ClepsydraCache – Preventing Cache Attacks with Time-Based Evictions

Jan Philipp Thoma, *Ruhr University Bochum;* Christian Niesler, *University of Duisburg-Essen;* Dominic Funke, Gregor Leander, Pierre Mayr, and Nils Pohl, *Ruhr University Bochum;* Lucas Davi, *University of Duisburg-Essen;* Tim Güneysu, *Ruhr University Bochum & DFKI*

https://www.usenix.org/conference/usenixsecurity23/presentation/thoma

This paper is included in the Proceedings of the 32nd USENIX Security Symposium.

August 9–11, 2023 • Anaheim, CA, USA

978-1-939133-37-3

ClepsydraCache – Preventing Cache Attacks with Time-Based Evictions

Jan Philipp Thoma
Ruhr-University Bochum

Christian Niesler
University of Duisburg-Essen

Dominic Funke
Ruhr-University Bochum

Gregor Leander
Ruhr-University Bochum

Pierre Mayr
Ruhr-University Bochum

Nils Pohl
Ruhr-University Bochum

Lucas Davi
University of Duisburg-Essen

Tim Güneysu
Ruhr-University Bochum & DFKI

Abstract

In the recent past, we have witnessed the shift towards attacks on the microarchitectural CPU level. In particular, cache side-channels play a predominant role as they allow an attacker to exfiltrate secret information by exploiting the CPU microarchitecture. These subtle attacks exploit the architectural visibility of conflicting cache addresses. In this paper, we present ClepsydraCache, which mitigates state-of-the-art cache attacks using a novel combination of cache decay and index randomization. Each cache entry is linked with a Time-To-Live (TTL) value. We propose a new dynamic scheduling mechanism of the TTL which plays a fundamental role in preventing those attacks while maintaining performance. ClepsydraCache efficiently protects against the latest cache attacks such as Prime+(Prune+)Probe. We present a full prototype in gem5 and lay out a proof-of-concept hardware design of the TTL mechanism, which demonstrates the feasibility of deploying ClepsydraCache in real-world systems.

A Clepsydra is an ancient time-measuring device worked by a flow of water.

1 Introduction

The multi-layer cache hierarchy is a fundamental design element in modern microprocessors that bridges the performance gap between the main memory and the CPU. By keeping frequently accessed data in close proximity to the CPU, lengthy pipeline stalls due to high memory latency can be avoided. Typically, modern desktop-grade CPUs implement three levels of cache of which the L1 and L2 cache are unique to each CPU core, while the last level cache (LLC) is shared among all cores. The size of cache memory is usually in the range of a few hundred kilobytes for L1 caches and multiple megabytes for last level caches. Since this is not nearly enough memory to store all relevant data, caches inevitably need to evict less frequently used entries to make space for new data.

From an architectural point of view, caches are transparent to the software, i.e., a process does neither need to manage data stored within the cache nor does it know what data is currently cached. However, caches store data based on temporal locality. That is, data that was accessed recently is cached since the CPU expects it to be accessed again soon. Unfortunately, the story of software-transparent caches which hide the latency of the main memory was disrupted by the introduction of cache timing attacks [5,35]. Attackers can measure the execution time of a victim program or even a single memory access and therefore determine whether data was cached. In [5], the cache side-channel is used to leak an AES encryption key in OpenSSL, while [56] recovers keys from a victim program running GnuPG. Cache side-channels are further used in the context of transient attacks like Spectre [25] and Meltdown [29]. Even software running in a trusted execution environment like Intel SGX can be attacked using cache side-channels as demonstrated in [9, 18]. The shared nature of the LLC makes it a particularly worthwhile target for attackers since the timing side-channel can be exploited beyond process boundaries or even on co-located virtual machines. While attacks like Flush+Reload [56] and Flush+Flush [15] exploit a special instruction that allows attackers to flush specific cache lines, the Prime+Probe attack does not require such an instruction and is universally applicable across ISAs [35, 47]. Moreover, in contrast to flush-based attacks, the latter does not require shared memory between the victim and the attacker.

Since the very design goal of caches is to accelerate slow accesses to the main memory, the timing side-channel cannot easily be mitigated without losing the crucial performance benefit that caches provide. This is also why cache side-channels are spread over a huge variety of CPUs including Intel, AMD, ARM and RISC-V processors [14, 21]. Since the vulnerability is rooted deeply in the hardware, it is especially difficult to mitigate it on the software level [12, 48]. Detection-based mechanisms [10, 11, 13, 16, 55] suffer from false positives since the cache access patterns vary drastically between software. Hence, most recent proposals feature hardware modifications that change the way new entries are cached. The architectural solutions can be divided into two classes: cache partitioning [30, 36, 39, 40, 42, 50, 54] splits the cache memory into disjunctive security domains, thereby preventing information leaks beyond security domain boundaries. The goal of partitioning is to prevent attackers from observing evictions originated from other security domains than their own. On the other hand, index randomization designs [31, 38, 46, 51–53] do not reduce the amount of cache conflicts but instead randomize the mapping of addresses to cache entries. This way, the cost of finding eviction sets - that is, a set of addresses that map to the same cache entries as the victim address - is drastically increased, preventing efficient PRIME+PROBE attacks. However, recent work [37] demonstrates that index randomization alone is insufficient for security and requires frequent rekeying. Attackers can still construct generalized eviction sets that have a high probability of evicting the target address. Unfortunately, frequent rekeying is not practical, as it induces high performance overhead and effectively causes a complete cache flush.

Contributions. With CLEPSYDRACACHE, we introduce a novel and secure index-based randomization scheme which effectively protects against state-of-the-art cache attacks while at the same time preserving the performance of traditional caches. This includes PRIME+PRUNE+PROBE [37], which is one of the most recent and sophisticated attack techniques. We show that the combination of cache decay and index randomization effectively protects against modern cache attacks. Our first line of defense is a randomized address to cache entry mapping (index randomization). Moreover, we introduce a hardware-integrated time-to-live-based eviction strategy (cache decay) that drastically reduces the amount of cache conflicts and renders reliable observation of such conflicts infeasible. This prevents the attacker from learning useful information when observing cache misses and hence, building eviction sets. Our security analysis shows that CLEPSYDRACACHE provides strong security properties against state-of-the-art attacks including the recent PRIME+PRUNE+PROBE [37] which bypasses pure index-randomization schemes. Meanwhile, our design maintains the flexibility and scalability of traditional caches. CLEPSYDRACACHE thereby avoids additional complexity on the critical path of cache accesses, such as the indirections used in [41]. In contrast to other works [31, 38, 46, 51–53], in Section 7.2 we provide a proof-of-concept CMOS design for the key time-to-live feature that CPU developers can build on for minimal overhead implementation of CLEPSYDRACACHE. In Section 8, we evaluate CLEPSYDRACACHE using gem5 [33] and provide a detailed performance analysis using state-of-the-art benchmarks including Parsec [6] and SPEC CPU 2017 [20]. Our results indicate a small performance overhead of 1.38%.

2 Background

In this section we provide background information on cache architectures and cache side-channel attacks.

2.1 Caches

Caches are small but highly efficient temporary storage components that are located in close proximity to the CPU. They bridge the gap between the slow main memory and the CPU. Typically, a hierarchical approach is taken for cache memory, i.e., each CPU core is equipped with a small L1 and a slightly larger L2 cache. On multicore systems, all cores typically share the LLC.

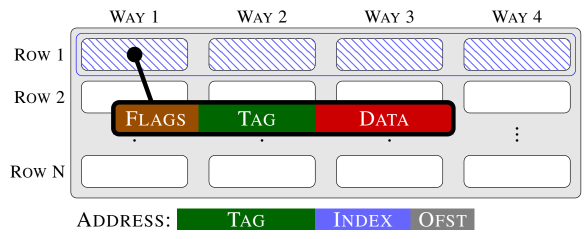

Figure 1: 4-Way set associative cache with N sets (the first set is highlighted in blue). The flags section of an entry typically includes a valid and a dirty bit.

Since caches by design need to be extremely fast, an addressing function is required that can efficiently determine whether the requested data is currently cached, where it is located, and - in case of a cache miss - quickly determine which entry is used for new data. Therefore, most architectures use set-associative caches which feature a table-like structure. Set-associative caches are divided into *ways* and *sets* which can be imagined as table columns and rows, respectively. An illustrative layout of a set associative cache is shown in Fig. 1. A cache *entry* is a cell in the table and uniquely addressed by the *set* (in the non-randomized setting equivalent to the row) and the *way*. Ways are typically managed in parallel hardware memory blocks (also called *banks*) to allow concurrent access to each way.

When a program performs a memory operation, the virtual address is first translated to a physical address by the memory management unit (MMU). The lower bits of the virtual address indicate the page offset and hence, are identical to the physical address. The upper bits of the virtual address hold the virtual page number which is translated to a physical frame number in the physical address. Hence, the upper bits are not directly controlled by user-level software running on the processor. For cache addressing, the physical address of the data is divided into a *tag*, an *index*, and an *offset*. If an entry holds m bytes, the offset part of the address is $log_2(m)$ bits. The index part of the address is $log_2(N)$ bit, where N is the amount of available cache sets. The remainder of the address is used as a tag which is always stored alongside the data in cache memory. The index determines the set (i.e., the row) in which the data is stored. Then, all entries located in the set are searched for the tag bits of the incoming address. When the tag matches in one of the ways, a cache hit occurs, and the memory operation is performed directly in the cache.

If the tag of the incoming address does not match in one of the ways, the data is requested from another memory module which is more distant to the CPU, e.g., another cache or the main memory. When the request is served, the replacement policy selects an entry within the set determined by the index bits and stores the data in that entry. The tag of the address is again stored alongside the data. If the entry was valid and a write was performed on this entry prior to the replacement, the data needs to be written back during the replacement phase since the modification is not yet committed to other memory modules (writeback dirty). Each entry has flags indicating whether the entry is modified (*dirty*) and *valid*. Since there exist diverse variations of cache memory, including different store and writeback policies, we refer the reader to [19] for a detailed description.

2.2 Cache Side-Channels

The ability to distinguish cache hits and misses based on the latency of a memory instruction can be leveraged for side-channel attacks, where an attacker extracts secret information from other processes by observing the cache timing behavior and therefore can draw conclusions about accessed data.

In the PRIME+PROBE attack [35,47], an attacker fills a section of interest in the cache with their own data (*prime*). Next, the victim process is executed which may evict some of the attacker's cache entries upon memory access. In the final step (*probe*), the attacker measures the access timing for re-loading the data placed during the prime phase to recover which entries were evicted by the victim. Initially, PRIME+PROBE was mostly used for caches closer to the CPU, since the cost of evicting large portions of the LLC is typically very high. However, in [32] the feasibility of PRIME+PROBE attacks on LLCs was demonstrated using minimal eviction sets. An eviction set is a set of addresses that map to the same cache set, i.e., the index bits of the address are equal. An eviction set is minimal if the amount of addresses within the set equals the number of cache ways. If an attacker manages to create such an eviction set, it is possible to efficiently clear selected victim data from all cache levels including the LLC. Due to the translation of virtual to physical addresses, it is not always trivial to create such an eviction set since the upper bits of the physical address depend on the virtual page number on which the attacker has only limited influence. To overcome this, efficient algorithms for obtaining minimal eviction sets have recently been developed [45, 49]. A variant of PRIME+PROBE is FLUSH+RELOAD [56]. While this attack does not require eviction sets, it does require shared memory between the attacker and the victim as well as a `clflush` instruction.

The recent PRIME+PRUNE+PROBE [37] targets cache architectures that use index-randomization to prevent the construction of minimal eviction sets. By using generalized eviction sets, the attack can reliably evict entries from the cache in reasonable time. As shown later in this paper, the attack is not applicable to CLEPSYDRACACHE.

3 Problem Description and Related Work

The fundamental intrinsic allowing cache attacks, i.e., the timing difference between cache hits and misses, is well intended and in fact, the very reason why caches exist. Hence, a successful mitigation technique must hide as much of the cache internals from a potential attacker without forfeiting the performance advantages. Software mitigations for cache attacks in general have proven to be costly in terms of computational overhead [12,48]. Hence, more practical approaches usually involve hardware changes, sometimes mediated by the software. Most recent mitigation techniques either rely on cache partitioning or index randomization.

Cache partitioning [36] splits a cache into multiple partitions, which are assigned to different security domains. Thus, the information flow between different partitions is constrained, preventing leakage between security domains. A common approach is to divide the cache among its cache sets. However, this approach often offers only one-way protection [50] meaning that information flow from a confidential to a public partition is prohibited, but not the other way around (public to confidential). Generally, a distinction is made between static and dynamic partitioning schemes. For dynamically partitioned caches like [39,42,54], timing side-channel attacks are possible because after resizing partitions, entries outside of the shifted boundaries remain valid. Therefore, the cache must be either partitioned statically or the run-time cache partitioning needs to be designed not to move existing entries which is a complicated process. The disadvantage of static partitioning is the high performance overhead resulting from the limited flexibility. Since cache demands vary during application runtime, the allocated partition is either too small or too large to achieve optimal performance. Furthermore, the amount of required partitions may vary within short time periods depending on the machine's workload.

A different approach to side-channel secure caches is based on randomizing the way a memory address is mapped to a particular cache set and line (index randomization). For instance, [31] proposes a logical direct-mapped cache with extra bits for the indexing. Those extra index bits significantly increase the search space for finding cache conflicts. Recent works combine the randomization approach with further techniques to improve the security against ever-evolving attacks.

SCATTERCACHE [53] uses a randomized cache mapping and adds software-assisted domain separation. This allows duplicating shared addresses in the cache for each process which prevents flush-based attacks like FLUSH+RELOAD. However, as shown in [37], SCATTERCACHE requires frequent re-randomization to avoid construction of probabilistic eviction sets to protect against the presented PRIME+PRUNE+PROBE attack. All designs that require software involvement, including partitioning or domain separation for the randomization function face the challenge of backwards compatibility. This particularly involves two features: (1) operating system adjustments to support security domains and (2) support for legacy software. Therefore by default, all applications belong to the same security domain if the software is not aware of the partitioning.

Another approach, dubbed PhantomCache [46], is a pure architectural solution and uses a localized randomization technique to bind the randomized mapping to a limited number of cache sets. The randomization technique used in PhantomCache allows an address to be mapped to multiple locations within a single cache bank. Since in the worst case, all possible entries for a given address map to the same cache bank, parallel lookup is not possible. Therefore, the lookup latency is increased over traditional randomization schemes.

Recent work [37] analyzed the security properties of randomized cache designs like SCATTERCACHE and PhantomCache and proposed a generic attack on randomized caches dubbed PRIME+PRUNE+PROBE. The attack challenges the assumption that index-randomization suffices to prevent PRIME+PROBE attacks by constructing probabilistic eviction sets with relatively small sizes. A probabilistic eviction set contains addresses that collide with the target in at least one cache way. By combining multiple such addresses, the target can be efficiently evicted. The attack is split into a profiling and an attack phase. During the profiling phase, the attacker selects a set of addresses k and accesses them repeatedly. Due to the randomization, each address can be stored at an independent index for each way. This leads to a large variety of combinations where the $|k|$ addresses can be stored in the cache. Eventually, the attacker obtains a set $k' \subseteq k$ of addresses that are co-located in the cache without evicting each other. Then, the attacker triggers the access on a target address x which evicts an address of k' with *catching probability* p_c. If such an eviction is observed, the evicted address from k' is known to collide with x in at least one cache way and is therefore added to the eviction set G. This phase is repeated until G contains a sufficient number of addresses. For an attack with 90% success rate, G needs to have between 36 (4-way cache) and 576 (16-way cache) addresses for schemes where each way has an independent addressing function, see [37, Tab. II]. Once G is established, the attacker can use G to evict x similar to the PRIME+PROBE attack.

A different approach for side-channel secure caches is to replicate the behavior of fully associative caches. Since those designs suffer from a high lookup latency for large caches, they are usually not suited for large LLCs. The recently proposed Mirage [41] mimics a fully associative cache by separating the tag store from the data store. By over-provisioning the size of the set-associative tag-store, conflicts are seldom. Mirage uses a bidirectional pointer mechanism to refer from the tag store to the data store and vice versa. However, the pointer indirection occurs directly on the critical path of the access. Hence, the access latency of Mirage is composed of $t_{rand} + t_{accessTag} + t_{accessData}$. The influence of t_{rand} can be minimized by choosing a low-latency randomization function. Since the tag-access only yields the location for the data-access, the accesses cannot be parallelized. The tag-store is managed in a set-associative structure and thus, $t_{accessTag}$ is similar to the data access latency in a traditional cache (t_{access}). If we assume that the tag- and data-access have similar latency, the overall access latency of Mirage is approximately double compared to traditional caches: $t_{rand} + t_{accessTag} + t_{accessData} \approx 2 \cdot t_{access}$. Besides the randomization function, CLEPSYDRACACHE does not add complexity on the critical access-path, and hence, the access latency in hardware is closer to traditional caches. In Section 7.2 we propose a hardware design for the TTL mechanism of CLEPSYDRACACHE that demonstrates the feasibility of efficient implementations of such designs both in respect to performance and area.

Our Goal. To summarize, the main limitations of all existing proposals are that they either do not prevent all attack strategies or induce high performance penalties. To prevent cache attacks, the goal is to ensure that an attacker gains no exploitable information about the cache content while preserving the efficiency of caches. To do so, we develop an original randomization mechanism, called CLEPSYDRACACHE that reduces information leakage far beyond currently proposed designs. The design employs a low-latency index-randomization scheme that operates on the entire address except the offset bits. The cornerstone of CLEPSYDRACACHE is an eviction strategy based on timing rather than contention. This strategy is also known as *cache decay* and has been considered for the purpose of reducing power leakage in [22] and for access-trace-based side-channels on AES in [23]. The combination of randomization and cache decay is unique for CLEPSYDRACACHE and is further improved by a dynamic TTL scaling mechanism. These features are indispensable for the security against state-of-the-art attacks including PRIME+PRUNE+PROBE. We propose the first hardware design of the cache decay feature which demonstrates the potential of low-overhead implementations of CLEPSYDRACACHE by CPU developers.

4 Threat Model

Our threat model follows previous work in this field [37, 46, 53]. We consider a black-box attacker in an ideal, noise-free scenario. Hence, the attacker is able to perfectly distinguish between a cache hit and a cache miss. As in real-world attacks, the attacker has no insights on the internal state of the cache except those leaked by the timing of memory accesses. Further, the attacker is able to access an arbitrary number of addresses and measure the execution time of a victim program, potentially revealing the cache hit and miss behavior of the victim. We assume the index randomization function to be pseudorandom, i.e., the attacker cannot predict or guess the cache entry to which an address is mapped. Attacks on the hardware level are out of scope for this paper, i.e., the attacker cannot snoop or manipulate the memory, memory buses, or random numbers. While we consider the influence of the chip temperature for our hardware design, attacks tampering the physical circuity, or the environment are also beyond the scope of this paper. Our design specifically aims to prevent conflict-based attacks like PRIME+PROBE and deviates. To protect against flush-based attacks like FLUSH+RELOAD, one can either restrict access to the flushing instruction (`clflush`), or use the memory duplication method presented in [53]. We investigate in detail the security of our approach in Section 6.

5 Concept

In this section, we provide a detailed description of the CLEPSYDRACACHE concept. The design is suited for all levels of caches, especially including shared last-level caches.

5.1 CLEPSYDRACACHE in a Nutshell

In traditional caches, entries can either be evicted using a special instruction defined by the ISA or as a result of conflicting addresses that are mapped to the same entry. All current cache attacks exploit either one of these two properties. While the former could easily be addressed by not offering such an instruction - in fact, many ISAs do not feature a cache line flush instruction - the latter is more of a fundamental problem in current cache designs and cannot easily be mitigated.

The main design goal of CLEPSYDRACACHE is to drastically reduce the overall amount of cache conflicts and to remove the direct linkage between cache accesses and cache evictions. With our design, we achieve this goal and therefore prevent conflict-based cache attacks including PRIME+PROBE and PRIME+PRUNE+PROBE. CLEPSYDRACACHE is compatible with cache duplication of shared memory as proposed in [53] to defend against flush-based attacks like FLUSH+RELOAD.

Instead of evicting entries when a conflicting address is accessed, in CLEPSYDRACACHE each entry is assigned with a time-to-live (TTL) value that is randomly initialized. The TTL is steadily reduced and when expired, the entry is evicted from the cache. By dynamically adapting the global rate with which the TTL of each entry decreases, we achieve a high cache utilization with minimal conflicts. Secondly, CLEPSYDRACACHE uses a highly randomized address to cache mapping, and in doing so, efficiently prevents constructing minimal eviction sets without observing conflicts.

The CLEPSYDRACACHE design is developed with large caches (i.e., typical LLC caches) in mind and therefore must respect the high performance requirement. We hold on to the traditional set-associative design and implement a variant of the randomized address-to-cache-mapping. The access-path of CLEPSYDRACACHE is therefore equal to traditional caches with the addition of the randomization function. Instead of checking the valid bit in traditional caches, CLEPSYDRACACHE needs to check if the TTL is not zero. Similar to all randomized cache architectures, a low-latency randomization function is key to a low access time of the cache. The TTL mechanism does not affect the critical path. As studied in recent years, index-randomization makes finding conflicting addresses much more challenging for an attacker, see [46, 53]. In combination with the time-based eviction, finding eviction sets becomes infeasible as we discuss in detail in Section 6.

5.2 Per-Entry Time-To-Live (TTL)

Each entry in CLEPSYDRACACHE is assigned with a TTL value that indicates for how long the entry remains in the cache. On a cache miss, the accessed data is loaded from the memory and the addressing function determines the target entry in the cache. Simultaneously, a uniformly random TTL in between a lower and an upper bound is chosen and assigned to the entry. From there on, the TTL is steadily reduced with a reduction rate R_{TTL}. When the TTL for an entry expires, the data is immediately invalidated and - if required - written back to the main memory. If a cache hit occurs, i.e., an entry with $TTL > 0$ is accessed, the data is served from the cache and the TTL is reset to a new uniformly random value between the lower and upper bound. This is to make sure that frequently accessed entries stay cached. The reduction rate R_{TTL} globally applies to all entries in the cache and is scaled dynamically as described later in this paper.

5.3 Addressing and Replacement

CLEPSYDRACACHE implements a randomized address-to-cache-mapping. In traditional caches, the index bits of the address determine the line in which the data is placed, such that the entries in one line of the cache form a set (c.f. Fig. 1). Therefore, building a minimal eviction set is trivial once the addressing scheme is known to the attacker. In CLEPSYDRACACHE, we use a low-latency randomization function that assigns pseudorandom cache lines in each way to a given address as shown in Fig. 2. A single address deterministically maps to the same entry in each respective way at every access. For an address A, we call such a collection of indices $A \rightarrow \{i_1, ..., i_w\}$ a *dynamic set* that maps the address to an index in each cache way. For now, we assume that upon access each dynamic set contains at least one invalid entry due to the time-based evictions. One of the invalid entries is randomly selected to store the data without causing a conflict. We consider the case where all entries in a dynamic set are filled (i.e., the TTL of each of these entries is greater than zero) in the following section.

Since the amount of cache entries is limited and much smaller than the address space, addresses with conflicting

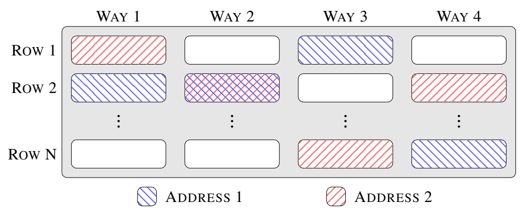

Figure 2: Address mapping of in CLEPSYDRACACHE design. Each address maps to a pseudorandom cache line in each way.

mappings are inevitable. In the following we state the properties of an ideal-keyed addressing function that is indistinguishable from a true random mapping of addresses to a dynamic set. Such a function $f_w : (\mathbb{F}_2^t, \mathbb{F}_2^i) \rightarrow (\mathbb{F}_2^{t'}, \mathbb{F}_2^i)$ is instantiated in way w and maps a t-bit input-tag and i-bit address-index to a t'-bit output-tag and an i-bit cache-index. We do allow $t \neq t'$ for practicality though an ideal instantiation has $t = t'$. For an address $\mathrm{A} = (\mathrm{tag}||\mathrm{index}||\mathrm{offset})$ in a W-way cache with N cache lines per way, f should yield the following properties:

- **Invertibility:** Given a set (tag, idx) in way w, it must be easy to compute $f_w^{-1}(tag, idx)$. This is required for write-backs of cache entries with expired TTL. Invertibility implies injectivity, i.e., there must not be a set of tuples $a = (tag, idx)$ and $b = (tag', idx')$ with $a \neq b$ such that $f_w(a) = f_w(b)$.
- **Index-Pseudorandomness:** The probability that a set of tuples $a = (tag, idx)$ and $b = (tag', idx')$ with $a \neq b$ map to the same index in way w must be close to $1/N$ for all $w \in W$. It must be difficult to construct such pairs. Note that N is typically small. Hence, this is not a genuine hash function.
- **Independence:** For $w, w' \in W$ and $w \neq w'$, f_w and $f_{w'}$ behave independently.

Note, that these properties do not require a collision-resistant hash function or a cryptographically secure block cipher. The reason for that is that the attacker never observes the actual index to which an address maps. Instead, the only observable behavior is when one address causes an eviction of another address, thus indicating that both addresses collide in at least one cache way.

Hence, we leverage a round-reduced block cipher with strong diffusion characteristics. The addressing function in CLEPSYDRACACHE is interchangeable and related work proposes several different techniques [31, 46, 53]. For our proof-of-concept design, we choose a round-reduced version of the lightweight block cipher PRINCE [7] which is designed to provide full diffusion - that is, every output bit depends on every input bit - after two rounds. In order to make hardware reverse engineering attacks on the key less attractive, the secret key should not be hardwired for an implementation but instead chosen on system startup.

Using a block cipher for the address randomization comes with the advantage of having encrypted tags in the cache and hence preventing attacks that tamper the address within the cache. Traditional block ciphers have a fixed block length which may result in a small storage overhead for the tag bits. By using a 64 bit block cipher like PRINCE the ciphertext has a length of 64 bit and all bits of the ciphertext are required for the decryption. The tag bits of the address are stored just like in traditional caches and the index bits of the encrypted address are stored implicitly by the location within the cache. However, in traditional caches the 6 offset bits can be discarded. When a 64 bit block cipher is used, the offset bits can be zeroed but the ciphertext includes 6 bits that are not zero and need to be stored as part of the tag. To get around this storage overhead, one could choose a format-preserving encryption (FPE) scheme [3, 4, 43]. These schemes encrypt an n-bit input to an n-bit output and are hence ideal for the task. Another, likely more efficient solution, is the design of a tailored mapping function that matches the security requirements and the ideal block size exactly.

5.4 Conflict Resolution

CLEPSYDRACACHE dynamically controls the global speed with which the TTL of the entries is reduced using the reduction rate R_{TTL}. In general, a high R_{TTL} means that entries are evicted faster while a low R_{TTL} yields longer lifetimes. As long as no cache conflicts occur, i.e. a dynamic set is accessed which does contain empty entries *or* the accessed entry is already cached in the dynamic set, R_{TTL} is slowly decreased towards the minimum value, e.g. $R'_{TTL} = R_{TTL} - 1$. When a conflict occurs, R_{TTL} is reset to a higher value, e.g. $R'_{TTL} = R_{TTL} \cdot 2$. This way many consecutive conflicts will lead to fast eviction of entries (and hence, less conflicts).

In an idealized model, one would adapt R_{TTL} such that every dynamic set has at least one free entry at all times. However, this would require expensive monitoring of the cache utilization in each dynamic set. Hence, with CLEPSYDRACACHE we approximate the desired behavior by dynamically adapting $\mathrm{R}_{\mathrm{TTL}}$ based on experienced conflicts. The approach is comparable to TCP congestion control [2]. Fig. 3 illustrates the conceptual evolution of the TTL reduction rate over time. We start by setting R_{TTL} to an initial value and over time decrease it slowly towards a minimal value, i.e., the lifetime of the entries increases. If a dynamic set with no empty entries is selected on a cache miss (i.e. a conflict occurs), a random replacement policy is used to replace one entry. The new entry is assigned with a uniformly random TTL value. Simultaneously, R_{TTL} is increased significantly, shortening the lifetime of all entries which increases the amount of empty entries and hence, reduces the probability of further conflicts. As before, R_{TTL} is slowly reduced towards the minimal value over time.

This results in a shark-fin shaped evolution of the reduction rate function as shown in Fig. 3. The choice of the initial value and the function with which the reduction rate develops is strongly dependent on the target architecture as well as the targeted security level and performance and hence needs to be optimized individually. We describe our implementation of CLEPSYDRACACHE using a CPU simulator in Section 7.

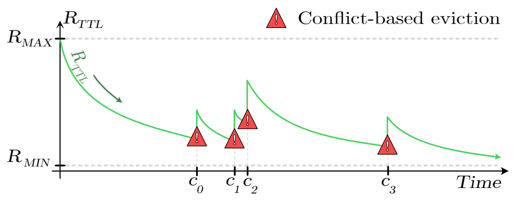

Figure 3: Evolution of the global TTL reduction rate that globally regulates the speed with which the TTL of each entry is reduced.

6 Security

In the following section, we evaluate the security properties of CLEPSYDRACACHE. The security of cache architectures using index randomization to thwart side-channel attacks has been extensively studied, and it has been shown that finding fully congruent eviction sets is not feasible in reasonable time [37, 46, 53]. However, recent work [8, 37] challenges the assumption that index randomization alone prevents cache attacks and proposes a variant of PRIME+PROBE called PRIME+PRUNE+PROBE which relies on partially congruent eviction sets. CLEPSYDRACACHE implements index randomization which prevents traditional PRIME+PROBE attacks by design since the attacker can no longer construct eviction sets trivially. The unique combination of randomization and cache decay adds a second layer of security, the benefits of which we will outline in the following. Our main focus lies on the PRIME+PRUNE+PROBE attack since this is the most relevant attack for index-randomization-based schemes. We consider other attack vectors in Appendix A.

6.1 Statistical Observations on TTL

We start the security analysis of CLEPSYDRACACHE by analyzing how an attacker may distinguish between time-based evictions and conflict-based evictions. The ability to tell those types of evictions apart enables the attacker to gain a more comprehensive insight to the cache state and therefore strengthen its capabilities.

In traditional caches, a conflict is detected in two phases: after populating the cache with a set of candidate addresses, the attacker accesses a target address which may evict a candidate address from the cache. We refer to this phase as the eviction phase. Subsequently in the probe phase, the attacker searches for the evicted address by measuring the access latency to all candidate addresses. A high latency in that phase indicates a cache miss and hence, the searched conflict. In the absence of noise, this works for an arbitrarily large set of candidate addresses. In CLEPSYDRACACHE however, the introduction of random time-based evictions introduces a level of uncertainty for the attacker: during the eviction phase, the access may or may not evict an address from the cache. Since the TTL is dynamically balanced to maintain empty entries in the cache, in most cases, no entry is evicted upon access. During the probe phase, the attacker then accesses a set of prior known-cached addresses. However, when they observe a cache miss, there is no way to distinguish whether the missed address was previously evicted based on timing or contention. That is, since the attacker's information is binary (hit or miss) and does not reveal whether a miss was based on an expired TTL or a conflict. This breaks the direct linkage between accesses and evictions in the general case. However, it must be noted, that for a small set of candidate addresses, the time between the eviction and the observation is short and hence, the likelihood of a time-based eviction is low. Therefore, the attacker can make an educated guess that the observed cache miss was caused by a conflict if the time delta between the initial access and the probe step is small. However, we show in Section 6.2 that for a useful attack, the attacker must probe extensively large sets of candidate addresses which make time-based evictions much more likely than conflicts.

A different aspect of cache conflicts in CLEPSYDRACACHE is that a sophisticated attacker might be able to monitor cache conflicts during the eviction phase by estimating R_{TTL}. Thereby, they aim to distinguish cache accesses that cause a cache conflict from those that do not. To achieve this, the attacker needs to access a set of addresses and periodically probe them for a cache miss. Note that continuously monitoring a single address is not feasible since a hit access resets the TTL of the respective entry to a new random value. The feasibility of this heavily depends on other actions taken by the attacker, the precision of the R_{TTL} estimation, and the overall noise level. Even if the attacker can observe when a conflict occurs, they cannot know during the probe phase *which* address caused the conflict. For example, if the attacker knows that a conflict occurred and during the probe phase two addresses result in a cache miss, they cannot distinguish which of these addresses was evicted by the conflict and which by timing. In the following, we assume that the attacker knows if an access caused a conflict even though in real world scenarios this may not be feasible.

6.2 Prime+Prune+Probe Attacks

The PRIME+PRUNE+PROBE [37] attack described in Section 3 builds a partially congruent eviction set $G = \cup_{i=1}^{w} G_i$, where addresses from G_i collide with the target address x in way i. The attack consists of a profiling phase, where

G is constructed, and an attack phase, where G is used to detect cache accesses by the victim. CLEPSYDRACACHE features three distinct characteristics that prevent efficient PRIME+PRUNE+PROBE attacks:

❶ The priming set needs to occupy the entire set of the target address since otherwise, it will always be stored in the empty cache entry. This increases the required size for the priming set.

❷ Time-based evictions add noise to the profiling and constrain the maximum time between two accesses to each address of the priming set. Conflicts during *prime* and *prune* further reduce this timeframe due to the dynamic scheduling of R_{TTL}.

❸ The generalized eviction set must occupy all possible target entries since otherwise, the target will always be stored in one of the remaining cache entries. Therefore, the eviction set needs to be much larger.

In the following, we give more details on these three characteristics and compare the success probability of PRIME+PRUNE+PROBE using CLEPSYDRACACHE to pure randomization approaches like SCATTERCACHE [53]. We assume that no noise from other processes is present. Finally, we give a conservative complexity estimation of profiling attacks. To put the theoretical security results into context, we use a 8 MiB, 16-way associative cache as reference. Similar cache configurations can be found in recent desktop-level CPUs. We further verified our theoretical results in a functional Python simulation.

Catching Probability (❶) During the *prime* and *prune* phases, the attacker repeatedly accesses a set of addresses k and removes those addresses that result in frequent cache misses, resulting in a set $k' \subseteq k$ of simultaneously known-cached addresses. The catching probability p_c describes the probability that after filling the cache with $|k'|$ addresses, the access to the target address x evicts an address from k'.

CLEPSYDRACACHE is different from most other cache architectures in that the cache is designed to never be completely filled. Though an attacker can deliberately fill many cache entries by accessing a huge amount of addresses in a very short time frame, the global TTL scheduling mechanism is designed to regulate the utilization and quickly invalidate entries in such a scenario. The TTL mechanism is designed to balance the utilization in a way that on average, each dynamic set contains at least one free entry and therefore, the amount of conflicts is minimized. Importantly, on a cache miss, the requested address will *always* be assigned to an empty cache entry, if one exists within the dynamic set of the address. Only if the dynamic set is completely filled, CLEPSYDRACACHE uses a random replacement policy to make room for the requested address. Hence, contrary to other randomized cache architectures, the catching probability will be zero if k' does not occupy the dynamic set of x, which in this case contains an empty entry. Since we assume that the attack takes place in a noise free scenario, we can assume that addresses cached

Table 1: Size of profiling set k' required to observe an eviction from accessing x with probability p_c in a 16-way cache with 8 MiB (131,072 entries). Both schemes use a random replacement policy. Resulting cache utilization by k' in parentheses.

p_c	CLEPSYDRACACHE	SCATTERCACHE [53]
1%	98,312 (75.0%)	1,311 (1.0%)
50%	125,520 (95.8%)	65,536 (50.0%)
90%	130,216 (99.3%)	117,965 (90.0%)
95%	130,656 (99.7%)	124,519 (95.0%)

prior to the attack will quickly be eliminated based on timing. Only a few very frequently accessed addresses by other processes will remain in the cache. If the attacker relies on these addresses from other processes, these addresses can, by the rules of transitivity, be considered part of k'.

The dynamic set of x is determined by the addressing function and in our attacker model is assumed to be indistinguishable from truly random. Hence, the probability that after filling the cache with $|k'|$ non-conflicting entries, the dynamic set of x is completely filled follows the hypergeometric distribution and calculates as

$$p_c(k') = \frac{\binom{|k'|}{w}}{\binom{N}{w}}. \tag{1}$$

Since the attack *catches* the conflict if and only if the dynamic set of x is filled, the above probability is also the catching probability of the profiling step.

Another way to think about this problem is that every address of the probing set k' taints the entry in which it resides. All cache entries that are empty remain untainted. Since the addressing scheme assigns random entries to the addresses of k', we receive a cache that is partitioned to $|k'|$ tainted entries at random positions and the remaining $N - |k'|$ untainted entries. Importantly, due to the random placement, these entries are equally distributed across the cache ways. The access to address x selects a random entry from each way, which - due to the equal distribution of tainted and untainted addresses - can be considered as choosing w random entries from the cache. Only if all w chosen entries are tainted, is the profiling successful.

Tab. 1 shows the size of k' required to observe an eviction by the access to the victim address x with probability p_c in a 16-way cache for solely randomization-based caches like SCATTERCACHE [53] compared to CLEPSYDRACACHE. The results for SCATTERCACHE are calculated using the findings by Purnal *et al.* [37]. The results show that the attacker requires a much larger profiling set to achieve a decent catching probability with CLEPSYDRACACHE.

Time-Based Evictions (❷) During the profiling, addresses in k' are selectively narrowed down from the initial set of random addresses k. As discussed in ❶, achieving a reasonable catching probability requires filling well over half of the cache storage. The attacker must aim to minimize the time-based evictions and therefore avoid conflicts unrelated to the target address. This keeps R_{TTL} low and thus enables the attacker to make useful observations during the probe phase. For an initial priming set k, the expected number of conflicts during the prime and prune can be computed using Eq. 2.

$$\mathbb{E}[\#c] \geq \sum_{i=1}^{|k|} \sum_{j=1}^{\infty} \left(\frac{\binom{i}{w}}{\binom{N}{w}} \right)^{j} \tag{2}$$

The amount of conflicts increases exponentially with the size of k. Using the above example cache parameters, priming the cache to a 50% catching probability of the target access would require more than 125,000 accesses and result in approximately 4,682 conflicts during the priming. In the original PRIME+PRUNE+PROBE attack, the attacker starts by accessing the entire set of addresses during the prime step. Subsequently, the set is pruned by re-accessing it until no more cache misses occur. However, the vast number of conflicts produced by priming the cache would increase R_{TTL} to a level where entries are evicted by timing rapidly, undoing any progress made. Hence, a better approach is to do the prime and prune step incrementally. Therefore, the attacker adds addresses to k until a conflict occurs. Then, they re-access all current addresses from k which reduces the probability of time-based evictions in k and gives time for R_{TTL} to recover. In the following we assume that the R_{TTL} increase caused by one cache conflict can be compensated by re-accessing the primed addresses once. In reality, further accesses may be necessary.

If we assume that the attacker managed to prime the cache to a sufficient degree and the access to x results in a conflict (increasing R_{TTL}), the attacker must now probe all addresses from k' to find the one that was evicted by x. As discussed above, even though the attacker may know *that* a conflict occurred by monitoring R_{TTL}, they do not know which address of k' was evicted. When the attacker finds the first cache miss in k', they cannot know if that address has been evicted due to the conflict with x, or due to an expired TTL. The attacker has no other option than to add all potential candidates to G and - if required - filter false positives once a sufficiently large G is constructed.

Eviction Probability (❸) Being able to carry out cache attacks requires the attacker to find many entries that can collide with x in the cache. We now investigate how many such addresses are required to detect the victim access with probability p_e.

Each address of G will occupy a conflicting cache entry with probability $\frac{1}{w}$. However, only if the entire dynamic set of x is occupied after accessing all addresses from G, the access to x will result in a conflict and therefore be observable. Thus, the probability calculates as

$$p_e = \left(1 - \left(1 - \frac{1}{w} \right)^{\frac{|G|}{w}} \right)^{w}. \tag{3}$$

Table 2: Size of the generalized eviction set G required to observe an eviction from accessing x with probability p_e in a 16-way cache. Both schemes use a random replacement policy.

p_e	CLEPSYDRACACHE	SCATTERCACHE [53]
1%	344	3
50%	784	172
90%	1,247	571
95%	1,425	743

Tab. 2 lists the required size of G to observe an eviction by x with probability p_e for a 16-way cache and compares it to pure randomization designs on the example of SCATTERCACHE [53].

Profiling Performance Estimation. We now conservatively estimate the time it would take an attacker to construct a generalized eviction set for CLEPSYDRACACHE and compare it to purely randomized caches. Therefore we assume a cache hit latency of $t_{hit} = 10\ ns$ and a miss latency of $t_{miss} = 20\ ns$. We assume that the attacker knows when a conflict occurred e.g. by using the statistical approach described in Section 6.1, and, therefore can use the described methodology in ❷. Each access to a new address results in a cache miss and is hence counted with 20 ns. Accessing the i-th new address during the construction of k results in a conflict with an already cached address of k with probability p_c. Furthermore, accessing the address evicted by the conflict leads to a new conflict with probability p_c again, repeating the process. Hence, the probability for n conflicts calculates as $p_c + p_c^2 + ... + p_c^n$. When a conflict occurs from accessing a new address, the attacker re-accesses all prior entries of k until all addresses from k, including the one that has been evicted by the conflict, are cached. For real implementations, the attacker would need to further keep re-accessing entries from k until R_{TTL} recovered to a low level. Hence, we assume that in each iteration the attacker adds an address to k and if this access causes a conflict, the attacker re-accesses k with which results in $|k|_i - 1$ cache hits and one cache miss. We do not consider the noise effects introduced by time-based evictions or noise by other processes which would make the profiling more complicated in reality. We assume that the attacker aims to build an eviction set G with $p_e = 50\%$. Overall, we calculate the time for one iteration of profiling as

$$t_{pp} = \sum_{i=1}^{k} \left(t_{miss} + \sum_{j=1}^{\infty} p_c(i)^j \cdot ((i-1) \cdot t_{hit} + t_{miss}) \right). \tag{4}$$

We estimate the time to construct G as

$$t_k = |G| \cdot \frac{1}{p_c(k)} \cdot t_{pp}. \tag{5}$$

Eq. 5 is composed of the size of G required for a probabilistic eviction set with eviction probability p_e, the probability that a conflict is *caught* during profiling (p_c), and the time for one round of profiling. Using this estimation, the minimal profiling time for a probabilistic eviction set with $p_e = 50\%$ and the 8 MB cache with 16 ways for CLEPSYDRACACHE is 4,105 s (1.1 hours) if k occupies 70% of the cache. If the attacker can only fill up to 50% of the cache, the construction of G takes 69,575 s (19.3 hours). If the attacker attempts to fill more than 70% of the cache, the conflicts will lead to a higher profiling time. The same estimation for SCATTERCACHE with random replacement policy yields a profiling time of 0.45 s at 1% cache utilization by k (a small k reduces the impact of conflicts during pruning). There are no directly comparable numbers available for the recently proposed Mirage [41] and careful analysis is required to assess the security of Mirage against PRIME+PRUNE+PROBE. However, Mirage [41] is similar to CLEPSYDRACACHE in that the over provisioned tag store has empty entries and thus, conflicts are avoided most of the time. Therefore, the characteristics regarding the catching probability are similar in principle. Mirage does not have time-based evictions which allows for more optimal prime and prune without re-accessing entries from k. Since our method uses very conservative assumptions, we expect all designs to have a much higher profiling time in reality. Our results show that CLEPSYDRACACHE can extend the profiling time by several orders of magnitude compared to SCATTERCACHE. Since CLEPSYDRACACHE does not introduce indirection on the cache lookup path, the lookup latency is only constrained by the randomization function which is present in all randomized caches including Mirage and SCATTERCACHE.

7 Implementation

Our implementation of CLEPSYDRACACHE is consists of two parts. First, we implement a functional model of CLEPSYDRACACHE using gem5 [33], which is the most established simulator for CPU microarchitectures. Secondly, to understand the hardware cost, we take the effort to simulate a proof-of-concept hardware design of CLEPSYDRACACHE using 65nm CMOS technology in Cadence Spectre.

Intuitively, the performance of CLEPSYDRACACHE heavily depends on time-to-live range supported by the implementation. While the TTL reduction rate R_{TTL} dynamically adapts the time-to-live to some extent, one should analyze the average lifetime of cache entries in a traditional cache for the given platform. We analyzed this time using an unmodified gem5 and different workloads. It shows that in regions of heavy cache usage, each entry hardly lives longer than 50 ms without being accessed. Though outliers exist, we expect them to hardly affect the performance since they are rarely accessed anyway. Following the setup chosen in related work [53], our simulation features a two level cache hierarchy where the 1MB L2 cache is a CLEPSYDRACACHE. Further information on the gem5 configuration is given in Section 8.

7.1 *gem5* Implementation

For the randomization of the addressing scheme, we use a round reduced version of the lightweight block cipher PRINCE [7] with three rounds and randomize the input in each cache way by xor-ing a way-specific 64-bit secret to the input of the cipher. We further sample the bits used for the cache index from the entire output of the randomization function instead of using, e.g., the least significant bits as motivated in the security analysis of the index-randomization function in Appendix A. Since PRINCE operates on 64-bit blocks, we use the entire memory address with zeroed offset bits as input, and therefore get a 64-bit output. The bits used for indexing do not need to be stored as they are implicitly indicated by the location. However, the tag size in this scenario increases by 6 bits due to the obfuscated offset bits. This overhead can be avoided by using a format preserving encryption scheme instead of PRINCE as discussed in Section 5.3.

For the TTL, we modified the base cache class of gem5 and implemented a counter for each cache entry. The TTL of each entry is reduced based on a periodic event. Whenever a conflict occurs – i.e., a new address cannot be cached without evicting existing data – the event reducing the TTL for each entry is immediately triggered. The time until the next event is quartered. Every time the event executes without a conflict, the time until the next event scheduling is slightly increased by adding a constant value. This results in a behavior as described for R_{TTL} (c.f. Fig. 3).

7.2 Efficient Hardware Implementation

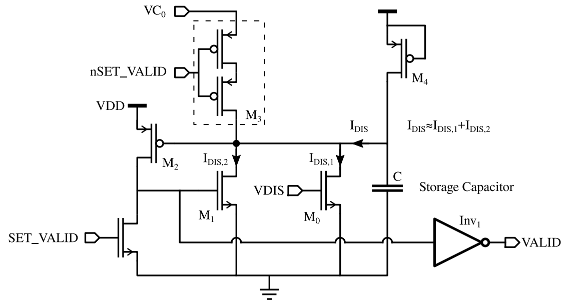

Figure 4: Schematic of the proposed delay element.

In this section, we introduce a proof-of-concept analog delay circuit that demonstrates the feasibility of highly efficient implementation of the TTL mechanism. In general, the placement of analog elements on digital circuitry requires careful decoupling, for example through shielding by ground planes and separating the supply voltage. Since these challenges are highly dependent on the technology and layout, they remain for the CPU developer. It is also possible to implement the TTL mechanism using digital counters for each entry. However, the analog solution is more area preserving as it is much

simpler. We have designed and simulated the analog delay element in a 65-nm CMOS technology. The delay element is able to achieve TTLs of up to 50 ms (in terms of transistor switching speed). The basic idea behind the proposed delay element is to charge a capacitor to an initial value VC_0 that determines the TTL and to discharge it by a defined current I_{DIS} until a threshold voltage is reached. We expect, that a metal-insulator-metal (MIM) capacitor with a capacitance of up to 100 fF can be integrated within the metal stack over each cache entry without competing for silicon area. The required maximum delay time of 50 ms consequently demands a very low current of $I_{DIS} < 1$ pA. Such small currents can be generated in an area-efficient manner by making use of the leakage current of transistors, that are either completely off (gate to source voltage = 0 V) or operate in the deep subthreshold region with very low gate to source voltages. The schematic of the proposed delay element is shown in Fig. 4. The signal SET_VALID is equivalent to the address line of the cache entry. When the entry is selected, SET_VALID is high, while the inverted signal nSET_VALID is low. As a consequence, the storage capacitor C is charged to VC_0 via transistor M_3. The value of VC_0 determines the delay time of the delay element and thus the TTL of the cache entry. M_3 is implemented by two series connected transistors by a technique called stack forcing in order to reduce unwanted leakage currents through M_3 which would otherwise affect the delay time.
After the cache entry is deselected (SET_VALID = 0), C is discharged by the leakage currents of M_0 and M_1. The global low voltage signal VDIS ($\ll 100$ mV) is used to adjust the discharge current through M_0. It is used to control the TTL-reduction rate R_{TTL}.
We found that the addition of M_4 improves the performance of the delay element in two ways. First, it increases the delay time without increasing the area of the capacitor and, second, it allows to compensate for the temperature variation of the leakage currents of M_0 and M_1 if all contributing transistors are carefully parameterized. M_1 and M_2 form a positive feedback loop referred to as a pseudo-CMOS-thyristor [17]. After C is discharged below the threshold voltage of M_2, M_2 charges the gate of M_1 which then becomes conductive and discharges the remaining charge of C in ≈ 10ns. The advantage of this feedback loop is the drastic reduction of short circuit current in the output inverter Inv_1, which would be severe if the complete discharge happened on a ms-timescale. The simulated delay time of this circuit as a function of VC_0 is shown in Fig. 5 for three different temperatures. For $0.89V < VC_0 < 1.2$ V delay times between 1 ms and almost 50 ms are achieved. It should be noted that the observed moderate variation with temperature is actually beneficial, as it adds randomness to the TTL.
To estimate the area required to implement this circuit, we have designed the layout shown in Fig. 6. With an area of $3.5\mu m \times 1.4\mu m = 4.9\mu m^2$ we can estimate the area overhead

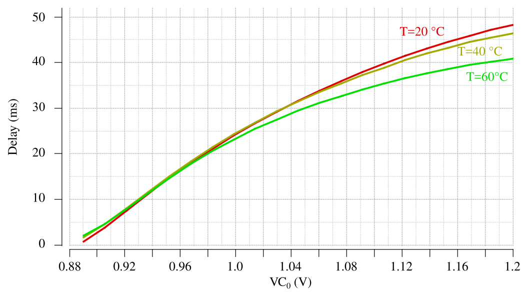

Figure 5: Simulated delay of the proposed delay element as a function of the initial capacitor voltage VC_0. VDIS was set to 0 V in this simulation.

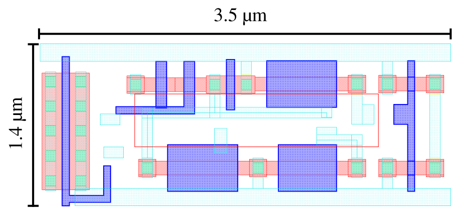

Figure 6: Layout of the proposed delay cell (excluding the MIM capacitor).

of this circuit: Assuming the size of an SRAM cell in 65-nm CMOS technology to be $0.5\mu m^2$ as reported in [34], the area of a 128 bit cache entry (64 bit data, ≈ 60 bit tag plus some flags) is $64\mu m^2$ (without overhead). We therefore estimate the area overhead of the analog TTL implementation circuit to be $< 8\%$. In the simulation we used a MIM capacitor with an area of $30\mu m^2$ which can be conveniently placed in the metal stack above the cache entry.
If the charge time of the capacitor t_c is too long, it may increase the access time of the cache. We have therefore simulated how fast the capacitor is charged. The result of our simulations was that t_c depends on VC_0. For $VC_0 = 1.2$ V the recharge time was 8 ns while for the minimal value of $VC_0 = 0.89$ V, t_c was 41 ns. This corresponds well to the reported L3 cache latency of an i7 CPU [27] of 40 to 300 cycles (13 ns to 100 ns with a clock frequency of 3 GHz).
The generation of the analog control signals VC_0 and VDIS is not within the scope of this work since many suitable digital-to-analog converters (DAC) have been published. DACs can be realized with a small area and power consumption. E.g. in [26] a current steering DAC with an area consumption of $0.1\,mm^2$ in 65-nm CMOS technology and a power consumption of 12 mW is reported.

8 Evaluation

We evaluate ClepsydraCache using the gem5 CPU simulator [33]. In order to compare the performance of our new cache concept we execute popular benchmarks including

Parsec [6] and SPEC CPU 2017 [20]. We provide a detailed performance evaluation of ClepsydraCache and compare it to traditional caches. We execute the Parsec benchmark using gem5's full system mode to get the most precise performance result. We simulate Ubuntu 18.04 LTS with kernel 4.19.83. For the SPEC CPU 2017 Benchmark we use gem5's Syscall Emulation (SE) mode. Due to the vast resource usage and high simulation time it is not feasible to use full system mode. The SE mode also provides high accuracy in performance measurements, but omits the simulation of the entire operating system. Other related work often only chooses representative code slices of the benchmark [46, 53].

For the purpose of the evaluation we use gem5's default cache implementation as reference, hereinafter referred to as *classic*. We focus on the comparison of ClepsydraCache and classic caches since the most related concepts, e.g., ScatterCache [53] are close to the classic cache performance. As discussed earlier in this paper, Mirage [41] changes the critical path of the cache access and may therefore induce additional overhead. In ClepsydraCache, the access latency is only affected by the randomization function which must be low-latency in order to avoid delays. Following related work [53], both the classic cache and ClepsydraCache have a two level cache hierarchy with a L1 and and a mostly inclusive L2 cache. Based on our observations regarding the average lifetime of cache entries (Section 7), we have chosen a maximum TTL of 50ms for the evaluation. We describe the simulation details including cache associativity and choosen benchmark workloads in Appendix B.

Performance. We use clock cycles as a metric to measure the absolute performance of the benchmarks. For the Parsec benchmark suite, the overall performance results are shown in Fig. 7. Overall, the performance of ClepsydraCache matches the performance of traditional caches with a penalty between -0.37% and +5.25%. On average the performance penalty is at 1.38%.

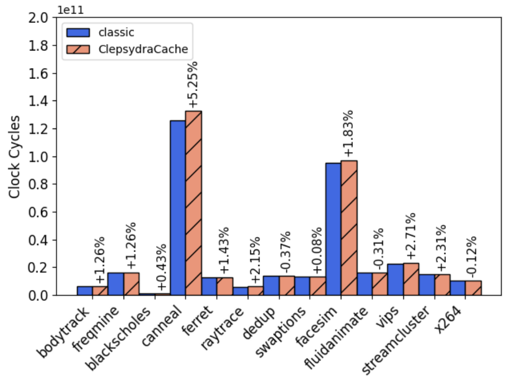

Figure 7: Comparison of clock cycles for programs from the Parsec benchmark suite.

While the fact that ClepsydraCache improves the performance for some benchmarks appears counterintuitive at first glance, the reason for this can be found by looking at the average miss latency depicted in Fig. 8. It shows that with the exception of the *streamcluster* benchmark, ClepsydraCache consistently lowers the average miss latency and thus, compensates for the slightly increased miss rate. The reason for that can be found in the writeback of dirty

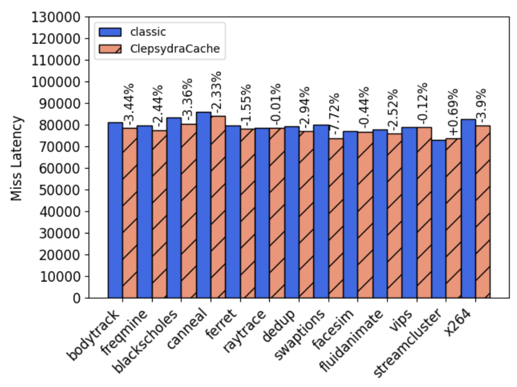

Figure 8: Comparison of the average miss latency for the programs from the Parsec benchmark suite.

cache entries. While a traditional cache performs writebacks on modified data when a conflict occurs, ClepsydraCache performs writebacks based on timing and thus is independent of any cache misses. Hence, the miss latency on traditional caches includes the time needed for a potential writeback whereas ClepsydraCache usually invalidates and writes back the entries before a conflict occurs. Therefore, nearly all benchmarks have a slightly lower average miss latency. The *streamcluster* benchmark is the one outlier of this observation. The workload characteristic given in [6] reveals that this benchmark performs exceptionally few writes and hence, hardly any cache entries need to be written back. Thus, the influence of this effect is minimized and the occurrence of *some* conflict-based evictions leads to a slightly higher average miss latency for ClepsydraCache. The low overall miss latency of *streamcluster* supports this finding.

While the performance increase of ClepsydraCache in gem5 appears like a genuine improvement over traditional caches, from a security point of view a constant time cache would be much preferable. Though in our simulation the timing difference between a cache miss with and without a writeback was sufficiently small to not be reliably measurable, this may not be the case with all caches. Equally, it may also be that some cache implementations do not have this timing difference. Even if we assume that the miss latency is constant between the classic cache and ClepsydraCache in our evaluation, the overall average miss latency only increases by about 2.3% on average. This would only marginally affect the performance data shown in Fig. 7.

As one would expect, ClepsydraCache experiences a

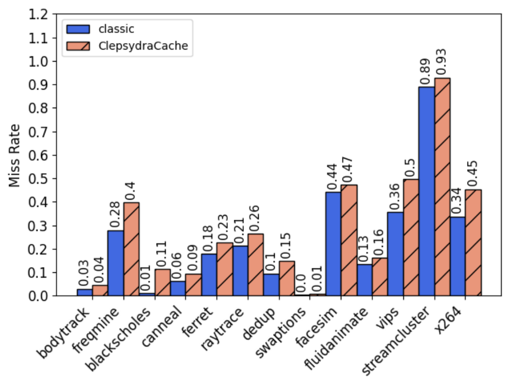

Figure 9: Comparison of the miss rate for the programs from the Parsec benchmark suite.

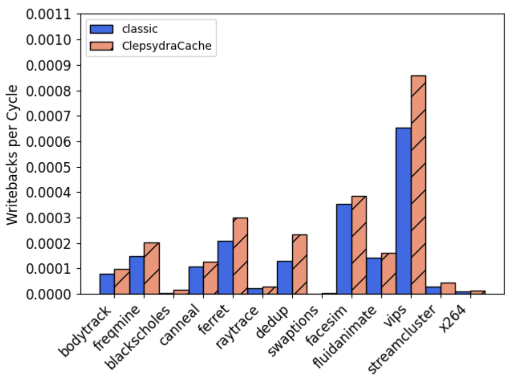

Figure 10: Writebacks per cycle for ClepsydraCache using the Parsec benchmark suite.

Table 3: Benchmark results of the SPEC CPU 2017 benchmark suite using ClepsydraCache in comparison to a traditional cache.

Benchmark	Clock	Miss Rate (difference)	Avg. Miss Latency	Conflict Evictions
Deepsjeng	-1.72%	±0%	-4.38%	-94.13%
Exchange2	+0.04%	+8%	-0.78%	-100%
Gcc	+0.39%	+5%	-2.87%	-95.71%
Leela	+0.11%	+2%	-2.34%	-95.55%
Perlbench	±0%	+3%	-0.2%	-100%
x264	+0.24%	+2%	-1.36%	-98.93%
Xalancbmk	+0.3%	+3%	-2.35%	-98.61%
Xz	-0.11%	+1%	-2.83%	-95.54%

slightly increased miss rate compared to classic caches. That is, since sometimes entries are evicted based on expired TTLs, but are later referenced and need to be re-loaded. Fig. 9 shows the miss rate for Parsec, i.e. the ratio of cache accesses that resulted in a cache miss. The hit rate is the inverted value and calculates as $1 - \text{Miss Rate}$. The miss rate is generally close to the miss rate of traditional caches. Benchmarks with a low miss rate for classic caches experience a low miss rate for ClepsydraCache. Table 3 reports the results for SPEC CPU 2017. Generally, the results we obtained using SPEC CPU 2017 are very similar to the ones reported for Parsec.

We investigate the number of writebacks issued with and without ClepsydraCache in Fig. 10. Since more entries are written back using ClepsydraCache, the writebacks per cycle are slightly increased. However, the main incentive for this evaluation was to not overload the writeback buffer. The increase in writebacks per cycle is only minor and hence, hardly affects the size of the hardware logic required for writebacks (e.g. the writeback buffer size). Finally, we compare the average lifetime of cache entries for classical caches and ClepsydraCache.Our implementation limits the maximum lifetime of an entry to 50 ms. This value is chosen based on the analysis of the average entry-lifetime in a classic cache for our setup. It shows that the average lifetime of entries in ClepsydraCache is very similar to the one in classic caches.

Security. Fig. 11 depicts the rate of cache conflicts in ClepsydraCache compared to traditional caches using the PARSEC benchmark suite. Cache conflicts are an important occasion for an attacker to learn conflicting addresses and construct eviction sets. With ClepsydraCache, the number of cache conflicts is reduced more than 90% for all Parsec benchmarks (on average 94.6%) which greatly reduces the chances of an attacker observing an eviction. As discussed in our security analysis, provoking a targeted cache conflict is very challenging with ClepsydraCache. Hence, the cost for a successful attack is not proportional to the observed conflicts. To verify that our ClepsydraCache implementation is secure against Prime+Prune+Probe attacks, we implement the attack and execute it in the gem5 SE mode. Using a pure randomized cache design which mimics ScatterCache [53], the construction of an eviction set with $p_e = 90\%$ took 2.15 seconds. Using ClepsydraCache we were not able to find conflicting addresses. We aborted the attack without success after 77 simulated seconds (two days of simulation). After that time, not a single conflicting address has been found. Furthermore, the small timing difference between cache conflicts with and without writebacks was not measurable in our experimental setup. However, if one were to implement a side-channel secure cache, such timing differences should be avoided. Note that it is easily feasible to reduce the amount of conflict-based evictions even further by reducing the maximum time-to-live, albeit at the cost of a higher miss rate and therefore, reduced performance.

9 Conclusion

We presented ClepsydraCache, a novel cache architecture that eliminates cache attacks by preventing the attacker from constructing (generalized) eviction sets. Our solution is purely architectural and therefore backwards compatible to the entire software stack. We are the first to explore an

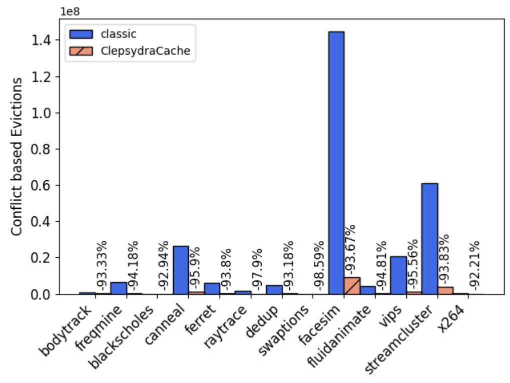

Figure 11: Comparison of conflict-based evictions for the programs from the Parsec benchmark suite.

analog proof-of-concept hardware design in 65nm CMOS technology which facilitates an area efficient implementation of CLEPSYDRACACHE. We showed that CLEPSYDRACACHE has minimal performance overhead and in some cases even outperforms traditional cache architectures using representative workloads.

10 Acknowledgements

This work is funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy - EXC 2092 CASA - 390781972 and by the DFG under the Priority Program SPP 2253 Nano Security (Project RAINCOAT - Number: 440059533). "Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the funding agencies". Date of this document: August 12th, 2022.

References

[1] 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.

[2] Mark Allman, Vern Paxson, and W. Richard Stevens. TCP congestion control. *RFC*, 2581:1–14, 1999.

[3] Mihir Bellare, Thomas Ristenpart, Phillip Rogaway, and Till Stegers. Format-preserving encryption. In *International workshop on selected areas in cryptography*, pages 295–312. Springer, 2009.

[4] Mihir Bellare, Phillip Rogaway, and Terence Spies. The FFX mode of operation for format-preserving encryption. *NIST submission*, 20:19, 2010.

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

[6] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 72–81, 2008.

[7] 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 - extended abstract. 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.

[8] Thomas Bourgeat, Jules Drean, Yuheng Yang, Lillian Tsai, Joel 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.

[9] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software grand exposure: SGX cache attacks are practical. In *11th USENIX Workshop on Offensive Technologies (WOOT 17)*. USENIX Associaton, 8 2017.

[10] Ang Chen, W. Brad Moore, Hanjun Xiao, Andreas Haeberlen, Linh Thi Xuan Phan, Micah Sherr, and Wenchao Zhou. Detecting covert timing channels with time-deterministic replay. In Jason Flinn and Hank Levy, editors, *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014*, pages 541–554. USENIX Association, 2014.

[11] Jie Chen and Guru Venkataramani. CC-Hunter: Uncovering covert timing channels on shared processor hardware. In *47th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2014, Cambridge, United Kingdom, December 13-17, 2014*, pages 216–228. IEEE Computer Society, 2014.

[12] Goran Doychev and Boris Köpf. Rigorous analysis of software countermeasures against cache attacks. In *PLDI '17: ACM SIGPLAN Conference on Programming Language Design and Implementation*, New York, NY, USA, 6 2017. ACM.

[13] Hongyu Fang, Sai Santosh Dayapule, Fan Yao, Milos Doroslovacki, and Guru Venkataramani. Prefetch-guard: Leveraging hardware prefetches to defend against cache timing channels. In *2018 IEEE International Symposium on Hardware Oriented Security and Trust, HOST 2018, Washington, DC, USA, April 30 - May 4, 2018*, pages 187–190. IEEE Computer Society, 2018.

[14] Abraham Gonzalez, Ben Korpan, Jerry Zhao, Ed Younis, and K Asanovic. Replicating and mitigating spectre attacks on an open source risc-v microarchitecture. In *Workshop on Computer Architecture Research with RISC-V (CARRV)*, 2019.

[15] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+Flush: A fast and stealthy cache attack. In *Proceedings of the 13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment - Volume 9721*, DIMVA 2016, page 279–299, Berlin, Heidelberg, 2016. Springer-Verlag.

[16] Berk Gülmezoglu, Ahmad Moghimi, Thomas Eisenbarth, and Berk Sunar. FortuneTeller: Predicting microarchitectural attacks via unsupervised deep learning. *CoRR*, abs/1907.03651, 2019.

[17] Gyudong Kim, Min-Kyu Kim, Byoung-Soo Chang, and Wonchan Kim. A low-voltage, low-power CMOS delay element. *IEEE Journal of Solid-State Circuits*, 31(7):966–971, 1996.

[18] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. Cache attacks on Intel SGX. In *EuroSys '17: Twelfth EuroSys Conference 2017*, New York, NY, USA, 4 2017. ACM.

[19] J.L. Hennessy and D.A. Patterson. *Computer Organization and Design: The Hardware / Software Interface*. Elsevier Science, 2014.

[20] John Henning. *SPEC CPU 2017 Documentation*. Standard Performance Evaluation Corporation (SPEC), Gainesville, VA, 6623 2021-04-07 edition, April 2021.

[21] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Cross processor cache attacks. In *ASIA CCS '16: ACM Asia Conference on Computer and Communications Security*, New York, NY, USA, 5 2016. ACM.

[22] Stefanos Kaxiras, Zhigang Hu, and Margaret Martonosi. Cache decay: Exploiting generational behavior to reduce cache leakage power. In *Proceedings 28th annual international symposium on computer architecture*, pages 240–251. IEEE, 2001.

[23] Georgios Keramidas, Alexandros Antonopoulos, Dimitrios N Serpanos, and Stefanos Kaxiras. Non deterministic caches: A simple and effective defense against side channel attacks. *Design Automation for Embedded Systems*, 12(3):221–230, 2008.

[24] Lars R. Knudsen. Truncated and higher order differentials. In Bart Preneel, editor, *Fast Software Encryption: Second International Workshop. Leuven, Belgium, 14-16 December 1994, Proceedings*, volume 1008 of *Lecture Notes in Computer Science*, pages 196–211. Springer, 1994.

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

[26] Y. Kwon, S. Lee, Y. Jeon, and J. Kwon. A 6b 1.4GS/s 11.9mW 0.11mm^2 65nm CMOS DAC with a 2-D INL bounded switching scheme. In *2010 International SoC Design Conference*, pages 198–200, 2010.

[27] David Levinthal. Performance analysis guide for Intel Core i7 processor and Intel Xeon 5500 processors. *Intel Performance Analysis Guide*, 30:72, 2009.

[28] Ankur Limaye and Tosiron Adegbija. SPEC CPU2017 Command Lines. Online, 2018.

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

[30] Fangfei Liu, Qian Ge, Yuval Yarom, Frank McKeen, Carlos V. Rozas, Gernot Heiser, and Ruby B. Lee. CATalyst: Defeating last-level cache side channel attacks in cloud computing. In *2016 IEEE International Symposium on High Performance Computer Architecture, HPCA 2016, Barcelona, Spain, March 12-16, 2016*, pages 406–418. IEEE Computer Society, 2016.

[31] Fangfei Liu, Hao Wu, Kenneth Mai, and Ruby B. Lee. Newcache: Secure cache architecture thwarting cache side-channel attacks. *IEEE Micro*, 36(5):8–16, 2016.

[32] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-level cache side-channel attacks are practical. In *2015 IEEE Symposium on Security and Privacy (SP)*. IEEE, 5 2015.

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

[34] S. Ohbayashi, M. Yabuuchi, K. Nii, Y. Tsukamoto, S. Imaoka, Y. Oda, T. Yoshihara, M. Igarashi, M. Takeuchi, H. Kawashima, Y. Yamaguchi, K. Tsukamoto, M. Inuishi, H. Makino, K. Ishibashi, and H. Shinohara. A 65-nm SoC embedded 6T-SRAM designed for manufacturability with read and write operation stabilizing circuits. *IEEE Journal of Solid-State Circuits*, 42(4):820–829, 2007.

[35] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of AES. In *Topics in Cryptology – CT-RSA 2006*, pages 1–20. Springer Berlin Heidelberg, Berlin, Heidelberg, 1 2006.

[36] Dan Page. Partitioned cache architecture as a side-channel defence mechanism. *IACR Cryptol. ePrint Arch.*, 2005:280, 2005.

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

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

[39] Moinuddin K Qureshi and Yale N Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, pages 423–432. IEEE, 2006.

[40] Himanshu Raj, Ripal Nathuji, Abhishek Singh, and Paul England. Resource management for isolation enhanced cloud services. In *Proceedings of the 2009 ACM Workshop on Cloud Computing Security*, CCSW '09, page 77–84, New York, NY, USA, 2009. Association for Computing Machinery.

[41] Gururaj Saileshwar and Moinuddin Qureshi. MIRAGE: Mitigating conflict-based cache attacks with a practical fully-associative design. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, August 2021.

[42] Daniel Sanchez and Christos Kozyrakis. Scalable and efficient fine-grained cache partitioning with vantage. *IEEE Micro*, 32(3):26—-37, 1 2012.

[43] Rich Schroeppel. Hasty pudding cipher specification. In *First AES Candidate Workshop*, 1998.

[44] Kang G. Shin and Parameswaran Ramanathan. Real-time computing: A new discipline of computer science and engineering. In *Proceedings of IEEE, Special Issue on Real-Time Systems*. IEEE, 1994.

[45] Wei Song and Peng Liu. Dynamically finding minimal eviction sets can be quicker than you think for Side-Channel attacks against the LLC. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*, pages 427–442, Chaoyang District, Beijing, September 2019. USENIX Association.

[46] Qinhan Tan, Zhihua Zeng, Kai Bu, and Kui Ren. PhantomCache: Obfuscating cache conflicts with localized randomization. In *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020*. The Internet Society, 2020.

[47] Eran Tromer, Dag Arne Osvik, and Adi Shamir. Efficient cache attacks on AES, and countermeasures. *J. Cryptol.*, 23(1):37–71, 2010.

[48] Stephan Van Schaik, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Malicious management unit: Why stopping cache attacks in software is harder than you think. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 937—-954, 1 2018.

[49] Pepe Vila, Boris Köpf, and José F. Morales. Theory and practice of finding eviction sets. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*, pages 39–54. IEEE, 2019.

[50] 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, New York, NY, USA, 6 2016. ACM.

[51] 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, New York, New York, USA, 1 2007. ACM Press.

[52] 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.

[53] Mario Werner, Thomas Unterluggauer, Lukas Giner, Michael Schwarz, Daniel Gruss, and Stefan Mangard. ScatterCache: Thwarting cache attacks via cache set randomization. In *28th USENIX Security Symposium (USENIX Security 19*, pages 675—-692, 1 2019.

[54] Yuejian Xie and Gabriel H Loh. PIPP: Promotion/insertion pseudo-partitioning of multi-core shared caches. *ACM SIGARCH Computer Architecture News*, 37(3):174—-183, 1 2009.

[55] Mengjia Yan, Yasser Shalabi, and Josep Torrellas. ReplayConfusion: Detecting cache-based covert channel attacks using record and replay. In *49th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2016, Taipei, Taiwan, October 15-19, 2016*, pages 39:1–39:14. IEEE Computer Society, 2016.

[56] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack. In Kevin Fu and Jaeyeon Jung, editors, *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014*, pages 719–732. USENIX Association, 2014.

A Security: Further Attacks

In this Appendix, we discuss attack scenarios beyond PRIME+PRUNE+PROBE.

Evict+Time Attacks. In an EVICT+TIME [35] attack, the attacker measures the execution time of a victim program before evicting a specific cache set. The attacker then executes the victim program again and measures whether the timing differentiates from the first run. If so, the attacker can conclude that the data in the set was accessed by the victim process during execution. This attack does not work with CLEPSYDRACACHE. Next to the requirement of performing targeted evictions of data from a victim process, the randomization of the TTL induces random deviations to the execution time of the victim program. This does not only affect the targeted cache set but any memory access performed during the execution of the victim process. Hence, the attacker has no indication whether timing deviations in the execution time of a program are a result of targeted evictions resulting from conflicts or random evictions resulting from expired TTL values.

Index Randomization. The TTL feature in combination with index randomization successfully prevents a useful observation of a cache conflict. However, the design of the index randomization function described in Section 5.3 requires that an attacker is not able to construct additional conflicting addresses from already observed ones. In particular, it implicitly requires that hardness of the problem deducing information about the key from the observation that two addresses are conflicting. As conflicting addresses correspond to certain bits being identical in the output of the index randomization function, this situation is similar to a truncated differential attack on block ciphers as originally introduced by Knudsen [24]. Here the attacker could in principle try to guess part of the first round key and, based on the guess, choose specific input differences that increase the probability of the truncated differential and thus the probability of creating conflicting addresses. This implies that the probability of the truncated differential should not be too large. Note that the attacker cannot choose the truncated output difference to consider, but this is implicitly defined by the design. In particular one possibility to ensure small probability is to spread the output bits used for defining the index across the entire output of the cipher and in particular across as many S-boxes as possible.

The rather limited data complexity an attacker can use in combination with the high level of noise created by the TTL feature suggest that a reduced round version of secure block cipher suffices to ensure security. However, analyzing the details of such an approach is beyond the scope of this work.

Denial of Service Since the reduction rate R_{TTL} of the TTL reduction depends on the number of conflicts that occurred, an attacker could effectively clear all cache entries by making a huge amount of memory accesses to uncached addresses. This behavior would cause frequent conflicts and hence increase the reduction rate, evicting all entries in a short time period. The attacker could further keep accessing new memory addresses and hence, effectively disable the cache as new entries are deleted relatively fast due to the high TTL reduction rate.

In a traditional cache, the attacker could achieve a similar behavior by accessing many uncached memory addresses and therefore evicting existing entries from other processes. Importantly, no information can be leaked during a DoS attack.

B Configuration Details

The Parsec benchmarks are executed in gem5's full system mode which simulates all of the hardware from the CPU to the I/O devices and gives a precise performance result. We simulate Ubuntu 18.04 LTS with kernel version 4.19.83. However, due to the vast resource usage and the corresponding simulation time, it is not feasible to run a full system simulation of SPEC CPU 2017. While related work mostly reduced the workload of SPEC by choosing representative code slices from the benchmark [46, 53], we simulate the SPEC Integer Speed suite based on the command line interface described in [28] using gem5's Syscall Emulation (SE) mode which provides accurate performance simulation of the benchmark but omits simulating the operating system. To further reduce the simulation time, we use the *test* workloads for SPEC which in many cases feature smaller input sizes but generally perform the same tasks as the *reference* workloads. For the Parsec benchmarks we use the input size *simmedium*.

For the simulated system we have chosen a setup with 4 GiB of RAM and a clock speed of 2 GHz. The L1 data and the instruction caches in both scenarios, i.e. the classic gem5 reference and CLEPSYDRACACHE, are modeled as 4-way set associative classic cache with a size of 64 KiB. The L2 cache, which is in our case also the LLC, has a size of 1 MiB and is 8-way set associative in both cases. We apply the new CLEPSYDRACACHE architecture to the LLC, i.e. L2, and use a Random-Replacement-Policy (RRP). For the classic caches we use LRU [1] replacement policy. We provide an overview of the cache configuration of our setup in Table 4.

Table 4: Summary of cache settings in gem5's simulation environment.

Settings	Classic	CLEPSYDRACACHE
L1 Size	64kB	64kB
L1 Architecture	classic	classic
L1 Replacement Policy	LRU	LRU
L2 Size	1MB	1MB
L2 Architecture	classic	CLEPSYDRACACHE
L2 Replacement Policy	LRU	Random Replacement

C CLEPSYDRACACHE and Real-Time-Systems

Systems can have different requirements towards expected or mandatory response times. In case a system requires strict deadlines towards response times, it is called a real-time system. There are different types of real-time systems: *hard, soft and firm real-time systems*. Those types can be differentiated by their strictness towards deadlines. A hard real-time system must meet its deadlines at all times. Otherwise, this can lead to severe consequences. A good example for hard real-time systems are control-units for airbags. In contrast, a firm real-time system can accept infrequent deadline misses. Voice transmission is such a system. A few missed bits of sound will not degrade the entire system, but frequent misses will. A real-time system which is neither hard nor firm, is called soft. In a soft real-time system the value of the information decreases over time. A good example is an air conditioning system with regular temperature measurements. The air conditioner operates on the most frequent measurements. Frequent misses will be tolerated as long as there are a few timely measurements available [44]. For a hard real-time system, it is very important to be able to predict a worst-case time to complete its deadlines.

Thus, for the compatibility of CLEPSYDRACACHE with real-time systems, the worst-case delay of CLEPSYDRACACHE needs to be considered. The worst-case delay is known and represents the case with no cached entries. We assume for the worst-case delay, that all cached entries are evicted due to an expired TTL. Thus, like for all other caches the (worst-case) influence of CLEPSYDRACACHE on the execution time is precisely known. In general there is no conceptual difference between a classic cache and CLEPSYDRACACHE as in both scenarios the worst-case scenario i.e. no cached entries needs to be considered by the real-time system. The choice of the cache influences the time required for the *CPU* to perform *load* and *store* instructions. For example, we consider a hard real-time system with strict response times $t_{response}$. The system needs to have enough computing power p to complete the task tsk before $t_{response}$ (deadline). A designer of a hard real-time system always chooses enough power p to complete tsk before $t_{response}$ under worst-case circumstances. As a consequence, there is no difference for the choice of a cache on a hard real-time system.

In case of a firm or soft real-time system an average response time t_{avg} for a deadline d can be considered. Since a few missed deadlines are allowed in firm or soft real-time systems, the system could be designed to operate on average execution times. The choice of the cache will now influence the (average) execution of *load* or *store* instructions on the *CPU*. However, different caches will lead to different average execution times for different use-cases. Thus, the design of such a real-time system, needs to determine the average execution times for the used cache c with workload w. Therefore, the only difference between a classic cache and CLEPSYDRACACHE is the average execution time under a specific workload w.