scarf [CGL⁺22] could further reduce the latency and the area overhead.

Since TLBs are virtually indexed, it is necessary that the randomization function yields different results for different processes to prevent trivial collisions by mirroring the virtual addresses used by the victim. By tweaking the randomization function with the PCID, we make sure that each process experiences a different address mapping behavior. This also prevents cross process profiling attacks since the attacker cannot learn the mapping function of another process by observing unrelated processes.

As stated earlier, especially in small TLBs, randomization cannot prevent an attacker from learning conflicting addresses and using them to perform, e.g., PRIME+PRUNE+PROBE attacks [PGGV21]. To thwart this, we must frequently change the randomization

²Addresses which differ only in the offset must be mapped to the same index for a functional TLB.

mapping. Therefore, we now describe how the re-randomization is performed. From a performance perspective, changing the address mapping is similar to a TLB flush. Though the entries still reside in the TLB, they are with high probability not found upon lookup since the address mapping will return different indices. To minimize the performance effect of frequent re-randomization on processes that simultaneously share the TLB, we perform the re-randomization on a per-process basis using the randomization value *rid* in the tweak. Every time a re-randomization is triggered, the *rid* is set to a new random value which by the design of the randomization function changes the mapping without affecting other processes. While it would be sufficient to increment the *rid* on each re-randomization, assigning a random value prevents the attacker from keeping track of the ID and predicting when the counter wraps around and hence, when the randomization function repeats.

5.2.2 Re-Randomization Counter

A unique feature of TLBs relates to context switching. We observe that during a single scheduling interval of a typical process, only a small number of pages are accessed. On processors that do not use PCIDs, the TLB is flushed entirely on each context switch. In this case, changing the randomization function is effectively free since the next time the process is scheduled, all entries will be populated from scratch. However, more and more processors implement PCIDs since it makes TLB flushes on context switches unnecessary. This also implies that entries may survive scheduling intervals of other processes and hence, the context switch no longer presents a free opportunity for changing the address mapping. A further problem that arises when using context switches for re-randomization is, that while benign applications usually do not encounter a large number of TLB conflicts during a scheduling interval, malicious applications can easily trigger such large number of conflicts in a small time interval, and thus may be able to complete an attack before a context switch is initiated.

Therefore, TLBCOAT provides a different measure to initiate re-randomization that is based on the number of conflicts experienced. Each process is assigned a counter (TLBCNT) that is decremented every time a TLB conflict occurs. The initial value of the counter value is thereby dependent on the TLB size. We will explore the limit as a function of the TLB size in Section 6.3. When the counter reaches zero, the *rid* is reset to a random value, initiating the re-randomization. The TLBCNT register is reset to the threshold value to restart the randomization procedure. Intuitively, it might seem beneficial to set TLBCNT to a random value in a range close to the threshold to hide the exact time when *rid* is updated. However, this does not add any security benefit since attackers can easily probe their own process for a re-randomization by alternately accessing a known cached page and a new unknown page. Importantly, attackers can only do this with their own randomization function, not with other processes.

5.2.3 Replacement Policy

Randomized designs pose new challenges for replacement policies since each entry can be combined in a large number of combinations with other entries to form a set. The simplest replacement policy would be to randomly replace one of the candidate entries. However, most deployed cache and TLB architectures use (pseudo-) LRU to replace entries as it reduces the miss rate and therefore increases the performance compared to random replacements [AMM04]. Implementing (pseudo-) LRU for randomized architectures is not trivial since the replacement policy must be able to determine the least-recently used entry within each random set of entries. This could be achieved by storing timestamps for each entry. However, the hardware overhead of such a solution would be immense.

For TLBCOAT we therefore designed RPLRU, a new replacement policy well suited for randomized set-associative architectures that approximates LRU. An overview of RPLRU is shown in Figure 3. In a nutshell, the entries are assigned with an age indicator that

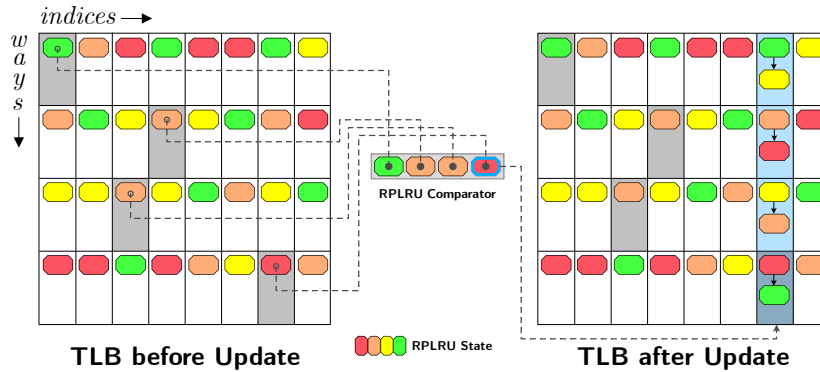


Figure 3: Overview of the Randomized-Pseudo-Least-Recently-Used replacement policy.

gives their respective age compared to other entries with the same index. The indexing function selects w entries in which the translation information of the accessed page can be stored. In the figure, these entries are colored gray. To select the entry that will be replaced by the access, the relative ages of the candidate entries are compared and the oldest entry is selected for replacement. If several entries share the greatest age, one of those entries is randomly selected for replacement. The age indicator is only updated at the index where the new entry was inserted. That is, the new entry becomes the most recently used and the others are adjusted accordingly. Since the replacement policy prioritizes the eviction of entries that have less recently been touched, it can be seen as a form of Pseudo-Least-Recently-Used (PLRU). Opposed to many traditional PLRU implementations that use a pointer-based approach (Tree-PLRU), RPLRU still requires storing age indications alongside the entries. However, these only need to indicate the age respective to the other entries with the same index and can therefore be very small.

The complexity of RPLRU is comparable to a PLRU implementation with $\log_2(w)$ status bits per entry. The candidate entries are selected by the randomized indexing function. When a TLB hit occurs, the replacement information is only updated in the entries that share the same index — i.e., what would be a set in the non-randomized setting. Hence, in this case, RPLRU behaves exactly the same as the $\log_2(w)$ -Bit PLRU. If a TLB miss occurs, the replacement policy needs to select one of the candidate entries to be replaced. Therefore, the age indicators of the candidate entries need to be compared. Since it may happen that multiple entries share the highest age, the replacement policy must then select one of those entries randomly. Finally, the updating of the replacement information of the replaced entry is exactly the same as in the hit scenario. The additional comparison occurs only in the miss case which is not as timing critical as the hit case. That is, since the translation needs to be fetched from another TLB level or the page table anyway. This can be done in parallel. Hence, overall the complexity is similar to $\log_2(w)$ -Bit PLRU.

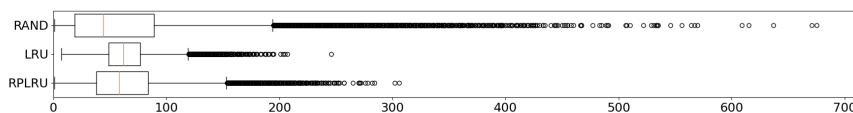


Figure 4: Distribution of the number of TLB misses until the eviction of the target address is achieved for random vs. LRU vs. RPLRU replacement policy. Simulated 4-way set-associative TLB with index randomization and 64 entries. 50,000 traces recorded for each policy.

The incentive to approximate LRU for TLBCOAT goes beyond performance reasons. Figure 4 depicts the number of TLB accesses until a given entry is replaced for each replacement policy. It is clear that random replacements reduce the average number of accesses required to evict the given entry. LRU significantly narrows the distribution and increases the average. This facilitates a higher threshold for re-randomization in TLBCOAT. While RPLRU does not quite match the distribution of LRU, it does significantly improve over the random replacement policy. We discuss the security implications of RPLRU in Section 6.3.

6 Evaluation

In this section, we evaluate the security and performance of TLBCOAT. We first describe our evaluation setup and demonstrate the leakage of current set-associative TLBs in our simulation environment. Then, we discuss how TLBCOAT defends against state-of-the-art attacks and show that the leakage found in classical TLBs no longer exists. Finally, we benchmark TLBCOAT using the PARSEC benchmark suite and analyze the hardware overhead.

6.1 Evaluation Setup

For the evaluation, we use two different simulation tools which are described in the following. Both the functional simulator as well as the `gem5` source code are provided online³.

Functional Simulation. Since the security parameters of TLBCOAT are dependent on many factors of the TLB, including the number of available sets and the replacement policy, we developed a purely functional simulator of TLBCOAT in Python. Precisely, the software simulates the address to TLB-entry mapping and replaces the entries based on the chosen replacement policy which can be chosen either as random, true LRU, or the new RPLRU. Addresses are represented as 64 bit integers which allows generating new addresses randomly without needing to worry about conflicts. For the index randomization, we use Python’s `hashlib` which enables faster simulation compared to simulating a, e.g., PRINCE-based [BCG⁺12] randomization function. Since the objective is to simulate a random mapping, for the functional simulator these methods are interchangeable. The lower bits of the resulting hash digest are used as the index in the given way. This ensures an efficient pseudorandom address to TLB mapping. Opposed to full CPU simulators, our simulation tool gives easy access to important data, like the candidate entries for a given address, which provides more detailed insights in relevant design decisions. However, note that this simulator is explicitly not designed for a performance analysis.

Timing-Accurate Simulation. For the performance evaluation, we implemented TLBCOAT in `gem5`, which allows us to simulate a real-world RISC-V Linux system without the need to modify the actual Linux kernel. Additionally, we can check its security, see Section 6.3, and its performance in a highly controlled setting. We base our implementation on Intel’s vulnerable 4-way 16-set set-associative configuration and build TLBCOAT on top of it. We added a 3-round PRINCE [BCG⁺12] variant as the randomization function as well as registers holding the global encryption key, the miss count (`TLBCNT`) and the `rid`. Both the `TLBCNT` and `rid` are emulated internally in the TLB module as 32-bit arrays with a capacity of 65,536 elements (corresponding to the 16-bit ASID on RISC-V). Since the ASID is provided by the software for each translation, we can efficiently store this information in `gem5` without modifying the process state in Linux. The re-randomization threshold is set to 64 in accordance with our results in Section 6.3.

³Available at <https://github.com/Chair-for-Security-Engineering/TLBCoat>.

6.2 Demonstrating Leakage in gem5

To demonstrate the security of TLBCOAT, we used `gem5` to implement both Intel’s vulnerable TLB as described by Gras *et al.* [GRBG18] and TLBCOAT and tested them using a Linux OS. Unlike the original attack by Gras *et al.* [GRBG18], we cannot simulate a SMT environment. However, Deng *et al.* [DXS19] demonstrated via bare-metal benchmarks on RISC-V that SMT is *not* a requirement if PCIDs are employed. In this section we show that Intel’s TLB design leaks spatial information on a RISC-V Linux.

We simulated a dual-core 64-bit RISC-V system using the SV39 page-based virtual-memory system [WLA⁺16, p. 57], based on the *HiFive* platform [SiF]. As the side channel is only dependent on the state of the TLB and the interactions with the memory subsystem, we decided to use the *TimingSimpleCPU* model, which models a single-issue CPU without any pipelining, but simulates all memory interactions. The TLB was modified to mimic Intel’s 16-set 4-way set-associative configuration. As seen in Table 1, set-associative TLBs are widely used even beyond Intel processors. We use the recent Linux 5.12 kernel, which introduces ASID support for RISC-V. Using kernel boot parameters, we isolated one of the cores to not be considered by the OS scheduler, allowing us to manually start the victim and attacker process in a low-noise environment.

A victim process is spawned accessing a specific page in a loop. As we know what page is accessed by the victim, we can calculate the corresponding set and create an adversary process that accesses specific pages which create an eviction set. For example, in our scenario and assuming 4 kB pages, the set is selected by calculating $address \gg 12 \bmod 16$ as the lowest 12 bits are the same for both the physical and virtual address. Therefore, the adversary can trivially map pages to addresses which will be placed in the same set as the victim’s page. The required knowledge for this attack is in line with the capabilities stated in Section 3 and was demonstrated by Gras *et al.* [GRBG18]. After allocating pages that form an eviction set, a tight loop is entered which first accesses addresses within these pages and measures the time to complete this task. Afterwards, the adversary yields the execution, which forces the scheduler to switch to another process. Once the adversary process continues, it starts at the beginning of the loop again. In our experiment one measurement consists of 5,000 repetitions, which are used to calculate an average. We observed that the access time will be higher, if the victim accessed a page inside the eviction set. If no access occurs, the access time will be lower. In our scenario, a TLB miss caused a delay of 1–3 cycles. To confirm the presence of a side channel we additionally performed Welch’s t-test in a *fixed vs. fixed* configuration [DS16]. For this test we performed 1,000 measurements on a victim, which *does not* access a page inside the eviction set, and on a victim, which *does access* a page inside the eviction set. As seen in Figure 5a, for traditional set-associative TLBs the *t*-value quickly rises above the threshold of 4.5, which generally indicates a leakage. *t*-values above 4.5 correspond to confidence levels > 0.99999 to *reject* the null-hypothesis, i.e., that the two measurement sets are from the same population which would mean that no side channel exists [SM15]. We are thus able to infer spatial information about the victim.

6.3 Security Analysis

In this section we evaluate the security aspects of TLBCOAT.

Absence of leakage. TLBCOAT utilizes index-randomization which is also a part of many proposed side-channel secure cache architectures. In TLBCOAT, the randomization is unique to every process due to the PCID and *rid*, which are part of the input to the randomization function. Thus, our approach makes it impossible for an adversary to construct trivial eviction sets by mirroring the virtual addresses used by the victim process or by constructing addresses that map to the same index in the TLB across processes. This implies that traditional PRIME+PROBE attacks are no longer feasible, as they rely

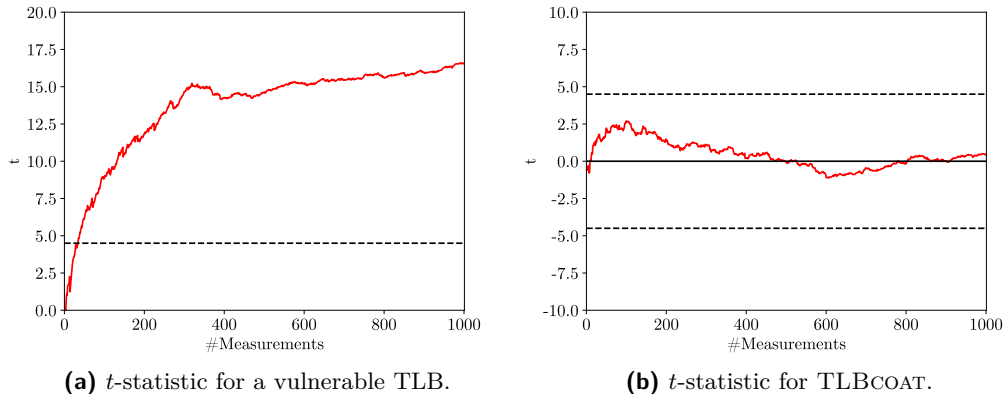


Figure 5: Leakage assessment for set-associative TLBs and TLBCOAT using a set of congruent addresses, i.e., aligned on the lower 12 bits.

on the deterministic mapping function. Therefore, TLBCOAT does not only inherently defend against TLBLEED [GRBG18], but also against all potential attacks discovered by Deng *et al.* [DXS19]. We confirmed that TLBCOAT defends against PRIME+PROBE by rerunning our experiment shown in Section 6.2. As seen in Figure 5b, our side-channel analysis via a t -test did not indicate significant leakage, as it was not possible to predict where the victim translation is placed inside the TLB.

Assembly of Eviction Sets. To this date, the best-known attack against randomized cache architectures is PRIME+PRUNE+PROBE [PGGV21]. Since the attack surface of randomized caches and TLBs is similar, PRIME+PRUNE+PROBE is the most relevant attack against TLBCOAT. The attack generally divides in two phases: First, a generalized eviction set is constructed by extensively profiling conflicts, and secondly, the eviction set is used to perform a PRIME+PROBE-like attack. By selecting an initial set of priming addresses that each map to a different page, the TLB can be populated with attacker controlled addresses for most parts (prime). Since it may happen that some translation entries within the set evict each other, the priming set needs to be re-accessed until no more TLB misses occur (prune). After pruning, the attacker triggers the access to the victim. With some *catching* probability, the access performed by the victim replaces one of the attacker’s page translation entries. By accessing all of the addresses from the priming set, the attacker can learn a conflicting page translation (probe). The attacker needs to repeat these steps several times to gather several such conflicting translations which form a generalized eviction set.

In the following, we use our functional simulator to analyze the number of TLB misses that occur during the profiling phase of PRIME+PRUNE+PROBE to establish a secure re-randomization threshold. After changing the randomized mapping by updating the *rid*-value, the progress of PRIME+PRUNE+PROBE is reset since collisions found under the previous mapping will no longer lead to conflicts with the new mapping. Hence, we find that the minimum requirement for a successful attack is the creation of a sound (not necessarily minimal) eviction set within one randomization period, i.e., the addresses that form the eviction set must be able to occupy all candidate entries of the victim page in the TLB. To establish the upper bound for the re-randomization threshold of TLBCOAT, we measure the number of TLB misses that occur during the profiling phase until a functional generalized eviction set is found. To ensure that our results represent the best-case scenario for the attacker, we give the attacker two pieces of additional information that would not be available in a real-world attack. First, we assume that the attacker knows when the

eviction set is functional, i.e., when it can potentially occupy all candidate entries of the victim page. Second, we give the attacker the information whether the victim translation is currently stored in the TLB or not. A real-world attacker would not have access to this information and would hence, need to guess. This will further increase the miss count since more iterations of the profiling phase are required if the attacker’s guess is incorrect.

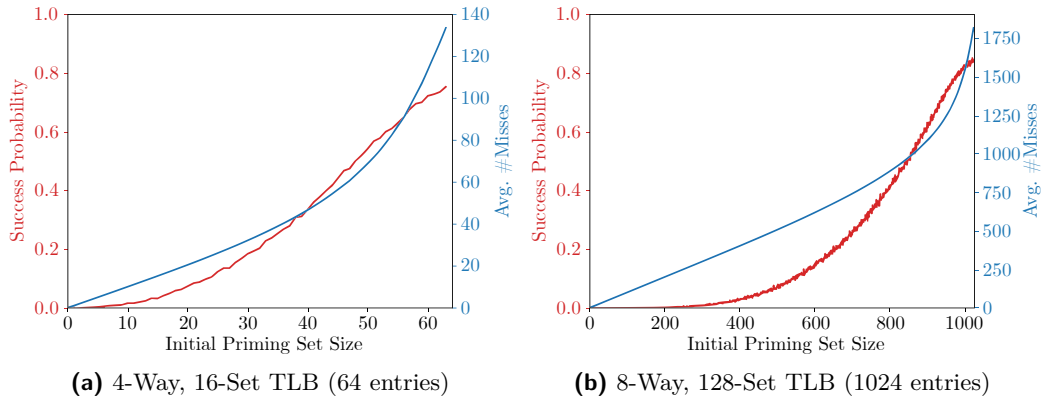


Figure 6: Catching probability and average number of misses for the prime and prune steps as a function of the initial probing set size for finding one translation conflict using the PRIME+PRUNE+PROBE attack. The modeled TLB uses the RPLRU replacement policy. It implements index-randomization but does not re-randomize the mapping during the attack.

Using our functional simulator, we first evaluate the dependency between the initial priming set size, the probability of observing a conflict when accessing the victim page (catching probability), and the number of misses occurring during that process. Therefore, we start by selecting a random target address which is not yet stored in the TLB. This represents an address in the victim page. Then, a set of random priming addresses is assembled and accessed which causes the translations to be stored in the TLB. The priming set is accessed until no more conflicts within the set occur (i.e., upon access, all addresses result in a TLB hit (prune step)). If prune yields repetitive misses due to conflicts within the priming set, some of the conflicting entries are dropped from the set. Then, the target address is accessed. If the access evicts an attacker controlled entry from the TLB, the profiling iteration is successful since a conflicting address has been identified. Otherwise, the profiling iteration is unsuccessful. The procedure is repeated 10,000 times for each initial priming set size. By simulating only the I/O behavior of the TLB, the results are independent of any software or benchmark that would be executed on a processor. The results of this analysis on a 4-way, 64 entry TLB (mimicking Intel’s L1D TLB) are shown in Figure 6a. For a typical L2 TLB with 1024 entries, the results are shown in Figure 6b. Naturally, the catching probability increases with a larger initial priming set. However, the TLB misses also increase with the size of the initial priming set. For very large priming sets, the number of misses increases exponentially since translations within the priming set are more likely to evict each other. This increases the complexity of the prune phase. For the TLB configuration with 8 ways, the point of exponential growth starts later than for the smaller configuration with 4 ways. We discuss the effect of associativity on the security of TLBCOAT in Appendix A.

The intuition is that if the first iteration of PRIME+PRUNE+PROBE profiling causes k TLB misses with catching probability p_c , each further iteration will result in a similar number of misses. Hence, the overall number of misses would be $w * k$ and the probability of success would be p_c^w . However, we found that the number of misses can be drastically

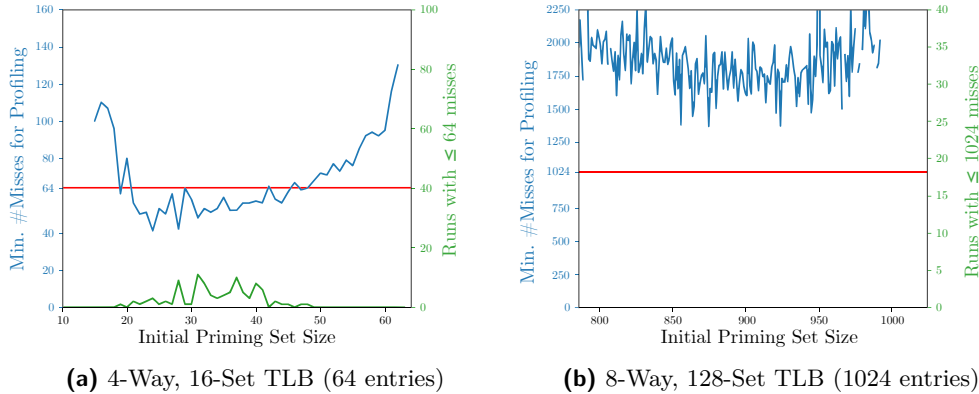


Figure 7: The blue line depicts the minimum experienced misses to obtain a functional probabilistic eviction set during profiling of PRIME+PRUNE+PROBE over 100,000 trials for each initial priming set size. Assuming a re-randomization threshold of N (the number of entries available), the green line depicts the total number of successful runs out of the 100,000 iterations.

reduced by optimizing the profiling. In order to minimize the TLB misses that occur during the profiling, we introduce some modifications to the algorithm. As shown in Figure 6, the initial priming of the TLB causes a lot of misses since each of the addresses must first be brought into the TLB. These initial misses cannot be avoided. After accessing the victim page and probing the prime set, the victim page must be removed from the TLB to allow starting the next iteration. A naive way of removing the victim translation would be to access new random pages that will at some point evict the victim. However, this would lead to a massive increase of TLB misses. Instead, we do three steps to increase the chances of evicting the victim without causing many misses. First, we access all addresses that are already part of the unfinished eviction set and therefore, known to collide with the victim. After that, we access all addresses from the priming set and hence make sure that they are all stored in the TLB. Since many of these address translations are still stored, only few misses occur. In many cases, the victim translation is already replaced after these two steps. If it is not, it is likely that other entries of the set have been accessed and hence, the victim is the least recently used entry. If the victim is not removed yet, we start by replacing one address from the priming set with a new random address. Then, the whole priming set is accessed, resulting mostly in hit accesses. This process is repeated until the victim is finally removed from the TLB. Then, the attack continues with the next iteration. Since we already accessed the priming set repeatedly to evict the victim, the prime and prune step are usually not required for the second iteration and result in hit accesses only. The modified algorithm is shown in Appendix B.

We now assume a threshold for TLBCOAT that matches the number of available entries in the TLB. This threshold is chosen as a rough estimation of how many accesses are required to construct a minimal eviction set in the best case, based on the observations made in Figure 6. In Figure 7, we plot the minimum number of TLB misses that occurred over 100,000 trials for each priming set size (blue line). For the evaluation we use a small 4-way set-associative TLB with 64 entries (Figure 7a) and a large TLB with 1,024 entries and 8 ways (Figure 7b). The profiling stops as soon as a functional eviction set is obtained, i.e., the set of addresses can populate all possible locations of the victim translation. While the average number of misses during the profiling is much higher, the relevant metric is the lower bound since the attacker can restart the profiling many times.

For our small evaluation configuration, it shows that the ideal priming set size is

between 20 and 40 addresses. During our profiling, the minimal number of misses occurred until a functional eviction set was constructed is about 40. The green line in Figure 7a depicts the number out of 100,000 attempts where the total number of misses was below the threshold of 64. Of all the $64 * 100,000$ profiling attempts, less than 0.01% resulted in fewer than 64 misses. We extend our analysis by setting the initial priming set size to 24 which led to the best results in Figure 7a. We run the profiling 1,000,000 times. Only 50 of these profiling attempts resulted in a success which corresponds to a profiling success rate of about 0.005%.

For the larger TLB, the results are even more clear. No profiling attempt created a functional eviction set with less than 1,300 misses which lies above the re-randomization threshold. Hence, none of these attempts would have led to a successful profiling. We found that for very small initial priming set sizes, the profiling did not yield functional eviction sets. That is, since the probability that any set of addresses occupies a randomized target set increases with the size of the set.

In our experiment, the attacker is artificially provided with the information if the probabilistic eviction set is theoretically functional. In a real-world environment, the attacker cannot know this without testing it. However, testing the eviction set for functionality causes even further TLB misses, accelerating re-randomization. Dropping the ideal assumptions — i.e., accounting for noise and not giving the attacker additional information about the state of the profiling — it is clear that the attacker cannot perform a successful attack below the threshold.

Replacement Policy. Since LRU is not a practical replacement policy for randomized caches, we designed RPLRU which approximates the behavior of LRU. Therefore, we must make sure that the new replacement policy does not facilitate efficient replacement of specific entries. This would be the case, if an attacker could increase the probability of evicting the target entry on access. To achieve this, they would have to increase the relative age of the target entry compared to those entries that share the index in the TLB. However, due to the index randomization, it is not possible for the attacker to simply choose such addresses that share the index with the target translation. Obtaining such addresses requires the same profiling steps as constructing an eviction set. We have shown above that this is not feasible below a given threshold value.

6.4 Performance

We validated our design using the `gem5`-adapted version of the PARSEC 3.0 benchmark suite by Peter Yuen [Yue]. Note that at the time of writing, not all benchmarks could successfully run on RISC-V, because of, for example, compilation errors or problems interacting with the `gem5` internal statistics framework. Thus, our evaluation subset excludes all non-working benchmarks. We recorded the total number of accesses to the

Table 2: PARSEC benchmark results comparing regular set-associative TLBs to TLBCOAT by their respective miss ratio. Each benchmark is the only program running on the system apart from some background tasks. Benchmarks marked with a * were run using the *simsmall* input, the others with *simmedium*.

Benchmark	Set-Associative TLB LRU	TLBCOAT RPLRU	TLBCOAT LRU
blackscholes	10.99%	0.02%	0.02%
canneal	7.20%	9.20%	8.73%
dedup	0.05%	0.04%	0.04%
fluidanimate*	0.41%	0.45%	0.44%
freqmine	0.17%	0.23%	0.21%
streamcluster*	0.22%	0.61%	0.21%
swaptions	<0.01%	<0.01%	<0.01%

TLB, the number of misses as well as re-randomization requests in case of TLBCOAT. All benchmarks were run on a simulated single core Linux 5.12 system.

Table 2 shows the results for a system without any additional processing intensive load. Each run was supplied with the *simmedium* input except for two benchmarks, which did not successfully finish because of processing and memory constraints. It demonstrates that TLBCOAT only slightly affects the TLB miss-rate, and hence, the overall performance. On the one side, we were also able to observe performance increases for some benchmarks such as *blackscholes*. These can be explained by suboptimal access patterns for the set-associative implementation. If a set of pages, which is greater than the available ways but all mapping to the same internal TLB set, are repeatedly accessed this will inevitably lead to evictions. TLBCOAT solves this problem by breaking the static structure up and selecting different ways for each requested page. On the other side, some benchmarks experienced a slight performance decrease. This may happen, as addresses which would normally not compete for the same set in standard TLBs may now do so because of the randomization. Furthermore, re-randomizations can also be triggered by normal program behavior, which is then equivalent to an unnecessary TLB flush for a specific process. On average a re-randomization is triggered after 665,219 accesses. The miss rate of TLBCOAT with LRU is only slightly increased with 0.43% of the accesses resulting in a TLB miss compared to 0.41% for a set-associative TLB. The impact to the overall runtime of the benchmarks is therefore negligible.

In addition to the evaluation on the idling system, we tested TLBCOAT on a system running an additional workload by executing the *blackscholes* benchmark in an endless loop in the background. *Blackscholes* was chosen because it is the first one alphabetically. It would have also been possible to choose any other load. Then, we started different benchmarks and measured the overall TLB miss rate, which is now affected by the TLB usage of the background application. We choose the miss rate as a metric since it allows directly comparing the TLB performance. Using the cycle count, it would be harder to actually draw conclusions about the TLB performance since running two benchmarks on a single core obviously increases the run time and the more or less random scheduling decisions can impact the results severely.

Table 3 shows the results for a system under load. Each run was supplied with the *simsmall* input. We note that only a subset of all benchmarks ran successfully, because of memory as well as processing constraints. The results indicate that the overall miss rate is similar to the results shown in Table 2. That is, since most of the benchmarks only rely on a few pages that remain cached even with two benchmarks running. Moreover, it shows that the miss rate remains similar for set-associative TLBs and TLBCOAT. The reason for this is that the re-randomization is done per process by changing the *rid* and therefore, a high miss rate in one process does not cause re-randomization in the other process.

6.5 Hardware Requirements

We now evaluate the hardware requirements for TLBCOAT. The exact values depend heavily on a multitude of parameters, including the technology, and the chosen TLB configuration, e.g., the number of entries and ways, and the randomization function. In

Table 3: PARSEC benchmark results for system under load comparing regular set-associative TLBs to TLBCOAT by their respective miss ratio. The system constantly executes the *blackscholes* benchmark in the background to generate noise.

Benchmark	Set-Associative TLB LRU	TLBCOAT RPLRU
blackscholes	0.021%	0.028%
dedup	0.046%	0.055%
canneal	3%	3.7%
freqmine	0.21%	0.30%

any case, TLBCOAT requires 3 new registers: the key register, TLBCNT and the *rid*. We also require a source of randomness for re-randomization via the *rid* and optionally for the key register, as the processor specific key may be randomly set after each reboot. We note that modern desktop- and server-grade CPUs usually integrate a True Random Number Generator (TRNG) [Int18] that can be used for this purpose. Furthermore, to keep the latency of TLBCOAT low, the randomization function should ideally complete in one clock cycle, which requires an unrolled implementation.

To evaluate TLBCOAT in practice we implemented TLBCOAT on an Field Programmable Gate Array (FPGA) by extending a standard set-associative design. A stand-alone implementation allows us to observe our TLB's functionality and debug errors. In the following, we briefly describe the structure of TLBCOAT as shown in Figure 8. First, the cache controller loads the virtual address into the address register. Afterwards, the randomization function determines the set for each way. The output of each way consisting of the data, tag and set is then forwarded to the comparator where the selected tags are compared to the expected tag. Depending on the comparison result either the `hit` or `miss` signal is asserted. The output of the comparator is also used to select the correct RPLRU set and way for the age update if the `hit` signal is active. In case of a miss, the cache controller can manually provide the data to write, as well as the set and way. When the miss signal is asserted, the cache controller can read the `sets` output to determine which sets to query for the age of the respective way.

After verifying the correctness of our implementation in simulation and on the FPGA, we synthesized TLBCOAT with a 4-way 16-set configuration using the Synopsys Design Compiler and Silvaco's Open-Cell 45nm and 15nm FreePDKs [Sil] to estimate the area overhead that would occur on a real CPU. The results are shown in Table 4. Most of the area is consumed by the storage elements which are included in the ways. Overall, the entries need to store various information for each of the 64 pages, such as the ASID or permission bits. TLBCOAT does not require modifications to the ways, but rather to the unit which selects the individual set in each way. Hence the four ways utilize the identical area as in a standard set-associative TLB. The randomization function, in

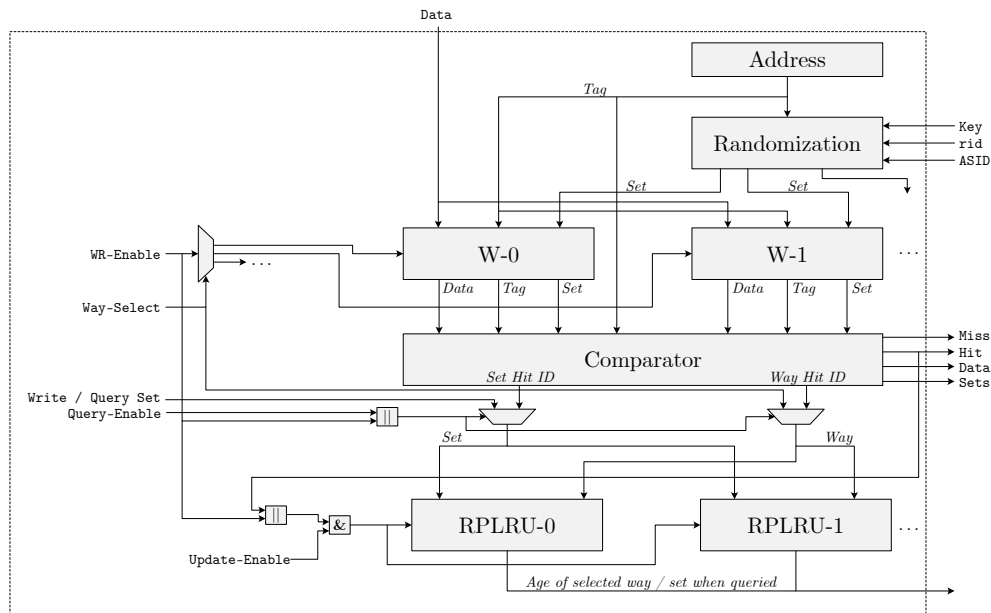


Figure 8: Simplified block diagram of the hardware implementation. For simplification the image shows a 2-set 2-way associative randomized TLB.

our case a 3-round **PRINCE**, represents one of the main modifications and only requires 2.93% (45nm) or 3.89% (15nm) of the overall area. The area required varies depending on the randomization function. In our case, we use the 64-bit output of **PRINCE** to index all ways by extracting four nibbles. Thus only one unrolled implementation needs to be placed in the cache. Furthermore, we store RPLRU information for each traditional cache set in a separate unit, which in sum take up 4.13% (45nm) or 5.07% (15nm) of the total area. Regarding the latency of **TLBCOAT**, [Table 4](#) shows that we only exhibit a delay of 143 ps (15nm) until the cache can determine a hit or a miss. In case of a hit, the corresponding RPLRU unit will be updated in parallel. In case of a miss, the cache controller can determine the respective ages of all selected entries and select the entry to be evicted while the L2 TLB or page table is queried. Intel or AMD do not publish the TLB latency for their processors. Additionally, the TLB latency overlaps with the latency for the L1 cache as the cache is virtually indexed and physically tagged, which requires a TLB lookup in parallel. Velten *et al.* [[VSIH22](#)] benchmarked the AMD EPYC 7702, which uses a 7nm process, and the Intel Xeon Gold 6248, which is manufactured in 14nm. The results show that the latency for the L1 cache is between 1.6ns and 2ns. Thus, **TLBCOAT** does not affect the overall latency or area significantly and is a competitive candidate for a side-channel secure TLB.

Table 4: Hardware overhead and time delay added by **TLBCOAT** for different manufacturing nodes.

	45nm	15nm
Randomization	1,493.66 GE	2,253.76 GE
4 Ways	45,951.87 GE	47,912.05 GE
16 RPLRU Units	2,104 GE	2,936 GE
Comparator	401 GE	501.5 GE
Other	953.71 GE	1,209.98 GE
Total Area	50,904.24 GE	57,813.29 GE
Total Delay	1.6 ns	0.143 ns

7 Conclusion

Timing side channels in CPUs have become a significant threat to the system security and we foresee that the relevance of these side channels will only increase in the coming decades. In this paper, we analyzed the current state of side-channel defenses on TLBs and uncovered that despite the architectural similarities, cache side-channel defenses are not necessarily suited to protect TLBs. We found that neither partitioning, nor current index-randomization proposals are good candidates to secure TLBs. Therefore, we presented **TLBCOAT**, a novel randomization-based TLB design that protects against state-of-the-art attacks including **PRIME+PRUNE+PROBE**. We demonstrated how **TLBCOAT** removes the side-channel leakage and renders attacks infeasible. Our analysis shows a minimal performance and area overhead compared to traditional set-associative TLBs.

Acknowledgments

We would like to thank all our reviewers as well as our shepherd Billy Bob Brumley for their constructive feedback and guidance during the submission process. The work described in this paper has been supported by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) through the project **RAINCOAT** (440059533), by the DFG through Germany’s Excellence Strategy - EXC 2092 CASA - 390781972, and by the German Federal Ministry of Education and Research (BMBF) through the project **FlexKI** (01IS22086I).

References

- [AAB⁺16] Krste Asanovic, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, et al. The Rocket Chip Generator. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17*, 2016.
- [AMD17] AMD. *Software Optimization Guide for AMD EPYC™ 7001 Processors*. Advanced Micro Devices, Santa Clara, California, United States, 55723, rev 3.00 edition, June 2017.
- [AMM04] Hussein Al-Zoubi, Aleksandar Milenkovic, and Milena Milenkovic. Performance Evaluation of Cache Replacement Policies for the SPEC CPU2000 Benchmark Suite. In Seong-Moo Yoo and Letha H. Etzkorn, editors, *Proceedings of the 42nd Annual Southeast Regional Conference, 2004, Huntsville, Alabama, USA, April 2-3, 2004*, pages 267–272. ACM, 2004.
- [BCG⁺12] Julia Borghoff, Anne Canteaut, Tim Güneysu, Elif Bilge Kavun, Miroslav Knezevic, Lars R. Knudsen, Gregor Leander, Ventzislav Nikov, Christof Paar, Christian Rechberger, Peter Rombouts, Søren S. Thomsen, and Tolga Yalçin. PRINCE - A Low-Latency Block Cipher for Pervasive Computing Applications. In Xiaoyun Wang and Kazue Sako, editors, *Advances in Cryptology - ASIACRYPT 2012 - 18th International Conference on the Theory and Application of Cryptology and Information Security, Beijing, China, December 2-6, 2012. Proceedings*, volume 7658 of *Lecture Notes in Computer Science*, pages 208–225. Springer, 2012.
- [BDY⁺20] Thomas Bourgeat, Jules Drean, Yuheng Yang, Lillian Tsai, Joel S. Emer, and Mengjia Yan. CaSA: End-to-end Quantitative Security Analysis of Randomly Mapped Caches. In *53rd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2020, Athens, Greece, October 17-21, 2020*, pages 1110–1123. IEEE, 2020.
- [Ber05] Daniel J Bernstein. Cache-timing attacks on AES. <http://cr.yp.to/papers.html#cachetiming>, 2005. Last accessed: 07/08/2022.
- [CBS⁺19] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtvushkin, and Daniel Gruss. A Systematic Evaluation of Transient Execution Attacks and Defenses. In Nadia Heninger and Patrick Traynor, editors, *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, pages 249–266. USENIX Association, 2019.
- [CGL⁺22] Federico Canale, Tim Güneysu, Gregor Leander, Jan Philipp Thoma, Yosuke Todo, and Rei Ueno. SCARF: A low-latency block cipher for secure cache-randomization. *IACR Cryptol. ePrint Arch.*, page 1228, 2022.
- [DFS20] Ghada Dessouky, Tommaso Frassetto, and Ahmad-Reza Sadeghi. HybCache: Hybrid Side-Channel-Resilient Caches for Trusted Execution Environments. In Srdjan Capkun and Franziska Roesner, editors, *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, pages 451–468. USENIX Association, 2020.
- [DS16] François Durvaux and François-Xavier Standaert. From Improved Leakage Detection to the Detection of Points of Interests in Leakage Traces. In Marc Fischlin and Jean-Sébastien Coron, editors, *Advances in Cryptology - EUROCRYPT 2016 - 35th Annual International Conference on the Theory*

- and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part I*, volume 9665 of *Lecture Notes in Computer Science*, pages 240–262. Springer, 2016.
- [DXS19] Shuwen Deng, Wenjie Xiong, and Jakub Szefer. Secure TLBs. In Srilatha Bobbie Manne, Hillery C. Hunter, and Erik R. Altman, editors, *Proceedings of the 46th International Symposium on Computer Architecture, ISCA 2019, Phoenix, AZ, USA, June 22-26, 2019*, pages 346–359. ACM, 2019.
- [GRBG18] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks. In William Enck and Adrienne Porter Felt, editors, *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, pages 955–972. USENIX Association, 2018.
- [HL17] Zecheng He and Ruby B. Lee. How Secure is Your Cache Against Side-Channel Attacks? In Hillery C. Hunter, Jaime Moreno, Joel S. Emer, and Daniel Sánchez, editors, *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2017, Cambridge, MA, USA, October 14-18, 2017*, pages 341–353. ACM, 2017.
- [Int16a] Intel. *Intel 64 and IA-32 Architectures Optimization Reference Manual*. Intel, Santa Clara, California, United States, 248966-033 edition, June 2016.
- [Int16b] Intel. *Intel 64 and IA-32 Architectures Software Developer’s Manual Volume 2 (2A, 2B, 2C & 2D): Instruction Set Reference A-Z*. Intel, Santa Clara, California, United States, 325383-060us edition, September 2016.
- [Int18] Intel. *Intel Digital Random Number Generator (DRNG) Software Implementation Guide*. Intel, Santa Clara, California, United States, revision 2.1 edition, October 2018.
- [Kan10] David Kanter. Westmere Arrives. *Real World Technologies*, March 2010.
- [KHF⁺19] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*, pages 1–19. IEEE, 2019.
- [LL14] Fangfei Liu and Ruby B. Lee. Random Fill Cache Architecture. In *47th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2014, Cambridge, United Kingdom, December 13-17, 2014*, pages 203–215. IEEE Computer Society, 2014.
- [LPAA⁺20] Jason Lowe-Power, Abdul Mutaal Ahmad, Ayaz Akram, Mohammad Alian, Rico Amslinger, Matteo Andreozzi, Adrià Armejach, Nils Asmussen, Brad Beckmann, Srikant Bharadwaj, et al. The gem5 Simulator: Version 20.0+. *arXiv preprint arXiv:2007.03152*, 2020.
- [LSG⁺18] 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 William Enck and Adrienne Porter Felt, editors, *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, pages 973–990. USENIX Association, 2018.

- [NPGB21] Ajay Nayak, B. Pratheek, Vinod Ganapathy, and Arkaprava Basu. (Mis)managed: A Novel TLB-based Covert Channel on GPUs. In Jiannong Cao, Man Ho Au, Zhiqiang Lin, and Moti Yung, editors, *ASIA CCS '21: ACM Asia Conference on Computer and Communications Security, Virtual Event, Hong Kong, June 7-11, 2021*, pages 872–885. ACM, 2021.
- [OST06] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache Attacks and Countermeasures: the Case of AES. In David Pointcheval, editor, *Topics in Cryptology - CT-RSA 2006, The Cryptographers' Track at the RSA Conference 2006, San Jose, CA, USA, February 13-17, 2006, Proceedings*, volume 3860 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2006.
- [Pag05] Dan Page. Partitioned Cache Architecture as a Side-Channel Defence Mechanism. *IACR Cryptol. ePrint Arch.*, 2005:280, 2005.
- [PGGV21] Antoon Purnal, Lukas Giner, Daniel Gruss, and Ingrid Verbauwhede. Systematic Analysis of Randomization-based Protected Cache Architectures. In *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*, pages 987–1002. IEEE, 2021.
- [PGM⁺16] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks. In Thorsten Holz and Stefan Savage, editors, *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*, pages 565–581. USENIX Association, 2016.
- [PTSM15] Misel-Myrto Papadopoulou, Xin Tong, André Sez nec, and Andreas Moshovos. Prediction-Based Superpage-Friendly TLB Designs. In *21st IEEE International Symposium on High Performance Computer Architecture, HPCA 2015, Burlingame, CA, USA, February 7-11, 2015*, pages 210–222. IEEE Computer Society, 2015.
- [QP06] Moinuddin K. Qureshi and Yale N. Patt. Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches. In *39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-39 2006), 9-13 December 2006, Orlando, Florida, USA*, pages 423–432. IEEE Computer Society, 2006.
- [Qur18] Moinuddin K. Qureshi. CEASER: Mitigating Conflict-Based Cache Attacks via Encrypted-Address and Remapping. In *51st Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2018, Fukuoka, Japan, October 20-24, 2018*, pages 775–787. IEEE Computer Society, 2018.
- [Qur19] Moinuddin K. Qureshi. New Attacks and Defense for Encrypted-Address Cache. In Srilatha Bobbie Manne, Hillery C. Hunter, and Erik R. Altman, editors, *Proceedings of the 46th International Symposium on Computer Architecture, ISCA 2019, Phoenix, AZ, USA, June 22-26, 2019*, pages 360–371. ACM, 2019.
- [Sel04] Thorild Selen. *Reorganisation in the Skewed-Associative TLB*. Department of Information Technology, Uppsala University, 2004.
- [Sez04] André Sez nec. Concurrent Support of Multiple Page Sizes on a Skewed Associative TLB. *IEEE Trans. Computers*, 53(7):924–927, 2004.
- [SiF] SiFive. HiFive Unleashed. <https://www.sifive.com/boards/hifive-unleashed>. Last accessed: 28/03/2022.

- [Sil] Silvaco. 15nm Open-Cell Library and 45nm FreePDK. <https://si2.org/open-cell-library/>. Last accessed: 07/08/2022.
- [SM15] Tobias Schneider and Amir Moradi. Leakage Assessment Methodology - A Clear Roadmap for Side-Channel Evaluations. In Tim Güneysu and Helena Handschuh, editors, *Cryptographic Hardware and Embedded Systems - CHES 2015 - 17th International Workshop, Saint-Malo, France, September 13-16, 2015, Proceedings*, volume 9293 of *Lecture Notes in Computer Science*, pages 495–513. Springer, 2015.
- [SQ21] Gururaj Saileshwar and Moinuddin K. Qureshi. MIRAGE: Mitigating Conflict-Based Cache Attacks with a Practical Fully-Associative Design. In Michael Bailey and Rachel Greenstadt, editors, *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, pages 1379–1396. USENIX Association, 2021.
- [TAL⁺22] Daniel Townley, Kerem Arıkan, Yu David Liu, Dmitry Ponomarev, and Oğuz Ergin. Composable Cachelets: Protecting Enclaves from Cache Side-Channel Attacks. In *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*, pages 2839–2856, Boston, MA, August 2022. USENIX Association, 2022.
- [Tan09] Andrew S. Tanenbaum. *Modern Operating Systems, 3rd Edition*. Pearson Prentice-Hall, 2009.
- [TNF⁺23] Jan Philipp Thoma, Christian Niesler, Dominic A. Funke, Gregor Leander, Pierre Mayr, Nils Pohl, Lucas Davi, and Tim Güneysu. ClepsydraCache - Preventing Cache Attacks with Time-Based Evictions. In *32nd USENIX Security Symposium (USENIX Security 23)*, Anaheim, CA, August 2023. USENIX Association.
- [TOS10] Eran Tromer, Dag Arne Osvik, and Adi Shamir. Efficient Cache Attacks on AES, and Countermeasures. *Journal of Cryptology*, 23(1):37–71, 2010.
- [TTGB22] Andrei Tatar, Daniël Trujillo, Cristiano Giuffrida, and Herbert Bos. TLB;DR: Enhancing TLB-based Attacks with TLB Desynchronized Reverse Engineering. In *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*, Boston, MA, August 2022. USENIX Association, 2022.
- [TZBR21] Qinhan Tan, Zhihua Zeng, Kai Bu, and Kui Ren. PhantomCache: Obfuscating Cache Conflicts with Localized Randomization. In *28th Annual Network and Distributed System Security Symposium, NDSS 2021, virtually, February 21-25, 2021*. The Internet Society, 2021.
- [VSIH22] Markus Velten, Robert Schöne, Thomas Ilsche, and Daniel Hackenberg. Memory Performance of AMD EPYC Rome and Intel Cascade Lake SP Server Processors. *CoRR*, abs/2204.03290, 2022.
- [WFZ⁺16] Yao Wang, Andrew Ferraiuolo, Danfeng Zhang, Andrew C. Myers, and G. Edward Suh. SecDCP: Secure Dynamic Cache Partitioning for Efficient Timing Channel Protection. In *Proceedings of the 53rd Annual Design Automation Conference, DAC 2016, Austin, TX, USA, June 5-9, 2016*, pages 74:1–74:6. ACM, 2016.
- [WL07] Zhenghong Wang and Ruby B. Lee. New Cache Designs for Thwarting Software Cache-Based Side Channel Attacks. In Dean M. Tullsen and Brad Calder,

- editors, *34th International Symposium on Computer Architecture (ISCA 2007)*, June 9-13, 2007, San Diego, California, USA, pages 494–505. ACM, 2007.
- [WL08] Zhenghong Wang and Ruby B. Lee. A Novel Cache Architecture with Enhanced Performance and Security. In *41st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-41 2008)*, November 8-12, 2008, Lake Como, Italy, pages 83–93. IEEE Computer Society, 2008.
- [WLA⁺16] Andrew Waterman, Yunsup Lee, Rimas Avizienis, David A. Patterson, and Krste Asanović. The RISC-V Instruction Set Manual Volume II: Privileged Architecture Version 1.9.1. Technical Report UCB/EECS-2016-161, Electrical Engineering and Computer Sciences, University of California at Berkeley, November 2016.
- [WUG⁺19] Mario Werner, Thomas Unterluggauer, Lukas Giner, Michael Schwarz, Daniel Gruss, and Stefan Mangard. ScatterCache: Thwarting Cache Attacks via Cache Set Randomization. In Nadia Heninger and Patrick Traynor, editors, *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, pages 675–692. USENIX Association, 2019.
- [YF14] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In Kevin Fu and Jaeyeon Jung, editors, *23th USENIX Security Symposium, USENIX Security 14, San Diego, CA, USA, August 20-22, 2014*, pages 719–732. USENIX Association, 2014.
- [Yue] Peter Yuen. Gem5 RISC-V PARSEC. <https://github.com/ppeetteers/gem5-RISC-V-PARSEC>. Last accessed: 28/03/2022.

A Influence of Associativity

In this section, we investigate the influence of the associativity on the re-randomization threshold of TLBCOAT. To extend the range of parameter sets covered in this paper we use a TLB with 128 entries for this purpose.

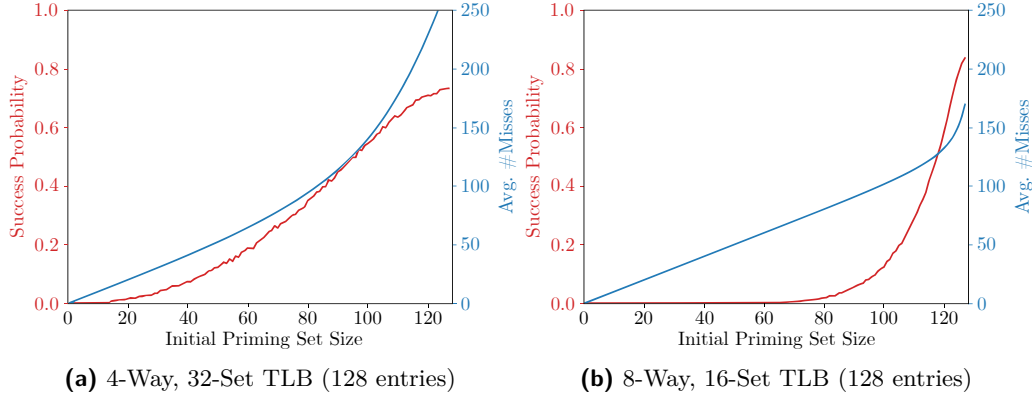


Figure 9: Catching probability and average number of misses for the prime and prune steps as a function of the initial probing set size for finding one translation conflict using the PRIME+PRUNE+PROBE attack. The modeled TLB uses the RPLRU replacement policy. It implements index-randomization but does not re-randomize the mapping during the attack.

Figure 9 shows the catching probability and the average number of misses for profiling sets containing between 0 and 128 entries. The results were obtained using our functional simulation tool. On the left, the TLB with 128 entries is divided in 4 ways, while the right figure shows the results on a TLB with 128 entries and 8 ways. The figure clearly shows that the success probability for the 8-way TLB is much lower for small priming sets compared to the 4-way TLB. For large priming sets, the success probability on the 8-way TLB outreaches the one from the 4-way TLB. The reason for this is that it is more likely that a set of random addresses fills the four required entries in the 4-way TLB compared to filling 8 ways in the 8-way TLB. At the same time, the 4-way TLB is more likely to cause conflicts for the profiling set since fewer addresses are required to fill any given randomized set. Therefore, the average number of misses is higher and conflicts start to appear earlier in the 4-way TLB.

Now, the question arises how the different characteristics of TLBs with different associativity affects the overall security against PRIME+PRUNE+PROBE attacks. Therefore, we repeat the experiment from Section 6.3 for the TLBs with 128 entries. The results are shown in Figure 10. The figures clearly show that the profiling produces fewer TLB misses on the 4-way TLB. The most successful attempts for the 4-way TLB have been made with a profiling set-size between 60 and 70 addresses. If we compare the results to the experiment shown in Figure 9, it shows that the catching probability for the 8-way TLB is very low in this range. Therefore, the profiling attempt depicted in Figure 10b yields the best results for priming set sizes between 80 and 100 addresses. However, since in this region the average number of TLB misses for observing a single conflict is much higher than at 70 addresses in the 4-way TLB, the overall number of misses for the profiling in the 8-way TLB is higher than in the 4-way TLB.

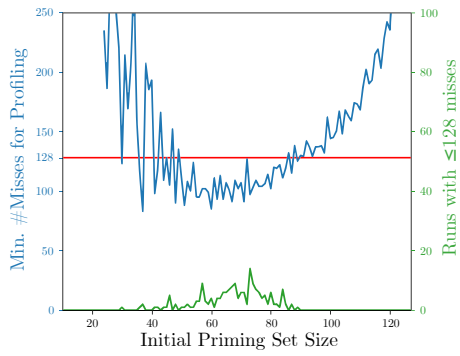
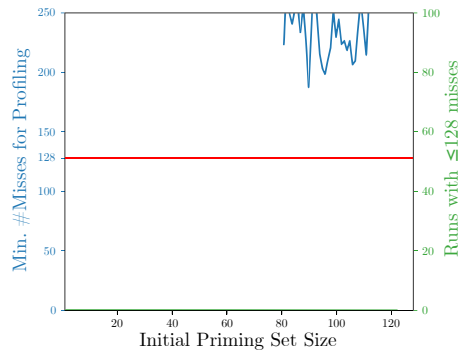
**(a)** 4-Way, 32-Set TLB (128 entries)**(b)** 8-Way, 16-Set TLB (128 entries)

Figure 10: The blue line depicts the minimum experienced misses to obtain a functional probabilistic eviction set during profiling of PRIME+PRUNE+PROBE over 100,000 trials for each initial priming set size. Assuming a re-randomization threshold of N (the number of entries available), the green line depicts the total number of successful runs out of the 100,000 iterations.

B Miss-Optimized Prime+Prune+Probe

Algorithm 1 PRIME+PRUNE+PROBE attack on the TLB optimized for low number of TLB misses. Blue functions are not normally available to an attacker.

```

Input: target, memory, prime_size
e ← {}
prime_adrs ← {}
for  $i \in \{1, \text{prime\_size}\}$  do
  prime_adrs ← prime_adrs  $\cup$  memory +  $i * 4096$ 
end for
while not is_functional(target, e) do
  for a in prime_adrs do ▷ Prime
    access(a)
  end for
  while true do ▷ Prune
    conflicts ← 0
    for a in prime_adrs do
      if probe(a) = miss then
        conflicts ← conflicts + 1
      end if
    end for
    if conflicts = 0 then
      break
    end if
  end while
  access(target) ▷ Access
  for a in prime_adrs do ▷ Probe
    if probe(a) = miss then
      e ← e  $\cup$  a
    end if
  end for
  for a in e do ▷ Remove target TLB entry
    access(a)
  end for
  if not is_victim_removed(target) then
    for a in prime_adrs do
      access(a)
    end for
  end if
  while not is_victim_removed(target) do
    a ← get_new_address()
    access(a)
    prime_adrs ← prime_adrs  $\setminus$  prime_adrs[0]
    prime_adrs ← prime_adrs  $\cup$  a
  end while
end while
return e

```

Algorithm 1 shows the modified PRIME+PRUNE+PROBE [PGGV21] algorithm used to construct eviction sets with reduced number of conflicts. Therefore, the attacker is given additional information that would not be available to them in a real-world attack. This information is highlighted in blue in the algorithm.

In the first step, the attacker needs to prime the TLB using the addresses in the initial priming set. Then, the addresses need to be pruned, i.e., the attacker must make sure that there are no self-caused evictions by accessing addresses of the priming set. After this, the TLB is in a prepared state for the attack and the access to the victim page is triggered. Next, the attacker probes the initial priming set for a TLB miss. If such a miss occurs, the address is added to the eviction set, since the access to the victim page has evicted this entry. Finally, the attacker needs to remove the victim page translation from the TLB to start the next profiling round. To reduce the conflicts during this process, first the addresses that are already known to collide with the target are accessed. This increases the chances of evicting the target translation quickly. As long as the victim translation is not removed, the attacker needs to access further addresses. By re-accessing the priming set, the attacker generates additional chances to evict the victim without causing many new TLB misses - that is, since most of the priming address translations are still stored in the TLB from the prime and prune step. If this also does not remove the victim translation, the attacker needs to access new addresses which cause at least one TLB miss each until the victim translation is evicted.