Fast and Secure LLC Caches via Spatial Windows

Roland Czerny[1]✉[0009-0000-9230-0833], Simon Lammer[1][0009-0005-8968-879X], Lukas Giner[1][0000-0002-7133-1515], Anirban Chakraborty[2][0000-0001-7411-7509], and Daniel Gruss[1][0000-0002-7977-3246]

[1] Graz University of Technology, Graz, Austria
{firstname.lastname}@tugraz.at
[2] Max Planck Institute for Security and Privacy, Bochum, Germany
anirban.chakraborty@mpi-sp.org

Abstract. We propose a generic technique, ObfusCache, to de-correlate cache activity from secret-dependent operations in secure randomized last-level caches (LLCs) within **configurable spatial windows**. We reduce the side-channel-observable spatial granularity to the order of magnitude of TLB or page cache channels. We show that the randomization window acts as a prefetcher, improving performance by up to 8 % in the Scimark2 benchmarks compared to ScatterCache, which has no prefetching for security reasons. Regarding security, we show that the lower bound for the slowdown for Prime+Prune+Probe eviction set generation is at least three orders of magnitude (680x). More importantly, the required eviction set is larger than the cache, which makes reliable eviction infeasible, e.g., we observe evictions in only 30 % of the attempts. We show that even an informed adversary with a hypothetical way to evict reliably requires linearly more traces in the window size. Our evaluation shows that ObfusCache thwarts state-of-the-art attacks with our recommended window sizes with a large security margin.

1 Introduction

Caches bridge the performance gap between fast CPUs and comparatively slower main memory. However, their limited capacity necessitates strategic data management, based on the principle of locality, which can inadvertently expose sensitive information through side-channel attacks. Prime+Probe [32, 41, 31, 21, 26, 28] is a widely used cache attack technique requiring minimal assumptions but leaking very precise information. Cryptographic secrets are high-value targets that are often processed in shared environments, e.g., cloud computing [41, 31] or browsers [31, 49, 45, 46]. Numerous works have demonstrated the feasibility of cache attacks on widely used cryptographic implementations, e.g., AES T-tables [32, 21, 26, 28, 61], RSA [65, 26, 68, 1], ECDH [48, 15] and ECDSA [34, 3, 12, 8]. Furthermore, attacks on post-quantum cryptography [20, 33], zero-knowledge proofs [30], machine learning [63, 58, 66] and blockchain-based applications [54, 42] have established the broad applicability of cache attacks on critical systems.

Mitigating cache attacks broadly follows two strategies—*constant-time code* and *building secure caches*. Constant-time code avoids secret-dependent branches, memory accesses, or variable-latency instructions, ensuring that all operations are independent of the secret. However, this approach is not always feasible, especially for applications that require user input or complex data structures. Additionally, modifying existing software to adhere to constant-time principles requires special expertise and is error-prone [13, 44, 16], leaving code vulnerable to side channels. The other strategy, *i.e.*, building secure caches, aims to mitigate cache attacks by scrambling address-cache-line mappings [59, 60, 23, 25, 24, 55, 38, 39, 62, 52, 43]. Multiple works use cryptography [38, 39, 62, 52, 43] to randomize address-to-cache-location mappings in hardware with a secret key. Thus, secure randomized caches make it challenging to induce interference between attacker and victim and, to a lesser extent, to measure the change in the cache state. However, ultimately, they do not de-correlate cache activity from the secret processed by the victim. Hence, even with randomized secure caches, eviction-based attacks are still possible as the capacity remains limited [36, 7].

In this paper, we propose ObfusCache, a generic extension to randomized secure caches that minimizes the correlation of the victim's secret with any cache activity. The design of ObfusCache stems from the observation that on a cache miss, the cache fill operation is directly correlated with the secret location accessed by the victim. ObfusCache breaks this link by retrofitting the concept of a "random fill" to randomized secure caches [25, 11]. The security of random fills has previously only been explored for smaller commodity cache designs [25]. ObfusCache handles cache hits through the underlying (unmodified) randomized secure cache. However, for cache misses, for both read and write operations, ObfusCache handles the operation as an uncacheable operation and instead caches a random location within a configurable access window. In contrast to the approach by Liu et al. [25], this also thwarts advanced attacks like Prime+Prune+Probe on secure randomized caches using write accesses for the eviction set.

We formally analyze different ObfusCache window sizes and show that, on average, leakage is reduced by a factor of $log_2(2r + 1)$. Based on a worst-case leakage analysis, we recommend a large window size of 4 kB ($r = 32$) to thwart a wide range of attacks while maintaining good performance. We validate these findings through a cache simulator implementing ObfusCache on top of ScatterCache [62]. The results confirm that achieving the same eviction probability as vanilla ScatterCache (without ObfusCache) requires $2r + 1$ times larger eviction sets, making Prime+Probe infeasible on ObfusCache with recommended settings. Furthermore, ObfusCache complicates the priming step by introducing self-evictions due to randomization. On a 16-way cache with $r = 32$ and 1 MB per slice, eviction sets become too large to fit in the cache, reducing the eviction probability to below 30 %, even in the best case for the attacker.

As ObfusCache does not change the cache structure but only the cache behavior, it does not introduce additional cache state compared to the baseline secure randomized cache. In terms of performance, ObfusCache introduces a behavior that is similar to a prefetcher. For every cache miss, ObfusCache fetches a

random cache line into the cache in spatial proximity. Therefore, if the principle of locality holds for a workload and the cache is not exhausted, this behavior may improve performance in our evaluation.

Contributions. In summary, the main contributions of this work are:

1. We propose a generic technique, ObfusCache, to de-correlate activity in secure caches from secret-dependent activity within a chosen window r.
2. We show, in worst-case and theoretical information leakage analyses, that ObfusCache reduces information leakage by a factor of $log_2(2r+1)$ on average.
3. ObfusCache exacerbates both eviction set construction and measurements in the attack by three orders of magnitude and, due to cache size constraints, makes attacks infeasible in our recommended configuration.
4. In gem5 simulation, ObfusCache with the recommended configuration reduces last-level cache misses by 7 %, and improves performance by up to 8 % in the Scimark2 benchmark suite.

2 Background

This section provides background on caches, cache attacks, secure randomized caches, and attacks on secure randomized caches.

Caches and Cache Attacks. Caches speed up accesses to recently used data by buffering it in the CPU. Their size is limited, so they only contain a small subset of the data available in main memory. *Set-associative* caches organize this data in uniquely tagged 64 B *cache lines* associated with one of many *sets*. The virtual or physical address determines which set a cache line is stored in, with each set containing n cache lines (*ways*), typically 4 to 20. A *replacement policy* determines which line is evicted from the set when new data is loaded.

Caches are usually in a hierarchy with private caches for each core (L1 and L2 caches) and a last-level cache (LLC) shared across cores. LLC sizes on modern CPUs have multiple megabytes and are split into *slices* (smaller independent caches) with addresses assigned to the slices based on a *slice function* [27].

An attacker can use a timing side channel to test whether data is in the cache. The most prominent timing techniques are Flush+Reload [64] and Prime+Probe [32]. Flush+Reload [64] flushes a (read-only shared memory) address from the cache and times the reload operation to determine whether a victim accessed it. Prime+Probe [32] constantly keeps a cache set filled (prime) and measures how long it takes to fill it (probe), using a so-called *eviction set*. There are many approaches to finding eviction sets, typically involving testing a large number of addresses for congruency in more or less sophisticated ways [26, 57].

Secure Caches. Secure caches make contention attacks (statistically) hard by making the address to set difficult to observe by an adversary and hinder the construction of eviction sets. In general, they follow two orthogonal dimensions: randomization and partitioning. While **randomization** obfuscates the address-to-set-index mapping, e.g., using lookup tables, hash functions, and block ciphers, the entire cache is still reachable for each security domain. **Partitioning** logically or physically partitions the cache, making parts unreachable for each secu-

rity domain. Randomization mappings can be static (e.g., pre-computed tables) or dynamic (e.g., PRFs). Mappings can be renewed periodically, e.g., *rekeying* or *remapping* to effectively destroy any congruence information an attacker may have learned, although it comes with a performance and power overhead [38].

Early secured cache designs used table-based randomized mappings, e.g., RPcache [59] and Newcache [60], requiring either full associativity or content addressable memory, impractical for large LLCs. Time-Secure Cache [55] uses a keyed function with cache line and process ID (PID) as input. CEASER [39] only uses the PID as input and frequent rekeying instead.

More contemporary designs employ stronger randomization, e.g., CEASER-S [39], ScatterCache [62], and PhantomCache [52], based on skewed cache designs [47] suitable for LLCs. They prevent Flush+Reload attacks by design, as they restrict shared memory across security domains. ScatterCache computes the indices of individual cache lines, CEASER-S and PhantomCache, randomly map from computed indices to entire sets. Most secure cache designs are based on set-associative architectures. MIRAGE [43], MAYA [5], and Chameleon Cache [56] try to mimic a fully associative cache by separating tag and data store and using bidirectional pointers to connect tag store and data store entries.

Attacking Secure Caches. Eviction set generation first occupies a large portion of the cache and then eliminates one address from the set at a time [26], *i.e.*, $O(n^2)$ for n LLC cache lines. Consequently, using a randomizing function for the address-to-set mapping [38, 55] eliminates this notion of set. However, improved algorithms [57, 39] discover eviction sets by eliminating groups of cache lines at once, resulting in lower runtimes ($O(n)$). Thus, impractically low rekeying periods would be required to prevent eviction set generation.

Designs like [62, 39, 52] adopt skewed associative designs, where different way-slices have independent set mappings. For a generalized eviction set, an adversary must now obtain a set of candidate addresses that are fully congruent in *each* partition. Recent works [36] generate eviction sets efficiently by observing the cache lines evicted by the victim and probabilistically eliminating non-conflicting cache lines. In particular, Purnal et al. [37, 36] propose Prime+Prune+Probe, *i.e.*, Prime+Probe with an additional *Pruning* step to occupy a large portion of the cache and eliminate non-conflicting addresses in a bottom-up manner. Song et al. [50] flush the attacker's cache lines to hinder Prime+Prune+Probe and avoid full cache evictions. They also gradually re-randomize mappings to increase resilience. However, Bourgeat et al. [7] recently showed that gradual re-randomization itself could accumulatively leak information over multiple epochs.

3 The ObfusCache Design

In this section, we overview the design of ObfusCache, the eviction and filling strategy, as well as other fundamental design decisions.

Idea. The central idea of ObfusCache is to reduce the spatial granularity of the side channel to mitigate the remaining leakage of secure random caches in contention-based attacks. ObfusCache employs a random fill architecture, where

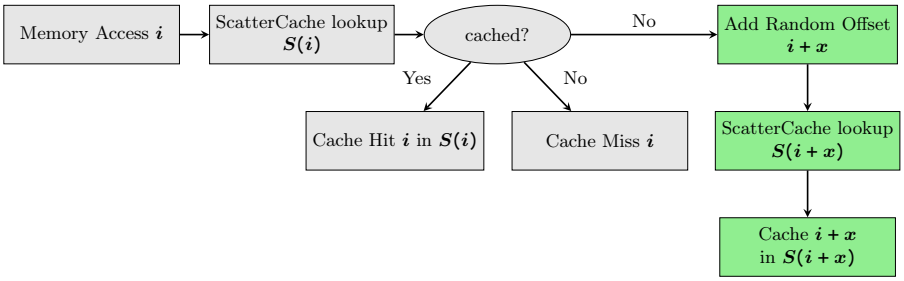


Fig. 1. ObfusCache employs a secure randomized cache but alters the behavior of cache misses: Cache misses now trigger caching of a random location within a window, like a prefetcher. The actually requested data is served from DRAM instead.

a memory location is randomly selected to be cached from a window around the loaded address. We introduce a radius parameter r that defines the window size (typically a power of 2 for simplicity). For example, for address x, we select a random value uniformly from $[x - r, x + r]$, multiplied by the cache line size, e.g., 64 B. Our design is the **first** that combines a randomized secure cache, namely ScatterCache [62], with a random fill mechanism [25]. While we chose ScatterCache, our approach could be applied to any other randomized secure cache design as well. In contrast to prior work, our random fill mechanism targets the larger LLC and additionally hides the cache location of memory writes. Hiding memory write locations, along with read locations, is crucial to thwart attackers who rely on filling specific cache lines during the priming phase.

Strategy. Figure 1 provides an overview of the different ObfusCache scenarios based on whether the accessed memory location is cached in the LLC. Upon any memory request, we retrieve the mapping of address to cache line using ScatterCache. When the data is cached, we proceed without randomization and benefit from a fast cache hit. Werner et al. [62] showed that the ScatterCache lookup to determine the cache line in the LLC can be performed in parallel to the L1 and L2 lookups with minimal performance impact.

We handle read and write cache misses differently. When **reading**, we forward requested data from memory directly to the CPU without caching it in the LLC. In parallel, we choose a second random memory location within the window around the loaded address. The system-wide random fill window size can be configured dynamically at runtime, e.g., by the trusted hypervisor or operating system. Each location in the window, including the loaded address, has an equal probability of being chosen to be cached in the LLC. This step hides the loaded address and serves as a prefetch mechanism, leveraging locality to reduce the performance impact. The random access is not on the critical path, and the CPU can continue without waiting for the random fill to complete, similar to a prefetch. When **writing** to uncached locations, we again choose a random address from the window around the loaded address to be cached in the LLC. The written value is not cached and is immediately stored in main memory, *i.e.*,

as in a write-through cache. For a write to a cached memory location, including a randomly chosen one, we update the cached value.

Architecture. We propose ObfusCache as a non-inclusive LLC ScatterCache with $P = n_w = 16$ and random replacement. As discussed by Werner et al. [62], random replacement achieves similar performance to other techniques and simplifies the overall architecture. For the random fill technique, we use an RNG seeded from a TRNG. The random values are pre-computed and buffered and have minimal influence on the architecture's performance. As the random fill acts similarly to a prefetch operation, where the CPU does not immediately use the data, the performance impact is minimal.

Our design separates the private L1 and L2 caches from the ObfusCache LLC by making the LLC non-inclusive, allowing the private caches to use traditional cache architectures. By separating ObfusCache and private caches, we gain two important advantages: First, maintaining cache coherency is simpler with non-inclusive caches. Second, and more importantly, this approach optimizes the efficiency of the private caches, reducing the overall performance impact.

Overhead. ObfusCache is stateless, *i.e.*, it has no storage overhead. The design requires an RNG, a buffer for random values, and control logic for the random fill mechanism. We did not implement the design in hardware. Therefore, we provide no quantitative hardware overhead estimates. Performance overhead arises from additional misses when requested data is not cached. In our evaluation, the prefetch-like effect of randomly caching may offset the overhead of additional misses, resulting in performance gains for some workloads (Section 6).

4 Worst-Case Leakage Bound of ObfusCache

To derive the worst-case leakage bound, we run ObfusCache on a ScatterCache (without loss of generality). The attacker can trigger victim executions and observe cache state changes by timing its own addresses. The best-case scenario for the attacker is when a target address y is cached in a set S_y, and the attacker's addresses $\mathbb{X}$, used to build eviction sets, are also cached in set S_y. Furthermore, for each address $x \in \mathbb{X}$, the random fill address o is cached in a different set ($\tilde{S_y}$). The number of addresses for a successful eviction set is n_w with $n_w \gg w$ (number of ways in a set), since the attacker does not have prior knowledge of the cache set (out of S sets in total) their addresses are mapped to.

We compute the probability of such an event X when an attacker address x is mapped to the same set S_y as the target address y; O is the event when the random fill address o is mapped to a different set $\tilde{S_y}$. The probability of the best-case scenario for the attacker is given by the joint probability $P(X, O) = P(X \cap O)$, or simplified: $P(X, O) = P(X)P(O|X)$. The joint probability signifies that the attacker address x is mapped to the same set as the victim while all the random fill addresses do not interfere with that particular set.

Recall that X represents the event where the attacker address x is mapped to the victim set S_y. Therefore, $P(X) = \frac{1}{S}$, as the victim address can be mapped to any of the S sets in the cache. To compute the probability of random fill

event O not getting mapped to the set S_y, we note that this particular event can take place in $^{S-1}/_{S}$ ways for each attacker access x. Note that for each access x, the random fill address is chosen from the range $[x-r, x+r]$, *i.e.*, $2r$ addresses besides x. Any address from the window, including x, is equally likely to be chosen. To ensure that none of the addresses in the random fill window is mapped to the cache set S_y, the event O requires each address to be mapped to $\tilde{S}_y$. Therefore, the probability of the random fill address not getting mapped to the set S_y is given by $P(O|X) = \left(^{S-1}/_{S}\right)^{2r}$. As the attacker needs n_w addresses for an eviction set, the expected number of such events can be computed as

$$\mathbb{E}[P(X,O)] = \sum_{i=0}^{n_w} P(X) \cdot P(O|X) = n_w \cdot \left(\frac{1}{S}\right) \cdot \left(1 - \frac{1}{S}\right)^{2r}. \tag{1}$$

The Equation (1) denotes the worst-case leakage of ObfusCache, where the attacker observes the cache as similar to ScatterCache. In other words, ObfusCache essentially reduces to ScatterCache (or any underlying secured cache design) in the worst case. However, the probability of such an event is extremely low (as shown in Figure 2), as the attacker needs to ensure that the random fill addresses do not interfere with the eviction set generation.

Average Case Leakage. The worst-case leakage provides a bound on the maximum information leakage of ObfusCache based on the **best possible scenario for the attacks**, which is highly unlikely in practice. Therefore, we also consider the average case leakage for a more realistic view of the cache design. The average case leakage is the expected information leakage over all possible scenarios. To model the average case leakage, we perform the Mutual Information (MI) [67] analysis, which quantifies how much uncertainty about a secret is reduced given the attacker's observations due to the side channel. In randomized caches, leakage is probabilistic, and MI naturally captures the expected information gain rather than worst-case scenarios, as observed by Genkin et al. [14].

Let $\mathbf{Y}$ and $\mathbf{O}$ be the random variables representing a normal access to the cache and a random fill address respectively. The mutual information, denoted by $\mathbf{I(Y;O)}$, provides the measure of information leakage of a normal cache access $\mathbf{Y}$ due to random fills $\mathbf{O}$. Mutual information can be represented in terms of entropies as $\mathbf{I(Y;O)} \equiv \mathbf{H(Y)} - \mathbf{H(Y|O)}$, where $\mathbf{H(Y)}$ is the initial entropy of the access and $\mathbf{H(Y|O)}$ is the system's entropy as observed after the random fill.

Any address y can be mapped to any of the locations in the cache. Considering total $N(= S \times w)$ locations in the cache, the initial entropy is $\mathbf{H(Y)} = \log_2 N$ [3]. Now, for every access y, a random address o is also loaded onto the cache, sampled uniformly from the range $[y-r, y+r]$, *i.e.*, $2r+1$ addresses. Therefore, the probability that a specific address is chosen from the window is $P(\mathbf{O}=o|\mathbf{Y}=y) = \frac{1}{2r+1}$.

With Bayes's conditional probability rule,

$$P(\mathbf{Y}=y|\mathbf{O}=o) = \frac{P(\mathbf{O}=o|\mathbf{Y}=y)P(\mathbf{Y}=y)}{P(\mathbf{O}=o)}. \tag{2}$$

[3] $\mathbf{H(Y)} = -\sum_{i=1}^{N} P(\mathbf{Y}=i) \times \log_2 P(\mathbf{Y}=i) = -N \times \frac{1}{N} \times \log_2 \frac{1}{N} = \log_2 N$

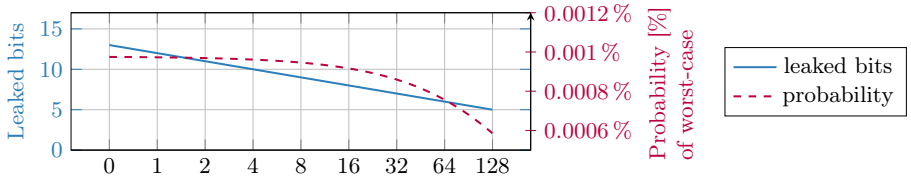


Fig. 2. Average case information leakage (in blue) and worst-case leakage probability (in purple) for different ObfusCache parameter r (x axis). Cache configuration: $N = 16384$ (1 MB cache slice), $w = 16$, $S = 1024$.

Given $\mathbf{Y}$ and $\mathbf{O}$ are uniformly distributed over the cache[4], we can assume $P(\mathbf{Y} = y) = P(\mathbf{O} = o)$. Therefore, $P(\mathbf{Y} = y|\mathbf{O} = o) = P(\mathbf{O} = o|\mathbf{Y} = y) = \frac{1}{2r+1}$. Substituting these in the definition for mutual information, we get

$$\begin{aligned}\mathbf{H}(\mathbf{Y}|\mathbf{O}) &= -\sum_{y} P(\mathbf{Y} = y|\mathbf{O} = o) \log_2 P(\mathbf{Y} = y|\mathbf{O} = o) \\ &= -(2r+1) \cdot \frac{1}{(2r+1)} \cdot \log_2 \frac{1}{(2r+1)} = \log_2 (2r+1).\end{aligned} \tag{3}$$

and with Equation (3), we get

$$\mathbf{I}(\mathbf{Y};\mathbf{O}) \equiv \mathbf{H}(\mathbf{Y}) - \mathbf{H}(\mathbf{Y}|\mathbf{O}) = \log_2 \left(\frac{N}{2r+1} \right). \tag{4}$$

This mutual information formulation (Equation (4)) directly relates the information gain of a side channel to ObfusCache's security parameter (r).

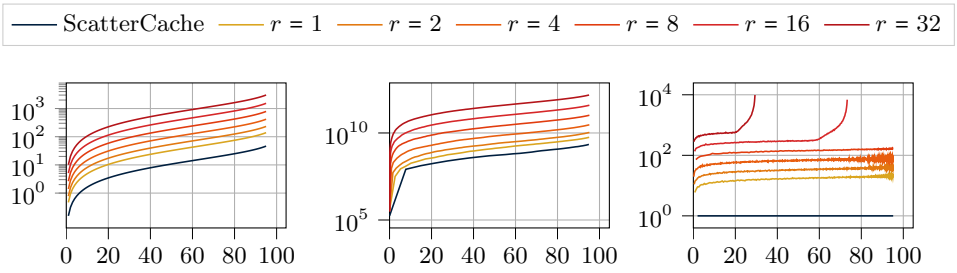
Information leakage reduces with an increase in window size (r) for a fixed cache size (N). For the underlying secured cache (ScatterCache in this case), the mutual information becomes $\log_2 N$, which is the maximum information leakage, contrasted by the much lower leakage of $\log_2 (N/2r+1)$ with ObfusCache.

> ObfusCache random fills add a noise factor $\log_2(2r+1)$ to the secure cache.

Figure 2 shows the average case leakage and the probability of the worst-case leakage for different r. The average case leakage reduces with an increase in the window size r, as expected. ScatterCache without ObfusCache leaks 14 bits of information for the given N. The leakage gradually reduces to ≈ 7 bits for ObfusCache with $r = 32$. Similarly, the probability of the worst-case leakage reduces with an increase in r.

> Worst-case leakage (best for the attacker) with $r = 32$: ObfusCache eliminates leakage in 99.9991 % of cases. It acts like ScatterCache in the remaining ones.

[4] The randomizing function provides the uniform distribution for the mapping of both normal access and random fill access.



a. Set Size G in kB. With ObfusCache, G is $2r + 1$ times larger for same p_e.

b. Profiling accesses per target. Grows with r and p_e due to larger $|G|$ required.

c. *Prime* iterations to load eviction set; $r \geq 16$ fails due to self-evictions.

Fig. 3. ObfusCache (n_{ways} = 16, 8× 1 MB slices) for different r. ScatterCache is $r = 0$. Normalized to the eviction (and victim access detection) probability p_e (x axis [%]).

5 Security Evaluation

In this section, we evaluate the security of ObfusCache in a cache simulator mounting Prime+Prune+Probe (PPP) [36] with a fully informed adversary taking windowing into account. Since victim accesses become invisible with ObfusCache, we instead observe the ObfusCache window, which requires more effort but is equivalent to observing the victim access. As a benchmark, we use an AES T-table attack, a well-known and easy to mount cache attack [32, 36].

Threat Model. ObfusCache is a generic extension to secure randomized caches under the same threat model [62, 52, 36, 53, 17]: An attacker can distinguish cache hits and misses via timing but is in a security domain isolated from the victim, on separate cores. We assume contention channels in the underlying secure cache and that the attacker can mount a noise-free attack. This intentionally strong attacker model allows reasoning about worst-case leakage. The noise-free assumption is an analytical convenience, as real systems exhibit additional noise and scheduling effects. Occupancy attacks and other non-cache side channels (e.g., port contention) are out of scope, as they require orthogonal mitigations [17].

Existing Attacks on Secure Randomized Caches. Purnal et al. [36] extended Prime+Probe with partially congruent addresses to a probabilistic PPP attack with an eviction set $G = \bigcup_{i \in w} G_i$, where each address in G_i collides with a target address x in (at least) one way i. The attack consists of three steps: *First*, loading a candidate set of random addresses k into the cache that may collide with target address x. *Second*, the candidate set is iteratively accessed, removing addresses that cause misses until only addresses with cache hits k' remain. *Third*, partial congruency is detected by triggering a victim access and observing the effect on the candidate set: If candidate address reaccess results in a miss, the address is partially congruent to the target address x and added to G. Repeating this step results in eviction set G. The probability of detecting a victim access in an attack grows with the size of G.

With ObfusCache, loading the candidate set k into the cache is harder: Each access is only cached with probability $1/2r+1$. The number of accesses n to cache an address with probability p is $n = ln(1-p)/ln(1-\frac{1}{2r+1})$. For $r = 32$ (window size 4 kB), an entry has a 50 % probability of being cached after 45 memory accesses, compared to **a single access** without ObfusCache.

For ScatterCache, an attacker must find many addresses that collide with x in at least one way. The eviction probability p_e determines the probability of detecting a victim access for a generalized eviction set of size $|G|$. The eviction probability for ScatterCache is $p_e = 1 - (1 - 1/w)^{|G|/w}$, where w is the number of ways [62]. To distinguish neighboring addresses, an attacker needs $2r + 1$ more addresses with ObfusCache to cover all addresses within the window. We use

$$|G| = w \cdot ln(1-p_e)/ln(1-\frac{1}{w}) \cdot (2r + 1)$$

for the required addresses for a given eviction probability p_e.

Figure 3a shows the size of G for specific eviction probabilities p_e for a 16-way ObfusCache with various r compared to ScatterCache only. For ObfusCache with a 4 kB window, the probe set G needed to reach 90 % eviction probability is 2.32 MB large. A probe set of this size fills a large portion of the cache, leading to self-evictions and less accurate results. An attacker can always resort to a smaller set G to reduce self-evictions, resulting in a lower eviction probability.

> ObfusCache makes Prime+Prune+Probe significantly harder: an attacker needs a set G that is $2r+1$ times larger than in ScatterCache to reach the same p_e, and randomization prevents simply caching candidate addresses.

Windowed Prime+Prune+Probe. With ObfusCache, eviction sets grow with the window size, quickly exceeding the cache size and making parallel monitoring of multiple addresses infeasible. Our *windowed* PPP lets an informed adversary monitor each full window and, instead of building separate sets per victim address, reuse overlaps to construct a single eviction superset covering all targets. This reuse reduces cache footprint and RAM usage and shows up to which parameters an adversary can still locate window borders and center.

Windowed PPP samples k random cache lines, prunes k to cache-hits-only k', and detects congruency via victim execution and miss detection in k'. In the third step, the victim must access **all** target locations x_j, an unrealistic but intentionally strong assumption for the security argument. The attacker adds partially congruent addresses to a superset S until it is large enough. The size of S balances profiling and attack effort. If all targets x_j are adjacent, S grows only by the partial-congruent addresses of one target, as neighbors share windows. In our simulation, we use $2.5 \cdot n_{ways}$ partial-congruent addresses per target.

Next, the attacker divides S into smaller sets S_j, where each set corresponds to a specific victim address x_j. For this, the attacker loads the superset into the cache and triggers a victim execution that does not access all victim addresses. With the information about the accessed target addresses and multiple iterations to account for the ObfusCache randomization, the attacker can assign addresses of the superset to sets belonging to specific victim addresses.

After building and separating S, the attacker loads it into the cache, runs the victim, and records evictions. As each element of S corresponds to a set of partial-congruent victim addresses, the attacker can infer which cache lines the victim accessed. Only addresses outside overlaps uniquely identify the access, which occurs with probability $1/2r+1$. On average, the attacker needs $E[X] = \frac{1}{1/2r+1} = 2r + 1$ additional victim executions to distinguish neighboring addresses, and even more for multiple neighbors because middle ones lack unique borders within their windows. We evaluate this in an AES T-table attack case study.

> Even when using partially congruent addresses across overlapping windows for an eviction superset, distinguishing neighboring addresses requires at least $2r+1$ more victim runs; more for addresses without uniquely identifying borders.

Evaluation: Profiling Phase Overhead. We compare profiling effort for ScatterCache with and without ObfusCache. We assume a 16-way 8 MB cache with 8 slices, random replacement, and varying ObfusCache window sizes. Profiling aims to find a set G congruent with victim address x. Since ObfusCache requires $2r + 1$ times more addresses to reach the same eviction probability p_e, we normalize for p_e. We use $3N$ full evictions between PPP iterations to ensure x is evicted. Figure 3b experimentally compares profiling effort for one target x across ObfusCache configurations and ScatterCache. ObfusCache requires orders of magnitude more accesses than ScatterCache. For example, at $r = 32$ about $680\times$ more are needed to reach $p_e = 50\,\%$.This mainly results from the larger set size required for the same p_e. Moreover, finding partial-congruent addresses requires caching k' before each victim access, which is harder with ObfusCache.

> Profiling needs orders of magnitude more accesses (e.g., $680\times$ for $r = 32$ and $p_e = 50\%$), and large windows cause self-evictions that make priming unreliable or even impossible for $r = 32$ and $p_e > 30\,\%$.

Evaluation: Prime Phase Overhead. We evaluate the overhead when priming the eviction set for different windows; again with a 16-way cache (8 MB, 8 slices) and random replacement. To detect a victim access, an attacker must load the eviction set G into the cache, *i.e.*, *prime* it. Loading specific addresses into the cache requires many accesses for ObfusCache. With addresses at least one page apart, due to L1 and L2 indices, the attacker does not benefit from ObfusCache's prefetching but instead suffers from self-evictions.

Figure 3c shows the required prime iterations for ObfusCache and ScatterCache, normalized for p_e. Priming fails entirely for large ObfusCache windows when aiming for $p_e > 30\,\%$ due to self-evictions, while ScatterCache typically succeeds in a single iteration. If the attacker can rerun the victim arbitrarily often, they can trade a smaller eviction set G for more executions, which is beneficial for large windows where high-p_e sets no longer fit reliably in cache.

Table 1. AES T-table attack ($n = 4$) on a one-level ScatterCache with/without ObfusCache ($r \in \{1, 2, 4\}$). $n_{\text{slice}} = 8$ (1 MB each). At $r = 1$ the attack is 289× slower than ScatterCache only. For larger r, it recovers only a few key nibbles unreliably after days. With $n_{\text{ways}} = 16$, ObfusCache further impedes the attack vs. $n_{\text{ways}} = 8$.

n_{ways}	r	#AES [10^3]	Accesses [10^9]	estimated time	effective p_e	correct nibbles
8	0	19.93±2.50	35.41±3.30	10.73±0.83 minutes	22.91%	16.00 ± 0.00
8	1	685.47±48.02	7388.33±10.88	2.16±0.00 days	12.61%	16.00 ± 0.00
8	2	33 374.04±13 248.71	20 088.16±3233.22	4.96±0.57 days	4.51%	4.75 ± 0.50
8	4	28 250.14±12 724.20	30 273.60±3676.95	7.93±0.65 days	3.64%	0.25 ± 0.50
16	0	103.65±57.35	160.76±75.99	45.16±19.22 minutes	6.19%	16.00 ± 0.00
16	1	20 336.81±9456.17	24 152.97±2475.07	6.56±0.43 days	4.02%	9.00 ± 1.63
16	2	19 126.99±6589.26	37 479.47±1996.43	10.43±0.35 days	1.49%	2.25 ± 0.50
16	4	15 627.89±7019.43	67 389.69±2795.71	19.16±0.49 days	1.85%	1.00 ± 0.00

AES T-table Attack Benchmark Case Study. As a benchmark, we mount the AES T-table attack from prior work [36, 32], recovering the 16 upper key nibbles. The informed adversary uses windowed PPP to distinguish neighboring accesses in the T-table AES implementation of OpenSSL 1.1.0g, a well-known side-channel benchmark [4, 32, 51, 18, 36]. We run the attack in a single-level cache simulator, favoring the attacker since every access reaches the targeted cache. We assume the attacker can monitor every random access without noise.[5]

We first profile the implementation to find addresses partially congruent to the victim's cache lines, *i.e.*, 64 cache lines for all four T-tables. We generate $256 \cdot n_{target}$ key-plaintext pairs that access all but one target line and use probabilistic full evictions (accessing $3N$ random addresses) between encryptions. This yields a histogram of how often each superset address was evicted per key-plaintext pair.[6] Thus, an attacker can still determine addresses corresponding to the excluded table accesses, but requires significantly more profiling iterations for similar confidence. For $r = 1$, we performed 600× more separation iterations than ScatterCache, yet achieved only $p_e = 12.61\,\%$ compared to $22.91\,\%$ for ScatterCache with $n_{ways} = 8$. The profiling phase yields a superset $S = \bigcup_{i \in n_{\text{target}}} S_i$, where each S_i contains addresses that collide with target i in at least one way.

For the exploitation phase, the attacker repeatedly loads the superset S into the cache (*i.e.*, the attacker *primes* the cache), triggers an encryption, and measures the access times for all addresses in S. After sufficiently many encryptions, the attacker uses the information about which addresses in S collide with which target addresses to recover 64 bits in the 16 upper nibbles of the 16-byte key.

We evaluate the attack on 8- and 16-way caches with $n_{\text{slice}} = 8$ (1 MB each) and random replacement, assuming a 4 GHz CPU with 1-cycle hits and 200-cycle misses. The implementation is not fully optimized, e.g., we use a smaller superset to simplify profiling and priming at the cost of more attack iterations. Results

[5] Unmapping memory around the T-tables hinders the attack. If random windows are truncated at T-table boundaries, accesses become indistinguishable for $r \geq 64$, and even for smaller r distinguishing neighboring addresses is harder.

[6] We use 256 key-plaintext pairs per target address to reduce access-frequency bias.

are shown in Table 1. The effective p_e is the post-profiling eviction probability, determined by comparing separation outputs with actual target collisions.

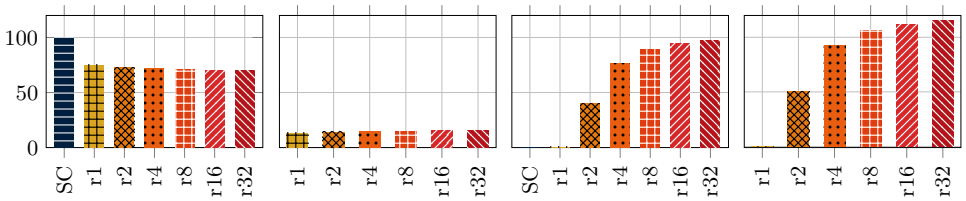
We could only successfully mount the attack for $r = 1$, which we consider insecure. For $n_{ways} = 8$, this required over 30 $\times$ more encryptions than ScatterCache only to recover the upper 16 nibbles, and overall took 289 $\times$ longer, increasing the runtime from minutes to days. For $r = 2$ ($n_{ways} = 8$), we recovered only 4 nibbles after several days with over 1600 $\times$ more AES encryptions, and for larger r the attack was even more unreliable and slower ($n_{ways} = 16$). We conclude that **our recommended r = 32 makes the attack infeasible**. Moreover, we assume attacker-favorable yet unrealistic conditions: loading data into a non-inclusive LLC is difficult, and suppressing shared-LLC noise is hard.

6 Performance Evaluation

We evaluate an ObfusCache-augmented ScatterCache against baseline caches in two simulation setups: gem5 [6] with benchmarks from prior work [17, 38, 39, 62], and a custom cache simulator for the SPEC CPU 2017 benchmarks [9]. All results stem from simulation and should be interpreted as indicative trends rather than absolute predictions for real hardware. Our gem5 setup does not model an end-to-end modern core and uncore, and we provide no HDL/FPGA implementation. Thus, effects from detailed microarchitectural mechanisms, such as complex out-of-order execution, and full memory-system interactions, are not captured in full detail. Overall, the prefetching effect of ObfusCache may improve performance for most benchmarks compared to ScatterCache only. In ObfusCache, prefetching emerges as a side effect of the security mechanism, whereas in other designs prefetchers typically weaken security. At the same time, randomized fills may increase cache pollution and memory traffic. As a result, ObfusCache may degrade performance for workloads with little spatial locality or under high cache pressure. For completeness, we also evaluate ObfusCache against other cache designs *with* prefetching. However, prefetchers amplify side-channel leakage [48, 10] and their effect on the security of these prior designs has not been studied, making any direct comparison of performance and security difficult.

Performance Evaluation in gem5. We implemented ObfusCache in gem5 [6] and first evaluate best- and worst-case microbenchmarks. Then, similar to prior work [17, 38, 39, 62], we use gem5 [6] to run the MiBench [19], LMbench [29], scimark2 [35], and GAP [2] benchmarks. We evaluate the performance of ObfusCache with different window sizes and compare it to a baseline cache design with `rand` and `LRU` replacement, SassCache, and ScatterCache.

gem5 Test Setup. We evaluate in gem5 [6] using 32-bit ARM mode, full-system simulation, and a `TimingSimpleCPU` at 2 GHz. The `TimingSimpleCPU` is an in-order core model and does not reflect the behavior of modern out-of-order processors, so we use gem5 to study relative trends under a controlled setup. L1 and L2 caches are identical across configurations to isolate L3 effects: split 32 kB 8-way LRU L1 data/instruction caches and a 256 kB 4-way LRU L2. The L3 is 1 MB with 16-ways. Modern CPUs use cache slices instead of a single shared



a. Best case L3 miss rates [%] (σ = 0.03).
b. Best case perf. gain [%] (σ = 0.03).
c. Worst case L3 miss rates [%] (σ = 0.02).
d. Worst case perf. loss [%] (σ = 0.43).

Fig. 4. *Best-case* (random accesses within ObfusCache window) and *worst-case* (stride=4 kB exceeding window) microbenchmarks for ObfusCache vs. ScatterCache in gem5 system emulation. 16 ways, 8 slices (1 MB each), 5000 buffer accesses, $n = 5$.

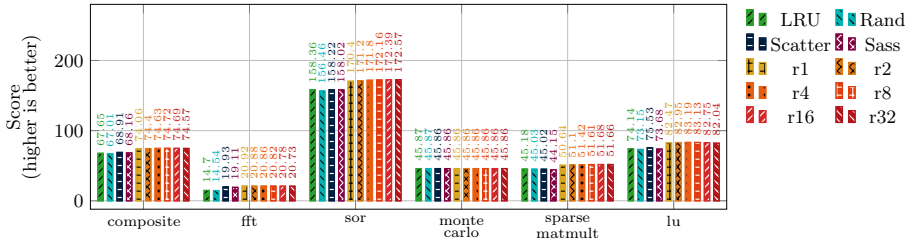


Fig. 5. Scimark2 (large) scores simulated with gem5.

L3. With all cores active, a single core effectively accesses a portion of the total cache, which is similar in size to a single slice. This makes our chosen sizes realistic for evaluation [17]. We test 5 cache designs: (1) LRU replacement, (2) random replacement, (3) ScatterCache, (4) SassCache, and (5) ObfusCache with various window sizes. As in prior work [62], we run Poky Linux 19.0.2 (Yocto 2.5 "sumo") with Linux 4.14.67, patched for gem5, and collect cache statistics.

gem5 Results. Cache performance is workload-dependent, and memory pressure from other cores or tasks may alter these results. We evaluate best- and worst-case scenarios in two microbenchmarks: The best case issues random accesses within the ObfusCache window, where ObfusCache reduces cache misses by over 20 % and improves performance by about 15 % compared to ScatterCache (Figures 4a and 4b). The worst case uses offsets exceeding the window range, making ObfusCache's prefetching ineffective due to lacking spatial locality (Figures 4c and 4d). For small r, the miss rate remains similar to ScatterCache since the originally accessed addresses are more likely to be cached after a few accesses. For larger r, the miss rate reaches 100 %, roughly doubling runtime.

Figure 5 shows the Scimark2 (large) score in gem5. ObfusCache outperforms ScatterCache, SassCache, and the baseline caches in most tests. Interestingly, a larger ObfusCache window improves performance in some tests. The composite score is more than 8 % higher compared to ScatterCache without ObfusCache for

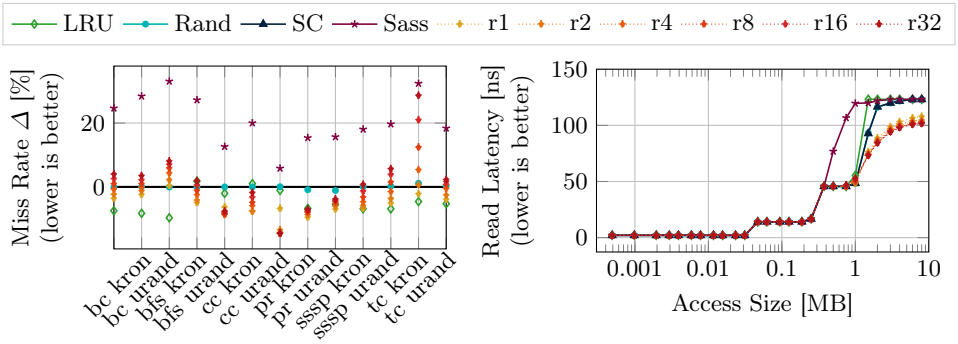
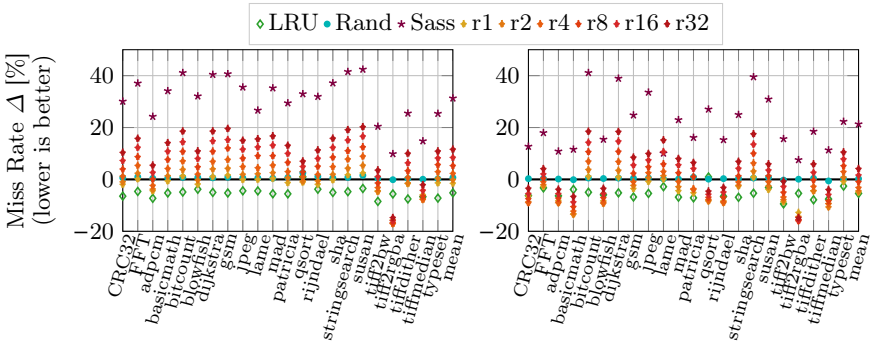


Fig. 6. Miss-rate Δ vs. ScatterCache on GAP benchmarks (gem5). ObfusCache's prefetching lowers miss rates in most cases.

Fig. 7. Memory read latency, simulated with gem5, with 64 B strides (*i.e.*, one access per cache line).

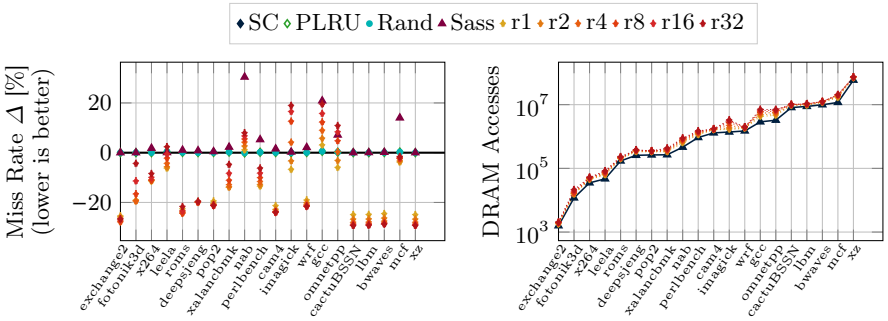


a. MiBench small configuration.

b. MiBench large configuration.

Fig. 8. L3 miss rate for MiBench benchmarks in gem5. %pt. Δ vs. ScatterCache.

our recommended window size of 4 kB ($r = 32$). Next, Figure 6 shows the L3 miss rate for the GAP benchmark suite in gem5. We evaluated all algorithms (*i.e.*, `bc, bfs, cc, pr, sssp, tc`) using both synthetically generated `kron` and `urand` trace sets. The results show the prefetching effect of ObfusCache, resulting in a lower cache miss rate than ScatterCache only in most GAP benchmarks. For this benchmark suite, smaller ObfusCache windows outperform larger ones. Figure 7 shows the memory read latency in the `lat_mem_rd` benchmark with 8 MB and 64 B strides, equivalent to one access per cache line. Again, ObfusCache's prefetching effect leads to better results compared to the other caches. Figure 8 shows the L3 miss rate for the MiBench benchmark suite simulated with gem5. For the small configuration, ObfusCache windows $r > 4$ lead to a higher cache miss rate compared to ScatterCache only. For the larger configurations, ObfusCache improves the miss rate for most tests of the MiBench suite.



a. Cache miss rate percentage points delta to ScatterCache without ObfusCache.

b. More DRAM accesses with ObfusCache due to random fill accesses.

Fig. 9. SPEC CPU 2017 benchmarks in custom cache simulator.

Performance Evaluation in Custom Cache Simulator. Like prior work [62, 17], we implemented ObfusCache in a custom cache simulator for performance evaluation with the SPEC CPU 2017 benchmark [9]. We evaluate an ObfusCache-augmented ScatterCache against baseline cache designs with `rand` and `PLRU` replacement, ScatterCache, and SassCache. Same as in gem5, we use three cache levels: split 32 kB 8-way LRU L1 caches (data/instruction), a 256 kB 4-way LRU L2, and a 1 MB 16-way L3. For the benchmarks, we collect traces from the SPEC CPU 2017 benchmark suite [9] using Intel PIN Tool [22], which is faster than gem5 but still requires trace size limits. Like prior work [40, 39, 62, 17], we use representative 250-million-instruction traces per benchmark.

Cachesim Results. Figure 9a shows cache miss rate differences (percentage points) for various configurations, relative to ScatterCache. The results indicate that the performance impact of ObfusCache strongly depends on the specific benchmark workload. Some benchmarks benefit from larger windows, others from smaller ones. In addition to the benchmarks evaluated in the ScatterCache paper [62], we also evaluate `gcc`, `wrf`, and `cam4`, which exhibit mixed behavior: while `wrf` and `cam4` benefit from ObfusCache, `gcc` shows higher miss rates compared to ScatterCache. On average across all evaluated benchmarks, ObfusCache improves the performance of ScatterCache by 10.87 pp ($r = 32$) to 15.20 pp ($r = 1$). Also here, additional memory pressure from other cores or tasks may alter these results. Figure 9b shows that ObfusCache leads to more DRAM accesses due to the random fill accesses.

Performance Comparison against Prefetch-Enabled Cache Designs. In ObfusCache, prefetching arises from the security mechanism. For completeness, we also evaluate the other designs with prefetchers enabled, but only compare performance: prefetchers amplify side-channel leakage [48, 10], and prior work has not studied their security impact in these designs. As the custom cache simulator does not support prefetching, we evaluate this only in gem5, using a stride prefetcher for the baseline caches, SassCache, and ScatterCache.

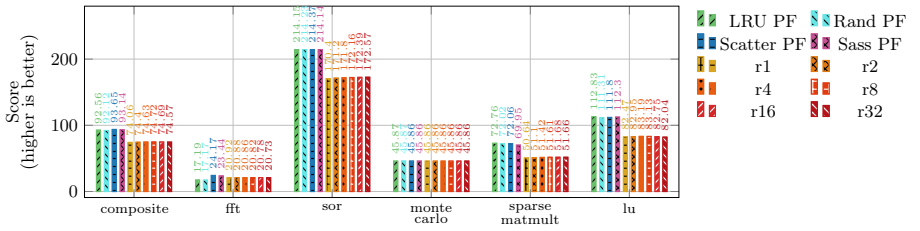


Fig. 10. Scimark2 (large) scores simulated with gem5. The results indicate that ObfusCache's prefetching is generally weaker than a dedicated prefetcher. However, for some benchmarks, it comes close or even outperforms the dedicated prefetcher.

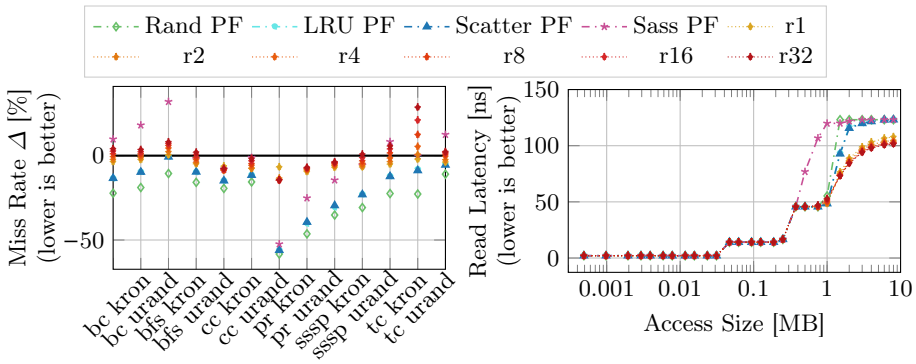
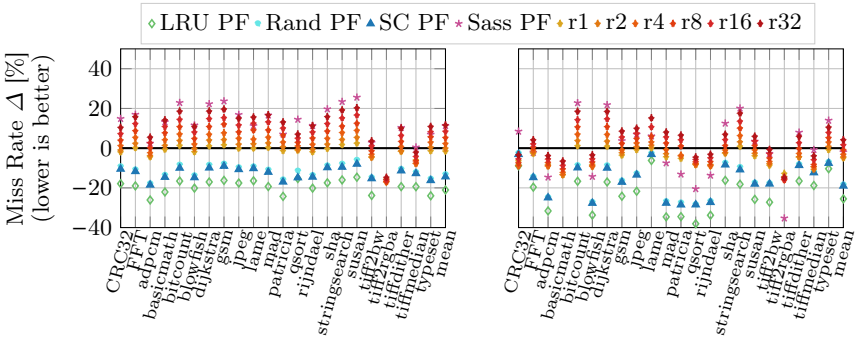


Fig. 11. Miss-rate Δ vs. ScatterCache on GAP benchmarks (gem5). ObfusCache's with gem5, with 64 B strides (*i.e.*, one ac-prefetching comes close to the dedicated cess per cache line).
prefetcher in some benchmarks.

Fig. 12. Memory read latency, simulated

Overall, ObfusCache's prefetching is weaker than a dedicated prefetcher, but comes close or even outperforms it in some benchmarks. Figure 10 shows that the dedicated prefetcher generally outperforms ObfusCache in Scimark2, although ObfusCache comes close or even outperforms it in some benchmarks. Here, larger windows perform better than smaller ones. Figure 11 shows the L3 miss rate for the GAP benchmark suite in gem5. Also here, ObfusCache's prefetching cannot match a dedicated prefetcher in most benchmarks. However, in some benchmarks, ObfusCache comes close to the dedicated prefetcher. For this benchmark suite, smaller windows outperform larger ones. Figure 12 shows the memory read latency evaluated using the `lat_mem_rd` benchmark. Interestingly, the results indicate that ObfusCache's prefetching may outperform the dedicated prefetcher for this benchmark. Figure 13 shows the L3 miss rate difference compared to ScatterCache with a prefetcher for the MiBench benchmark suite. For this suite, ObfusCache's prefetching is generally weaker than a dedicated prefetcher.



a. MiBench small configuration.

b. MiBench large configuration.

Fig. 13. Cache miss rate for MiBench benchmarks simulated with gem5. Percentage points difference to ScatterCache.

7 Discussion and Conclusion

ObfusCache reduces the information leakage from victim activity to cache activity, which can also be expressed as the entropy of the learned result: Flush+Reload and Prime+Probe leak whether a 64 B block was accessed whereas page cache and TLB leak whether a 4 kB block was accessed. As ObfusCache only leaks whether a 4 kB block was accessed (with a 4 kB window) it effectively reduces the spatial leakage to the same order as page cache and TLB side channels.

ObfusCache is a generic technique to de-correlate cache activity within configurable spatial windows from secret-dependent operations. We formally analyzed and evaluated the security of ObfusCache, showing attack slowdowns, e.g., 680x, requiring days to build an eviction set. We showed that resulting eviction sets exceed the cache size and that attacks fail, including on trivial targets like AES T-tables. The 8 % performance gain in gem5 on average shows that ObfusCache is a secure alternative to prefetching for secure cache designs. As these results are obtained in simulation, they may not fully translate to real-world systems. In particular, the prefetch-like effect may not be beneficial for all workloads due to cache pollution. Overall, ObfusCache augments secure caches by reducing cache side-channel leakage and may improve performance through a prefetch-like effect.

Acknowledgments This research is supported in part by the European Research Council (ERC project FSSec 101076409), and the Austrian Science Fund (FWF project NeRAM 10.55776/I6054 and FWF SFB project SPyCoDe 10.55776/F85). Additional funding was provided by generous gifts from Google and Intel. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding parties.

References

1. Aldaya, A.C., García, C.P., Tapia, L.M.A., Brumley, B.B.: Cache-timing attacks on RSA key generation. In: CHES (2019)
2. Beamer, S., Asanovic, K., Patterson, D.A.: The GAP Benchmark Suite. arXiv:1508.03619 (2015)
3. Benger, N., van de Pol, J., Smart, N.P., Yarom, Y.: Ooh Aah... Just a Little Bit: A small amount of side channel can go a long way. In: CHES (2014)
4. Bernstein, D.J.: Cache-Timing Attacks on AES (2005), `http://cr.yp.to/antiforgery/cachetiming-20050414.pdf`
5. Bhatla, A., Panda, B., et al.: The Maya Cache: A Storage-efficient and Secure Fully-associative Last-level Cache. In: ISCA (2024)
6. Binkert, N., Beckmann, B., Black, G., Reinhardt, S.K., Saidi, A., Basu, A., Hestness, J., Hower, D.R., Krishna, T., Sardashti, S., et al.: The gem5 simulator. ACM SIGARCH computer architecture news (2011)
7. Bourgeat, T., Drean, J., Yang, Y., Tsai, L., Emer, J., Yan, M.: CaSA: End-to-end Quantitative Security Analysis of Randomly Mapped Caches. In: MICRO (2020)
8. Brumley, B., Hakala, R.: Cache-Timing Template Attacks. In: AsiaCrypt (2009)
9. Bucek, J., Lange, K.D., v. Kistowski, J.: SPEC CPU2017: Next-Generation Compute Benchmark. In: International Conference on Performance Engineering (2018)
10. Chen, Y., Pei, L., Carlson, T.E.: AfterImage: Leaking control flow data and tracking load operations via the hardware prefetcher. In: ASPLOS (2023)
11. Deng, S., Xiong, W., Szefer, J.: Secure TLBs. In: ISCA (2019)
12. Fan, S., Wang, W., Cheng, Q.: Attacking OpenSSL implementation of ECDSA with a few signatures. In: CCS (2016)
13. Geimer, A., Vergnolle, M., Recoules, F., Daniel, L.A., Bardin, S., Maurice, C.: A Systematic Evaluation of Automated Tools for Side-Channel Vulnerabilities Detection in Cryptographic Libraries. In: CCS (2023)
14. Genkin, D., Kosasih, W., Liu, F., Trikalinou, A., Unterluggauer, T., Yarom, Y.: CacheFX: A Framework for Evaluating Cache Security. In: AsiaCCS (2023)
15. Genkin, D., Valenta, L., Yarom, Y.: May the Fourth Be With You: A Microarchitectural Side Channel Attack on Several Real-World Applications of Curve25519. In: CCS (2017)
16. Gerlach, L., Pietsch, R., Schwarz, M.: Do Compilers Break Constant-time Guarantees? In: FC (2025)
17. Giner, L., Steinegger, S., Purnal, A., Eichlseder, M., Unterluggauer, T., Mangard, S., Gruss, D.: Scatter and Split Securely: Defeating Cache Contention and Occupancy Attacks. In: USENIX Security (2023)
18. Gruss, D., Spreitzer, R., Mangard, S.: Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In: USENIX Security (2015)
19. Guthaus, M.R., Ringenberg, J., Ernst, D., Austin, T., Mudge, T., Brown, R.B.: MiBench: A free, commercially representative embedded benchmark suite. In: WWC (2001)
20. Huang, S., Sim, R.Q., Chuengsatiansup, C., Guo, Q., Johansson, T.: Cache-timing attack against HQC. Cryptology ePrint Archive, Report 2023/102 (2023)
21. Hund, R., Willems, C., Holz, T.: Practical Timing Side Channel Attacks against Kernel Space ASLR. In: S&P (2013)
22. Intel: Pin - A Dynamic Binary Instrumentation Tool (2012), `https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool`

23. Kong, J., Acıiçmez, O., Seifert, J.P., Zhou, H.: Hardware-software integrated approaches to defend against software cache-based side channel attacks. In: HPCA (2009)
24. Liu, F., Ge, Q., Yarom, Y., Mckeen, F., Rozas, C., Heiser, G., Lee, R.B.: CATalyst: Defeating Last-Level Cache Side Channel Attacks in Cloud Computing. In: HPCA (2016)
25. Liu, F., Lee, R.B.: Random Fill Cache Architecture. In: MICRO (2014)
26. Liu, F., Yarom, Y., Ge, Q., Heiser, G., Lee, R.B.: Last-Level Cache Side-Channel Attacks are Practical. In: S&P (2015)
27. Maurice, C., Le Scouarnec, N., Neumann, C., Heen, O., Francillon, A.: Reverse Engineering Intel Complex Addressing Using Performance Counters. In: RAID (2015)
28. Maurice, C., Weber, M., Schwarz, M., Giner, L., Gruss, D., Alberto Boano, C., Mangard, S., Römer, K.: Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud. In: NDSS (2017)
29. McVoy, L., Staelin, C.: Lmbench: Portable Tools for Performance Analysis. In: USENIX ATC (1996)
30. Mukherjee, S., Rechberger, C., Schofnegger, M.: Cache Timing Leakages in Zero-Knowledge Protocols. Cryptology ePrint Archive, Report 2024/1390 (2024)
31. Oren, Y., Kemerlis, V.P., Sethumadhavan, S., Keromytis, A.D.: The Spy in the Sandbox: Practical Cache Attacks in JavaScript and their Implications. In: CCS (2015)
32. Osvik, D.A., Shamir, A., Tromer, E.: Cache Attacks and Countermeasures: the Case of AES. In: CT-RSA (2006)
33. Pessl, P., Groot Bruinderink, L., Yarom, Y.: To BLISS-B or not to be: Attacking strongSwan's Implementation of Post-Quantum Signatures. In: CCS (2017)
34. van de Pol, J., Smart, N.P., Yarom, Y.: Just a little bit more. In: CT-RSA 2015 (2015)
35. Pozo, R., Miller, B.R.: Scimark 2.0 (2004), `https://math.nist.gov/scimark2/`
36. Purnal, A., Giner, L., Gruss, D., Verbauwhede, I.: Systematic Analysis of Randomization-based Protected Cache Architectures. In: S&P (2021)
37. Purnal, A., Verbauwhede, I.: Advanced profiling for probabilistic Prime+Probe attacks and covert channels in ScatterCache. arXiv:1908.03383 (2019)
38. Qureshi, M.K.: CEASER: Mitigating Conflict-Based Cache Attacks via Encrypted-Address and Remapping. In: MICRO (2018)
39. Qureshi, M.K.: New attacks and defense for encrypted-address cache. In: ISCA (2019)
40. Qureshi, M.K., Jaleel, A., Patt, Y.N., Steely, S.C., Emer, J.: Adaptive insertion policies for high performance caching. ACM SIGARCH Computer Architecture News **35**(2), 381 (2007)
41. Ristenpart, T., Tromer, E., Shacham, H., Savage, S.: Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-Party Compute Clouds. In: CCS (2009)
42. Roy, S., Morais, F.J.A., Salimitari, M., Chatterjee, M.: Cache Attacks on Blockchain Based Information Centric Networks: An Experimental Evaluation. In: ICDCN (2019)
43. Saileshwar, G., Qureshi, M.K.: MIRAGE: Mitigating Conflict-Based Cache Attacks with a Practical Fully-Associative Design. In: USENIX Security (2021)
44. Schneider, M., Lain, D., Puddu, I., Dutly, N., Capkun, S.: Breaking Bad: How Compilers Break Constant-Time Implementations. In: AsiaCCS (2025)

45. Schwarz, M., Maurice, C., Gruss, D., Mangard, S.: Fantastic Timers and Where to Find Them: High-Resolution Microarchitectural Attacks in JavaScript. In: FC (2017)
46. Schwarzl, M., Schuster, T., Schwarz, M., Gruss, D.: Speculative Dereferencing of Registers: Reviving Foreshadow. In: FC (2021)
47. Seznec, A.: A case for two-way skewed-associative caches. ACM Computer Architecture News (1993)
48. Shin, Y., Kim, H.C., Kwon, D., Jeong, J.H., Hur, J.: Unveiling Hardware-based Data Prefetcher, a Hidden Source of Information Leakage. In: CCS (2018)
49. Shusterman, A., Agarwal, A., O'Connell, S., Genkin, D., Oren, Y., Yarom, Y.: Prime+Probe 1, JavaScript 0: Overcoming browser-based side-channel defenses. In: USENIX Security (2021)
50. Song, W., Li, B., Xue, Z., Li, Z., Wang, W., Liu, P.: Randomized Last-Level Caches Are Still Vulnerable to Cache Side-Channel Attacks! But We Can Fix It. In: S&P (2021)
51. Spreitzer, R., Plos, T.: Cache-Access Pattern Attack on Disaligned AES T-Tables. In: COSADE (2013)
52. Tan, Q., Zeng, Z., Bu, K., Ren, K.: PhantomCache: Obfuscating Cache Conflicts with Localized Randomization. In: NDSS (2020)
53. Thoma, J.P., Niesler, C., Funke, D., Leander, G., Mayr, P., Pohl, N., Davi, L., Güneysu, T.: ClepsydraCache – Preventing Cache Attacks with Time-Based Evictions. In: USENIX Security (2023)
54. Tramèr, F., Boneh, D., Paterson, K.: Remote Side-Channel attacks on anonymous transactions. In: USENIX Security (2020)
55. Trilla, D., Hernandez, C., Abella, J., Cazorla, F.J.: Cache Side-channel Attacks and Time-predictability in High-performance Critical Real-time Systems. In: DAC (2018)
56. Unterluggauer, T., Harris, A., Constable, S., Liu, F., Rozas, C.: Chameleon Cache: Approximating Fully Associative Caches with Random Replacement to Prevent Contention-Based Cache Attacks. In: IEEE SEED (2022)
57. Vila, P., Köpf, B., Morales, J.: Theory and Practice of Finding Eviction Sets. In: S&P (2019)
58. Wang, H., Hafiz, S.M., Patwari, K., Chuah, C.N., Shafiq, Z., Homayoun, H.: Stealthy inference attack on DNN via cache-based side-channel attacks. In: DATE (2022)
59. Wang, Z., Lee, R.B.: New cache designs for thwarting software cache-based side channel attacks. ACM SIGARCH Computer Architecture News **35**(2), 494 (2007)
60. Wang, Z., Lee, R.B.: A Novel Cache Architecture with Enhanced Performance and Security. In: MICRO (2008)
61. Weiß, M., Heinz, B., Stumpf, F.: A Cache Timing Attack on AES in Virtualization Environments. In: FC (2012)
62. Werner, M., Unterluggauer, T., Giner, L., Schwarz, M., Gruss, D., Mangard, S.: ScatterCache: Thwarting Cache Attacks via Cache Set Randomization. In: USENIX Security (2019)
63. Yan, M., Fletcher, C.W., Torrellas, J.: Cache Telepathy: Leveraging Shared Resource Attacks to Learn DNN Architectures. In: USENIX Security (2020)
64. Yarom, Y., Falkner, K.: Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. In: USENIX Security (2014)
65. Yarom, Y., Genkin, D., Heninger, N.: CacheBleed: A Timing Attack on OpenSSL Constant Time RSA. JCEN (2017)

66. Yuan, Y., Pang, Q., Wang, S.: Automated side channel analysis of media software with manifold learning. In: USENIX Security (2022)
67. Zhang, T., Lee, R.B.: New models of cache architectures characterizing information leakage from cache side channels. In: ACSAC (2014)
68. Zhang, Y., Juels, A., Reiter, M.K., Ristenpart, T.: Cross-VM Side Channels and Their Use to Extract Private Keys. In: CCS (2012)