Revisiting Prime+Prune+Probe: Pitfalls and Remedies

Moritz Peters, Florian Stolz, Jan Philipp Thoma, Tim Güneysu, and Yuval Yarom
Ruhr University Bochum

***Abstract*—Randomizing the mapping of memory addresses to cache locations is a promising approach for protecting computer systems against cache attacks. Multiple randomized caches have been proposed recently, with the aim of preventing adversaries from creating eviction sets – collections of addresses that compete with target memory addresses on cache space. However, Purnal et al. (IEEE SP 2021) demonstrated the PRIME+PRUNE+PROBE attack, which allows attackers to efficiently build generalized eviction sets, which evict the target memory address with a high probability. As the complexity of constructing eviction set is a key factor in randomized cache design, the PRIME+PRUNE+PROBE attack significantly reduces the security bounds of these randomizing designs.**

Since the PRIME+PRUNE+PROBE attack is probabilistic, generalized eviction sets often *get stuck* after repeated use, making them ineffective for typical cache attack settings. Prior works have noticed this behavior and proposed mitigation approaches, based on evicting members of the eviction set from the cache, using either probabilistically, by random memory accesses, or directly, using dedicated flush instructions. However, these techniques are not accompanied by an analysis of their effectiveness or any evaluation of their success.

In this work we revisit the analysis of the PRIME+PRUNE+PROBE in light of the possibility of eviction sets getting stuck. We first observe that flushing does not behave as anticipated in realistic cache architectures where invalid cache lines are filled first before evicting other lines. We also propose a new technique for allowing repeated attacks – combining random noise with flushing. We conduct an in-depth analysis of all discussed techniques and compare their complexity attacking an AES T-table implementation. We find that combining probabilistic eviction with flushing outperforms the traditional approaches by a factor of two, allowing attackers to run with higher granularity, being able to observe victim processes even better than before.

***Index Terms*—Cache Side Channels, Cache Randomization, Prime+Prune+Probe**

1. Introduction

The microarchitecture of modern CPUs is riddled with side channels that can be exploited to leak sensitive information, such as secret encryption keys, across process boundaries [22, 39, 40]. Side channels introduced by memory access latency variations due to hardware caches have been extensively analyzed in recent years [9, 12, 14, 20, 44, 50, 53]. These caches are essential components of almost every computer microarchitecture, bridging the performance gap between the CPU and the memory. Caches shorten and even remove pipeline stalls by storing the contents of recently accessed memory regions close to the CPU, to allow faster access when that memory region is used again. Due to the extent of the performance difference, attackers can reliably determine whether a given memory access was served from the cache or from memory. Moreover, cache side-channel attacks are often used as a building block in other attacks. For example, transient execution attacks, such as Spectre [16] and Meltdown [18], leverage cache side channels to leak secrets through microarchitectural state.

Generally, there are two types of cache side-channel attacks. Flush-based attacks like FLUSH+RELOAD [50] and FLUSH+FLUSH [12] leverage a cache maintenance instruction like `clflush` on x86 to flush a shared memory address from the cache. Based on timing information, the attacker can then determine whether a victim process has accessed the flushed memory address and thus recover a secret being processed by the victim. Since flush-based attacks rely on cache maintenance instructions, which are rarely used in benign workloads, they are easy to detect and can be prevented by defining the cache flush instruction as privileged on the ISA level.

Contention-based attacks, on the other hand, exploit the physical hardware architecture of modern caches and are, thus, much more challenging to mitigate. Here, the attacker must construct an *eviction set*, i.e., a collection of addresses guaranteed to replace the victim's address from the cache. This is feasible because the set-associative structure of most caches allows a given address to map to only a small number of cache lines. The simple addressing function allows efficient construction of such eviction sets [20, 37, 42, 45]. The PRIME+PROBE attack [20, 25, 44] uses an eviction set to replace a victim address from the cache. Then, the victim is triggered to perform a secret-dependent operation, which, depending on the secret, reloads the victim address to the cache and replaces one of the eviction set addresses. The attacker then probes the eviction set addresses, measuring the access latency to detect cache misses and learn whether the victim has accessed the address. There are many variants of contention-based cache attacks, including PRIME+SCOPE [29], PRIME+ABORT [9], and EVICT+TIME [25].

Cache-index randomization schemes attempt to mitigate contention-based attacks by randomizing the mapping of addresses to cache lines [21, 41, 46, 47], hampering construction of eviction sets. In response, Purnal et al. [28] gen-

eralize some of the designs and propose PRIME+PRUNE+PROBE, a generic attack on randomized cache architectures. While cache randomization still provides a significant security improvement over traditional caches, the authors show that attackers can still cause targeted evictions when using a *generalized eviction set*, i.e., a set of addresses that is likely to replace the victim's address. In response, a variety of advanced randomized cache architectures that protect against PRIME+PRUNE+PROBE attacks have been proposed [11, 34, 43]. While they indeed hinder such attacks they increase the performance and area overheads or rely not-standard hardware.

In this paper, we first reintroduce a critical property of cache attacks, where eviction sets *get stuck* in the cache after using them repeatedly, rendering them unusable for any further use. Since many practical cache side-channel attacks rely on many iterations to recover, e.g., a secret key, this problem makes PRIME+PRUNE+PROBE impractical in these scenarios. While this issue has already been observed [28, 38, 47], it was not discussed in much detail. Therefore, we investigate why and when generalized eviction sets get stuck in randomized caches and how an attacker can efficiently resume the attack when in this state. To this end, we first perform a theoretical analysis of the problem and verify our findings using CacheFX [10], a framework for evaluating the security of randomized caches. Following this we discuss already known techniques to alleviate the mentioned issue in more detail. These are namely using multiple eviction sets [47], probabilistic eviction via random accesses [10, 28] and flushing the eviction set after each probe step [38]. The latter, i.e., flushing the eviction set, appears the fastest and cleanest option, however, we identify a fundamental issue with this technique stemming from a gap between simulated and real-world caches. Specifically, we note that real-world caches prioritize invalid cache lines over replacing valid lines when serving cache fills. Ultimately, this property leads to the flush technique inoperative. To fix this, we propose a new strategy, which combines flushing the eviction set followed by accesses to random memory location. This combination effectively re-enables the use of cache line flushes. We then evaluate these methods, again using CacheFX, to measure their effectiveness in allowing attacks to progress through multiple iterations. Specifically, we are interested in the probability with which the attack can successfully detect victim accesses. To set the evaluating into more context, we run an attack against an AES T-table implementation with the discussed techniques. Our analysis shows that our proposed technique combining flushing and random accesses serves effectively re-enables the utilization of flushes while outperforming random accesses.

To summarize, the contributions of this work are:

- We revisit PRIME+PRUNE+PROBE and analyze a known but until now sidelined issue with generalized eviction sets and victim addresses getting stuck in the randomized caches during PRIME+PRUNE+PROBE (Section 3).
- We perform a theoretical analysis of this problem (Section 3.1) and discuss its root cause (Section 3.2).
- Using CacheFX [10], we perform an experimental evaluation and quantify the security of different randomized caches against PRIME+PRUNE+PROBE (Section 3.3).
- We discuss different techniques an attacker can employ to prevent their eviction set from getting stuck in randomized caches and find that flushing the eviction does not function as intended in real-world cache architectures (Section 4).
- We propose combining random noise with flushing as a strategy for overcoming eviction sets getting stuck.
- We analyze mitigation techniques to evaluate their effectiveness of enabling an attacker to observe victim accesses and compare their complexity attacking an AES T-table implementation (Section 5).

2. Background

In this section, we introduce the relevant background to caches, cache attacks, eviction sets, and randomized cache architectures.

2.1. Caches

Caches are moderately small, SRAM-based memory modules located close to the execution units. Since most workloads exhibit spatial and temporal locality, an access to a memory address is likely to be followed by an access to a close by address soon after. Caches exploit this locality to bridge the performance gap between the main memory and the processing unit. By storing recently accessed data, they reduce the latency of future accesses to the stored data. Modern CPUs usually feature three levels of cache. The L1 cache is the smallest and fastest, while the L3 cache (also known as Last Level Cache, or LLC) is much larger but also slower. The L1 and L2 caches are usually instantiated for each core, while the LLC is shared among all cores.

To allow identifying the memory addresses stored in the cache, each cache entry contains a *tag* that identifies the address, in addition to the data. In fully associative caches, a cache entry can be placed in any location within the cache. However, due to the hardware and power costs associated with such designs, most modern caches are *set-associative*. A set-associative caches, the cache is divided into s sets, each containing w cache entries. The cache associativity is the number of cache entries, or ways, in each set, i.e., a cache with w ways is often called w-ways associative. To locate or to store data in set-associative caches, the processor maps each memory address to a cache set, using a predefined mapping function. A common choice of function is to take a sequence of i index bits of the address, where $2^i = s$. However, many caches use more complex functions [14, 51]. Address bits that are used as index do not need top be included in the tag, because they are implied by the cache set. When storing a new cache entry into the cache, the replacement policy selects one of the w entries within the set in which the data will be stored. Any data previously stored in the selected entry is written back to memory, if needed, and removed from the cache.

2.2. Cache Attacks

Since the design goal of caches is to reduce the latency of memory accesses, the fact that cache hits are measurably faster than cache misses is actually well intended. However, many attacks have demonstrated that this timing difference can be exploited to leak secret information like private keys from co-located processes.

There are two types of cache attacks: Flush-based attacks like FLUSH+RELOAD [50] and FLUSH+FLUSH [12] leverage a cache maintenance instruction to flush an address shared between the attacker and the victim from the cache. Then, the attacker observes whether the victim reloads the address based on timing. FLUSH+RELOAD has, for example, been used to recover secret keys across virtual machines [50], while FLUSH+FLUSH has been used as a key logger and to recover secret keys from an AES T-table implementation. The requirement that the attacker and the victim have shared memory and the unprivileged access to the cache maintenance instruction limits the applicability of flush-based attacks. By defining the cache line flush instruction as privileged on the Instruction Set Architecture (ISA) level, flush-based attacks can be prevented easily.

Contention-based attacks, on the other hand, are much harder to mitigate since they exploit the set-associative structure of the cache. For the PRIME+PROBE attack [25, 44], the attacker must first find addresses that map to the same cache set as the victim address. In particular, they need to find w addresses that collide with the victim address. This set of addresses is called a *minimal eviction set*. By accessing the eviction set addresses, the attacker can force an eviction of any other addresses stored in the targeted cache set, including the victim address. Thus, the attacker first primes the cache set for the attack by accessing the eviction set addresses. Then, the victim process is triggered to perform an operation that, depending on the secret, accesses an address that maps to the target set or not. By re-accessing the eviction set addresses and measuring the latency, the attacker learns whether one of the eviction set addresses has been replaced or not, thus leaking the secret. There are many other variants of contention-based attacks, including PRIME+SCOPE [29], PRIME+ABORT [9] and EVICT+TIME [25]. All contention-based attacks have in common that they rely on the attacker's ability to construct eviction sets. Several algorithms have been proposed for the efficient construction of eviction sets [15, 20, 37, 42, 45, 49, 53].

2.3. Randomized Cache Architectures

Several techniques have been explored to mitigate contention-based side-channel attacks, including attack detection [2, 5, 7, 24, 52], cache partitioning [13, 26, 32, 35, 36, 48], and cache randomization [8, 11, 21, 30, 31, 34, 41, 43, 47]. While the security of attack detection schemes based on Hardware Performance Counters (HPCs) has recently been questioned [6, 17, 54], mitigation techniques based on partitioning are not well scalable to the requirements of general-purpose CPUs. On the other hand, randomization appears to be a promising solution, with multiple independent works finding a significant improvement in the security against contention-based cache attacks. Generally, randomized cache architectures use a randomization function, e.g., SCARF [4], to derive a randomized cache set from the physical memory address. Thus, attackers can no longer trivially construct eviction sets by choosing addresses that match in the bit range used for the cache-set selection. Most randomized cache architectures use multiple partitions to prevent attackers from profiling addresses that collide in a given cache set. Often, the number of partitions is chosen equal to the associativity of the cache. Then, a physical address maps to a different cache set of size 1 in each cache way. Given a w-way set-associative randomized cache, the attacker would have to find w addresses that map to the exact same candidate entries as the target address in each cache way to obtain a traditional eviction set that is guaranteed to replace the target address. Due to the large number of possible combinations, this is infeasible [47].

We now briefly describe the cache architectures considered in this paper.

CEASER [31] is a randomized cache architecture with only one partition, i.e., it uses traditional cache sets. The randomization function derives the set index from the physical address, and the replacement policy selects the cache line in which the data is placed.

CEASER-S [30] introduces partitions to CEASER to increase the security against profiling attacks. In particular, CEASER-S proposes using two partitions divided by the cache ways and using different keys for the randomization. However, it is possible to use CEASER-S with more than two partitions. In this paper, we denote CEASER-S with x partitions as CEASER-S x.

PhantomCache [41] is based on a set-associative cache where x salt values are combined with a hash function to compute x set indices for a physical address. If the requested address is not cached, one of the x sets is chosen randomly. The way in which the data is stored is then determined by the replacement policy. When writing PhantomCache x, we mean PhantomCache with x random sets per address.

ScatterCache [47] is concurrent work with CEASER-S [30] and takes a similar approach. However, ScatterCache defines the number of partitions $P = w$, where w is the cache's associativity

While randomized cache architectures are secure against traditional PRIME+PROBE attacks that rely on eviction sets, Purnal et al. [28] found that attackers can still observe cache accesses by using so-called *generalized eviction sets*. A generalized eviction set contains addresses x_i that collide with the target address V in at least one cache way: $G = \{x_i \mid \exists j \in \{1, ..., w\} : f_j(v) = f_j(x_i)\}$. For a large $|G|$, the generalized eviction set has a high probability of evicting V and can, thus, be used for PRIME+PROBE-style attacks. The PRIME+PRUNE+PROBE attack [28] can be used to construct such generalized eviction sets *and* to make valuable observations during the attack using the generalized

```
Input: Victim V, pruning set P, eviction set size s
Output: Eviction Set G
while |G| < s do
    // Prime and Prune
    do
        for addr in P do
            access(addr)
        end
    while one access misses the cache
    // Trigger victim access
    access(V)
    // Probe
    for addr in P do
        is_miss = access(addr)
        if is_miss then
            G = G ∪ {addr}
        end
    end
end
```

Algorithm 1: Algorithmic overview of the PRIME+PRUNE+PROBE attack.

eviction set. To construct G, the attacker first primes the cache with a large set of random addresses P. Then, they re-access these addresses until no further self-evictions occur. This step is called pruning and is essential since, otherwise, the initial set of random addresses may have partially evicted itself due to internal collisions, which would lead to false observations during the probe phase. The pseudorandom mapping allows the entries from the initial priming set to rattle into place by accessing them multiple times. Finally, the attacker triggers an access to the victim address V. If this access misses in the cache, the CPU will replace an entry in the cache with the victim entry. If the victim address replaces one of the addresses from the priming set, the attacker can observe this access by probing the priming set. When a cache miss occurs, the attacker learns that the replaced address from P collides with the victim address in one of the cache ways. Thus, the address is added to G, and the process is repeated until G is large enough. The full algorithm can be found in Algorithm 1.

In the attack phase, the attacker first accesses the addresses from G, similar to the PRIME+PROBE attack. If G is large, the attacker must prune the eviction set by re-accessing it until no more self-evictions occur. Then, the attacker triggers the victim program, which, depending on a secret, accesses V. By probing the addresses from G, the attacker learns with some probability (depending on the size of G) whether the victim accessed the address or not.

2.4. Previous work on PRIME+PRUNE+PROBE

Since the publication of the PRIME+PRUNE+PROBE attack, various works have noted deficiencies that degrade its performance. Purnal et al. [28] mentioned a *penalty* when the victim is already cached, which requires additional actions by the adversary. They suggest either accessing random addresses to evict the victim or accepting a degraded success probability. While they do provide formulas for calculating the penalty an attacker faces when the victim is cached, they do not further elaborate on the number of random accesses required to evict the victim probabilistically. During the evaluation of different cache designs, Genkin et al. [10] also found that when no victim access is detected during an attack round, both the eviction set and the victim are cached and thus remain stuck. In line with Purnal et al. [28], they state that an additional step is required to dislodge the victim and the eviction set if they both fit into the cache. Here again, accessing random addresses or, more generally, probabilistic eviction was proposed as a solution but not further evaluated with respect to how many addresses are required or the impact on attacks. Song et al. [38] discovered that the case of the victim and eviction set being in the cache must happen after some iterations in both CEASER-S and ScatterCache, if the eviction set is not fully congruent. Contrary to previous works, they suggest that using cache flush instructions to evict the attacker's eviction set is a much cleaner approach than using probabilistic eviction. With the attacker's eviction set not being cached anymore, it can be used again to (probably) evict the victim address and detect its reintroduction in the cache. However, the authors seem to have overlooked the effect of invalid cache lines prioritization when the processor performs cache fills.

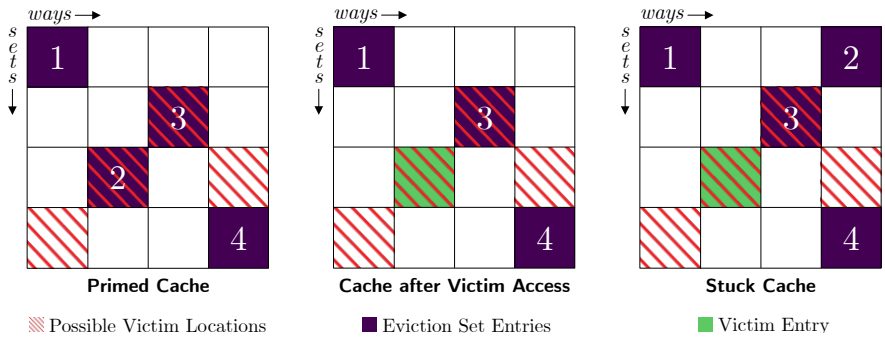
In summary, the issue of eviction sets getting stuck in the cache has been noted in past works and some approaches for addressing it have been proposed. However, a systematic evaluation of these approaches and their interaction with cache behavior is still missing.

3. Revisiting PRIME+PRUNE+PROBE

In this section we analyze a previously identified, but sidelined, problem [10, 28, 47]. Specifically, mounting a conventional PRIME+PRUNE+PROBE style attack poses significant additional challenges for the attacker due to an inherent cache property, which we define as *stickiness*. In the following we first explain the emerging problem and then evaluate the designs analyzed by Purnal et al. [28], namely CEASER(-S) [30, 31] and ScatterCache [47].

3.1. PRIME+PRUNE+PROBE in Practice

To showcase practical issues with PRIME+PRUNE+PROBE, we use ScatterCache, which is based on the calculation of a unique index for each cache way based on the requested address and a security domain identifier. However, our results can also be applied to other cache architectures, which are vulnerable to PRIME+PRUNE+PROBE, albeit with different probability functions. We perform our analysis on a cache consisting of four sets and four ways. We assume the attacker has already established a generalized eviction set G. The attacker's goal is to observe an access to the victim address V. We introduce some reasonable limitations to simplify the theoretical analysis. At the end of this section, we discuss the consequences of lifting these limitations. First, we assume that only the adversary and the victim



(a) An eviction set occupies some of the possible victim locations.

(b) A victim access is caught by the eviction set, replacing one of its entries.

(c) The cache fails to replace the victim with the removed eviction set entry.

Figure 1: Visualisation of the problem occurring when performing PRIME+PRUNE+PROBE.

access the cache. Second, we assume that no self-evictions occur when the adversary accesses the finalized eviction set. This makes the prune step obsolete. Third, we assume that the cache uses a random replacement policy, as initially proposed by Werner et al. [47]. Lastly, all accesses are required to take place within one rekeying period.

In order to perform PRIME+PRUNE+PROBE, the adversary starts by accessing the generalized eviction set G. As G is not fully congruent with the victim, it has some catching probability p_c, i.e., the probability p_c defines the ability of the attacker to observe an access to the victim address. Generally, a larger G increases the catching probability. For this example, we assume that G has $p_c = 0.9$, meaning whenever the victim performs a cache access we have a 90% probability of being able to observe it using G. This initial step of loading G into the cache is shown in Figure 1a, where entries 1–4 are occupied by G. Since G is not fully congruent, not all potential victim entries are occupied by addresses from G. Afterwards, the adversary triggers a victim access, which can result in two different outcomes: (1) the adversary catches the access or (2) the adversary does not catch the access. In case (1), the entry of G has been replaced, as seen in Figure 1b, where green denotes the victim entry. During the `probe` step, the replaced entry will be re-accessed, leading to two more possibilities: (3) the replacement policy selects the victim as the replacement candidate, or (4) the policy selects an entry that is not the victim as the replacement candidate. Case (3) represents the ideal case for the attacker, as it mimics the behavior of a traditional non-randomized cache. The victim is removed from the cache, thus reverting the state back to the beginning (Figure 1a), and the adversary can continue the attack. As we defined the replacement policy to be random, there will be w possible locations for the entry of G, hence the probability of ending up in case (3) equals $\frac{1}{w}$. However, case (4), shown in Figure 1c, causes a problem for the adversary as now all of G, as well as the victim, are present in the cache. Consequently, further victim accesses will result in a hit, thus never replacing any entry of G. Moreover, any access to addresses from G will result in cache hits and, thus, will not replace any data on access. We refer to this state as being *stuck* because the eviction set *and* the victim entry are cached, and the attacker has no opportunities to remove them using G. Similarly, case (2) will immediately result in being stuck as the state of G was never modified in the first place. Therefore, the overall probability of getting stuck in our example is $0.1 + 0.9 \cdot 0.75 = 0.775$ or more generalized:

$$p_{sc,stuck} = (1 - p_c) + p_c \cdot (1 - \frac{1}{w})$$

In the case of ScatterCache, $p_{sc,stuck}$ scales with the number of cache ways.

We now discuss how lifting our limitations will impact our theoretical results. If we allow other processes to access the cache, stray accesses may evict the victim, enabling the adversary to continue the attack. However, the success probability heavily depends on the activity of the other processes. When self-evictions are allowed, the previously evicted entry of G may be placed at a location currently occupied by G. Similarly to stray accesses to the cache, this may lead to cascading evictions during the `prime` step, thus giving the adversary a slight chance of removing the victim from the cache. Lastly, suppose the cache uses a least-recently-used replacement policy. In that case, the problem worsens as the victim will most likely be the most recently used entry, thus forcing the replacement policy not to select the victim for the entry of G.

In summary, two main properties of PRIME+PRUNE+PROBE and its eviction set can be identified as causing the problem of the attack getting stuck.

(a) The pruning behavior of PRIME+PRUNE+PROBE assumes to have the entire eviction set in the cache at once.

(b) The resulting eviction set is only partially congruent with the victim.

Property (a) causes all attacker-available addresses that might conflict with the victim to be cached and cannot be used to evict the victim again. Even if a conflict is detected – the victim has replaced an address in the eviction set – the attacker has only a single address at their disposal to evict

the victim again with only a certain probability. Property (b) has two aspects to it. First, since the produced eviction set is only partially congruent to the victim, no conflict might be detected, resulting in both the eviction set and the victim being cached, with no way for an attacker to efficiently get the victim out of the cache again. Second, as already mentioned for the first property, even if an address is evicted, due to the partial congruence, the evicted address might not evict the victim if reaccessed by an attacker.

3.2. The Root Cause

We introduced that PRIME+PRUNE+PROBE does not work as expected because an adversary gets *stuck* after just a few iterations. We refer to the underlying property which causes this behavior as *stickiness*. A *sticky* cache is designed to retain cache entries within the cache. The attentive reader may be led to the conclusion that this is the objective of all caches. However, when one considers randomized caches, a different picture emerges.

While standard caches try to keep cache entries in the cache as long as possible, they do not implement any precautions in case an entity tries to force evictions. Randomized caches, on the other hand, implement such precautions and try to keep entries in the cache even in the presence of forced evictions. We define stickiness as the success of a cache in keeping certain entries cached in the presence of forced evictions.

In order to gain further insight, we consider the role of a potential attacker. The objective of randomized cache architectures is to protect against cache side-channel attacks, such as PRIME+PROBE [25, 44]. In a cache that is vulnerable to such attacks, the attacker can construct an eviction set and access it to evict a victim entry from the corresponding cache set. This approach is effective in the majority of instances, as it results in the eviction set filling the entire cache set, assuming the use of least recently used (LRU) replacement. We define this as non-sticky behavior, in that, an attacker can evict victim cache entries from the cache with an eviction set in a reliable and repeatable manner.

Considering randomized caches, this is no longer the case. Building upon the work of Purnal et al. [28], it can be seen that the reliability of eviction sets in randomized, skewed caches such as CEASER-S [30] and ScatterCache [47] is reduced. As a result, it becomes more difficult for the attacker to evict victim entries. We would therefore describe such a cache as more *sticky*, since it is harder for the attacker to precisely evict entries.

3.3. Result Validation

In this section, we use CacheFX [10] to verify that the problem of getting stuck does occur in different cache architectures and to find how long it takes for the attacker to reach a stuck state. CacheFX is a flexible framework for evaluating different cache designs regarding their security.

The original paper that proposes the PRIME+PRUNE+PROBE attack [29] presents a successful attack on an AES T-Table implementation. The authors used memory trace recordings to simulate the attack rather than using a known cache simulator or a system-level simulator. Thus, the results of our theoretical analysis deviate from the results of the attack analysis. We suspect that the discrepancy results from the approach taken for the attack evaluation. We stress that the attack, in principle, remains valid. However, we show that the effectiveness of generalized eviction sets is lower than originally assumed.

The theoretical analysis shows that there is some probability for the attack to get stuck after each iteration. Using CacheFX, we evaluate CEASER, CEASER-S, ScatterCache, and PhantomCache with a fixed size of 4096 cache lines and random replacement. The victim is a single-access victim that only accesses one specific address per attack iteration. As also required in real-world cache attacks, we assume that the attacker can trigger the victim's access. We implemented a new attacker, the single access attacker, that performs a standard PRIME+PRUNE+PROBE attack without any additional actions, such as flushing the eviction set or accessing random addresses after each iteration. We then measure the number of successful victim access detections by the attacker over 1000 attack iterations for each combination of cache design and associativity.

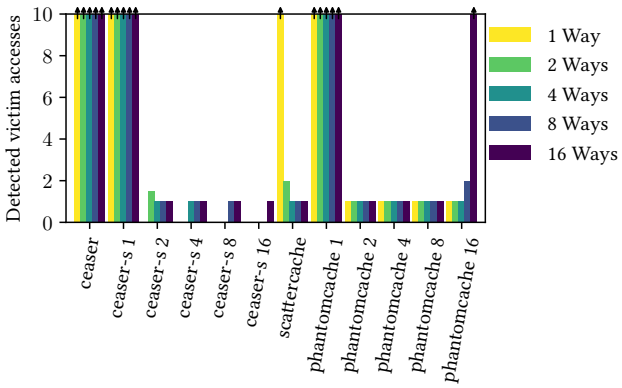


Figure 2: The number of successfully detected victim accesses for different caches and associativities for 1000 attack iterations. Results are averaged over 100 runs.

Figure 2 shows the results of the experiments. The y-axis is cut off at 10 victim accesses to keep our focus on lower values. For CEASER, CEASER-S with one partition, ScatterCache with one way, and PhantomCache 1, all victim accesses are successfully caught. This is to be expected since CEASER, CEASER-S 1, and PhantomCache 1 only select one random set for each address, which results in the behavior of a non-randomized set-associative cache. In these cases, the victim and all the eviction set addresses are exclusively mapped to the same cache set. Even with a random replacement policy, the probability of evicting the victim in either the prime or probe step is very high. In contrast, ScatterCache, with only one way, is essentially a direct-mapped cache, meaning that the eviction set contains

only a single address, which is guaranteed to evict the victim. The final outlier is PhantomCache 16 with 16 ways. Unlike the previous cases, the reason that in this configuration almost all victim accesses are successfully caught is no similarity with set-associative caches. Instead, in this configuration, the size of the eviction sets grows beyond the cache size. Hence, in each iteration, there are enough addresses to either evict the victim or cause self-evictions, which, in turn, can evict the victim or cause more self-evictions. Coming full circle, the reason behind the large eviction set is that PhantomCache 16 with 16 ways almost behaves the same as a fully associative cache.

The more interesting cases, however, are those where almost no victim accesses are caught. This directly indicates that the attackers get stuck after only a few iterations, where the eviction set and the victim are all cached. There are, at most, two detections for such test cases before the attackers get stuck. Those two detections directly correlate to the iteration in which the attack gets stuck.

In conclusion, we see that for almost all caches the attack gets stuck after one or two iterations. This also confirms, at least for ScatterCache, our calculated value of being stuck with a probability of 0.775. Next, we present approaches to restore the functionality of the attack.

4. Remedies

As previously shown, getting stuck presents a realistic and serious challenge to the adversary. In this section we discuss techniques used to solve this issue and let the adversary continue the attack. First, we lay out techniques previously used in literature [10, 28, 38, 47] and discuss their benefits and limitations. We identify that flushing on its own does not function as intended in realistic cache scenarios and propose a new technique to alleviate the problem. Generally, in order to solve the issue, the attacker must either cause a replacement of the victim address or any address from G, which can then be used to replace the victim with some probability.

Rotating Eviction Sets. One option is to use multiple eviction sets in a rotating, round-robin fashion, somewhat similar to what was proposed in Werner et al. [47]. It can be applied either alone or in combination with either of the above remedies. In the preparation for the attack, the adversary generates multiple eviction sets for the same victim address. Subsequently, a distinct eviction set is used in each attack iteration. The idea is that the other eviction sets can evict addresses from a previously used eviction set or the victim itself, as long as it is not stuck. There is, however, no guarantee that the attack will succeed without any eviction set becoming stuck. In particular, if there are only a few eviction sets, they may all get stuck in the cache along with the victim. This could occur as soon as each of the eviction sets has been used once. Using a sufficiently large number of eviction sets, such that the total number of addresses in all the eviction sets exceeds the cache's capacity, ensures that some addresses in the eviction sets will always be evicted. Those uncached addresses could potentially evict the victim when priming the corresponding eviction set again. Although this may seem like a vast number of addresses, it is important to recall that a single eviction set is small. Only the addresses belonging to one eviction set are accessed during one iteration of the attack, ensuring that the number of addresses accessed per iteration is kept low. The disadvantage, however, lies in the profiling phase before this, where multiple eviction sets need to be constructed instead of a single one.

Purge. Another way an adversary may try to resolve being stuck is to introduce a purge step after each attack iteration, enabling repeated use of the same eviction set [10, 28]. As noted in Section 3.1, stray accesses of other processes may evict the victim. However, the adversary can perform these accesses independently by generating a set of random addresses and accessing them. Therefore, adding a `purge` step after each `probe` iteration will achieve the required eviction with some probability. This step must be performed every time since the cache state is unknown, and the adversary does not know whether they are stuck or not. While it would be preferable for the adversary to completely overwrite the current cache state by accessing as many addresses as possible, this approach would take a long time, delaying the next iteration of the attack. Thus, they have to find a balance between the amount of purging and attack granularity. In general, the adversary should strive for a high granularity to catch all victim accesses, limiting the time for the purge step. However, as we show, even a relatively low number of addresses can lead to a successful eviction of the victim. For our analysis, we first determine the worst-case scenario for the adversary: the purge step neither hits the victim nor G, therefore not changing the relevant cache state. The probability p_{wc} can be modeled via the binomial distribution, as each access represents an independent event with success (e.g., hitting the victim or G) or failure outcome. Thus, for ScatterCache:

$$\begin{aligned} p_{sc,wc} &= \binom{n}{n} \cdot \left(\frac{l-|G|-1}{l}\right)^n \cdot \left(1-\frac{l-|G|-1}{l}\right)^{n-n} \\ &= \left(\frac{l-|G|-1}{l}\right)^n, \end{aligned}$$

where n denotes the number of attempts and l is the number of cache lines. If, for example, 400 random addresses are used for purging, the purge step will fail for a ScatterCache configuration with 4096 lines and four ways and an eviction set of size 34, with a probability of 3.23%. Purging hitting the victim at least once represents the best-case scenario, and its probability can be computed via

$$p_{sc,bc} = \sum_{k=1}^{n} \binom{n}{k} \cdot \left(\frac{1}{l}\right)^k \cdot \left(1-\frac{1}{l}\right)^{n-k}.$$

In our example, the attacker will succeed with 9.29%. Lastly, we consider the case in which purge accesses hit G, which

happens with a probability of

$$p_{sc,ec} = \sum_{k=1}^{n} \binom{n}{k} \cdot \left(\frac{|G|}{l}\right)^{k} \cdot \left(1 - \frac{|G|}{l}\right)^{n-k},$$

which equals 96.43% in our example. Removing some addresses of G basically grants the adversary another try to evict the victim during the `prime` stage, where G is loaded. The removed addresses act, in essence, like a smaller eviction set G_s, which is a subset of G. For our configuration, the average size (i.e., the expected value) of $G_s = 3.32$. Under the assumption that they share a mapping with the victim, the probability of success can be calculated using the formula stated by Purnal et al. [28]:

$$p_e = 1 - \left(1 - \frac{1}{w}\right)^{\frac{|G_s|}{P}},$$

where P denotes the partitions equal to the number of ways for ScatterCache. Therefore, in our example, the adversary has a 19.40% chance to remove the victim during the `prime` step. If we account for self-evictions, the probability may increase. Thus, the total probability of evicting the victim becomes approximately $0.09 + 0.96 \cdot 0.19 = 0.27$. We, therefore, conclude that an additional `purge` step effectively enables the adversary to continue the attack after only a few iterations. We note that this analysis focuses on ScatterCache; therefore, the probabilities differ for other cache architectures.

Flushing The Eviction Set. Song et al. [38] proposed that flushing the eviction set G after each probe would be much cleaner and faster than using random accesses to the LLC. They argue that cache flush instructions such as `clflush` will be available for the attacker in user-space in the foreseeable future. Even if only privileged software could access those instructions, attacks from malicious kernels or hypervisors might still be possible.

Similar to other recent research [10, 28], Song et al. [38] assume that replacement policies in caches do not differentiate between valid and invalid cache lines. However, filling an invalid line, if one exists, offers performance benefits over evicting a valid cache line, because it reduces the number of conflict misses and avoids the overhead required for handling the coherency state and writebacks. Consequently, most caches prioritize filling invalid lines. We verified that invalid cache lines are filled first on Intel i9-7900X, i7-12700KF, i9-13900T, i7-Ultra 265K and Xeon E-2224G processors. By filling a cache set and flushing a line, then accessing another address mapping to the same set, we observe that invalid lines are filled first. Abel and Reineke [1] observed the same behavior when reverse-engineering the replacement policy of contemporary processors.

In the context of the approach of Song et al. [38], whenever G is flushed, the cache lines of all of the address in G are marked invalid. When G is immediately accessed in the next prime step, there is a high probability that one or more of the possible placements of each address is an invalid cache line. These are now filled with priority over replacing existing cache lines. Subsequently, the victim address will most likely not be evicted by priming the eviction set again, leading to the same state as prior to flushing G. While the authors seem to have modeled the cache correctly by prioritizing invalid cache lines, they have not tried to use the same eviction set repeatably. This ultimately lead to the issue remaining unseen.

Flush and Purge. Building on Song et al. [38], we propose to combine flushing and purge accesses, to partially recover the benefits of flushing over priming the LLC with numerous randomly chosen addresses. Under ideal conditions, when the attacker only employs purging, hitting an eviction set entry of G with a purge access becomes

$$P_{success} = P_{hit,set} \cdot P_{hit,way} = \frac{1}{|sets|} \cdot \frac{1}{|ways|},$$

as the random address must fall into the same set as the eviction set entry and also must hit the eviction set entry by random chance. By flushing the eviction set G before performing purging, we can actively modify the $P_{hit,way}$ probability and enhance the purge step. More specifically, flushing sets the probability $P_{hit,way} = 1$, because the cache will prioritize the flushed locations previously occupied by G. In general, the success probability of removing a stuck eviction set using Flush+Purge is larger than that of pure purging because:

$$\frac{1}{|sets|} > \frac{1}{|sets|} \cdot \frac{1}{|ways|},$$

as long as there are at least two ways, which can be safely assumed for typical cache configurations. When using Flush+Purge the randomly chosen purge address only needs to hit the target set. As a result, the subsequent prime step will place addresses of G in new locations, increasing the likelihood of evicting the victim. In the best-case scenario, this combination of flush and purging completely fills the gaps left by the eviction set, increasing the probability of evicting the victim to the original catching probability. If the purging step does not fill the gap completely, some entries of the eviction set may still be placed at their old location, thus not affecting the victim. This ultimately leads us to higher success rates of letting the attack continue without increasing the number of addresses used during the purge.

5. Evaluation

In this section, we evaluate the previously discussed remedies enabling the attacker continue the attack. We use the following cache configurations (See Appendix A for additional configurations):

1) CEASER-S [30] with two partitions
2) ScatterCache [47]
3) PhantomCache [41] with eight random cache sets.

We omit CEASER [31] because, as mentioned in Section 3.3, it is not susceptible to the issue. We first evaluate our proposed solutions individually for each randomized cache architecture. Then, we compare the results to see

which solution works the best and which cache gets stuck the most.

Setup. We use CacheFX [10] as the cache simulator. In all configurations we use a cache of size 4096 cache lines, with random replacement, and enable a (previously existing) CacheFX flag to prioritize invalid caches lines. We use the single access attacker and victim, extending the attacker with implementations of the Purge, Flush+Purge, and using multiple eviction sets strategies. For purge accesses we use 10% of the number of total cache lines.

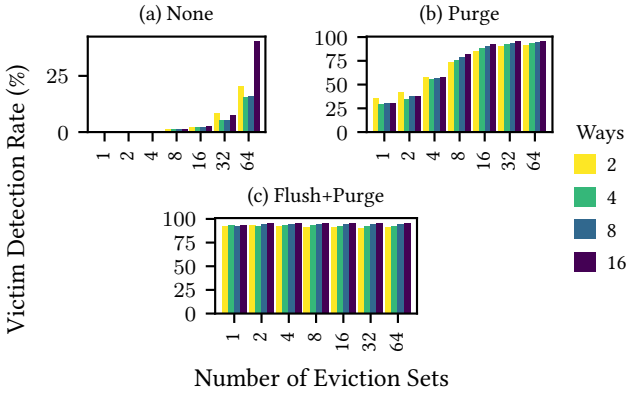


Figure 3: Plot for CEASER-S 2 over the number of ways for different solutions (None, Purge, Flush+Purge) and with different numbers of rotating eviction sets.

CEASER-S. We first evaluate 1–64 rotating eviction sets and with no additional purge step. As seen in Figure 3a, with few eviction sets the detection rate remained low for all associativities. This is due to the small eviction set sizes of only eight addresses for two ways and 16, 34, and 68 for other associativities, respectively. Since the total size of the eviction sets is only a fraction of the cache size, all eviction sets get stuck early. For high numbers of eviction sets (32, 64), the detection rate finally increased. Still, detection rates are below 25%. It even decreased comparing two and four ways of associativity, although an increase in total number of addresses. For 64 rotating eviction sets and 16 ways the detection rate spiked, reaching a 40% success rate. This can be attributed to the increased total size of the eviction sets, covering over a quarter of the cache, delaying the point where all eviction sets become stuck.

Adding a purge step increased the detection rates to a minimum of around 30% for only one eviction set (Figure 3b). 409 random addresses were accessed after each probe step (10% of all cache lines). In contrast, without any purging, the attacker detected at most two victim accesses with one eviction set. As the number of rotating eviction sets increases, so too does the detection rate. The most significant jumps can be observed from 2 to 4 and 4 to 8. Starting with 16 eviction sets, we witness diminishing returns. Only a minor differences were observed between 32 and 64 eviction sets, regardless of doubling in size. Success rates reached around 95%, which conforms approximately with the probability the eviction sets were initially built for. This highlights

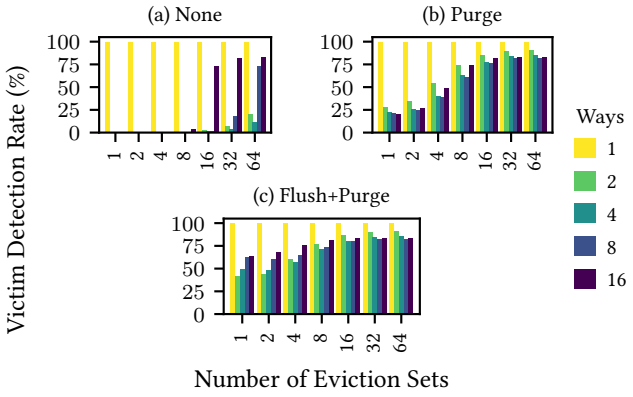


Figure 4: Plot for ScatterCache over the number of ways for different solutions (None, Purge, Flush+Purge) and with different amounts of rotating eviction sets.

the effectiveness of combining the two approaches, purging and rotating eviction sets, by pushing the detection rate from 30% to 95% by investing more computation time in the profiling phase.

The results for FLUSH+PURGE on CEASER-S with two partitions are shown in Figure 3c. Detection rates constantly reach around 95% over all cases with only minor differences between associativities. This highlights the effectiveness of flushing the eviction set and letting the purging replace the invalid cache lines.

ScatterCache. Figure 4a shows the results for ScatterCache with only rotating eviction sets. With one way, the detection rate was 100% due to the direct-mapped nature of this configuration. For low numbers of eviction sets the results are similar to those of CEASER-S. Starting with 16 eviction sets, the detection rate started to increase for higher associativities. This is due to quickly increasing eviction set sizes for those higher associativities in ScatterCache. For example, for eight ways of associativity, the size of one eviction set with a targeted eviction probability of 90% is approximately 135, resulting in 8640 used addresses when using 64 such eviction sets. This is more than double the cache size; however, the success rate does not reach 90%, indicating that this number is still insufficient for reliably attacking ScatterCache. While there is a factor of two in the total number of addresses between 32 and 64 eviction sets, this seems to have only a minimal impact on the success rate.

Performing purging after the probe step again increased the detection rates (Figure 4b). The 25% mark is already crossed using two one eviction set over all associativities. However, we observed lower values than for CEASER-S 2. Using more eviction sets constantly increased the detection rate, with the biggest jumps being from two to four and from four to eight. They peak at around 90% for 32 and 64 eviction sets. With an increasing number of ways, the detection rates declined until eight ways. This is due to ScatterCache effectively scaling with the number of ways and becoming harder to attack. For 16 ways of associativity,

detection rates increased again. This is the result of the quickly growing eviction set sizes, which in this case reaches over 500 addresses. In total, purging increased the detection rates in contrast to using only rotating eviction sets, however, not as well as for CEASER-S 2. Additionally, the detection rate being around 20–25% for one eviction set indicates that if the attacker gets stuck, he stays in that state for around three iterations on average. This confirms our calculated probability of 0.27 that the purge step can evict the victim and enable the attacker to detect further accesses.

Adding flushing of the eviction sets mostly increased the detection rates for smaller numbers of eviction sets (Figure 4c). For example, with two and four eviction sets, the detection rate is at its minimum at 40% for two and four ways and even reaches over 60% for eight and 16 ways. Generally, for two, four, and eight eviction sets there is an increase in detection rate with increasing ways of associativity compared to using only purging. We still see a decline with growing associativity for the higher numbers of eviction sets. While the additional flush does not further push the detection rate up for high numbers of eviction sets, it enables a smaller number to reach a much higher detection rate. In this way, the time required during the profiling phase to construct numerous eviction sets can be reduced by employing fewer sets together with an additional flush.

PhantomCache. For only rotating eviction sets, we observed clear differences to CEASER-S and ScatterCache (Figure 5a). Starting at four eviction sets, detection rates reached 90% for high associativity. Increasing the number of eviction sets brought up the detection rates for lower associativities one by one. This is primarily due to the high number of addresses per eviction set for smaller associativities compared to ScatterCache and CEASER-S. One eviction set for 16 ways contains over 1035 addresses. Computing a set of such size takes significant effort in the profiling phase.

Figure 5b shows that with purging the detection rate improved, particularly for configurations with low initial detection rates. Due to the quickly growing eviction set size, the detection rate rises as the associativity increases. For example, the detection rate for two eviction sets and 16 ways already reach 70%. Starting with 16 eviction sets, detection rates cross the 75% mark for all associativities.

Adding an eviction set flush (Figure 5c) further increased detection rates, bringing all detection rate to a minimum of 75%. While the results are higher than for ScatterCache, we see a strong dependency to the associativity. Detection rates of 95% are only reached for high associativities because of very large evaluation sets. Still, the results again highlight the effectiveness of flushing the eviction set, boosting previous detection rates of 25% to over 75%.

5.1. Comparison

Comparing the results for all three caches, it is clear that the `Flush+Purge` technique yields the best results. Combining it with rotating eviction sets can result in even higher detection rates and lower the attacker's probability of getting stuck. From the point of getting stuck and detection

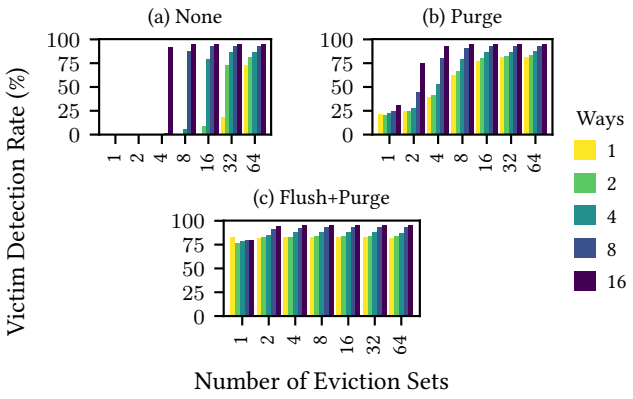


Figure 5: Plot for PhantomCache 8 over the number of ways for different solutions (None, Purge, Flush+Purge) and with different amounts of rotating eviction sets.

rates, ScatterCache [47], with high associativities, seems to be the best option in terms of security since it has the overall lowest rates, although not by much. For PhantomCache [41], the detection rates are high even without any purging due to the size of a single eviction set, and for CEASER-S [30], we also see a higher detection rate even though it is very similar to ScatterCache.

5.2. AES Case-Study

For evaluation in a practical context, we conducted a case-study attacking an AES T-table implementation. Here, we focus on two aspects: (1) highlighting the difference between only using noise and only flushing cache lines, (2) showing that applying both at the same time, yields values between the two cases of (1). Rotating eviction sets are omitted here, as their complexity would have obscured the results and insights of primary interest. Creating an eviction set requires around 100 times more accesses than the total accesses needed for the attack.

For this case study, we extended the AES attack implementation of CacheFX, enabling the AES attacker to also implement Purge and Flush+Purge. We used ScatterCache with 4096 total cache lines, four ways, and random replacement. We ran the attack 1000 times for each case, and report in Table 1 the medians of the number of encryptions and the number of memory accesses required to break the AES implementation. The first two rows show the case where neither noise nor flushing is used to dislodge the stuck eviction set or victim. The attack stalls quickly, irrespective of whether invalid lines are prioritized.

To implement probabilistic eviction [10, 28], we configured CacheFX to accesses random memory addresses amounting to 10% of the number of cache lines. The attack does not get stuck, requiring about 10k encryptions and 3M memory accesses. Increasing the number of random accesses increases the probability of evicting the eviction set or the victim, reducing the number of encryptions required

Table 1: Complexity of attacking an AES T-table implementation.

Flush	Noise	Prefer Invalid CLs	Result	Values Total (Noise fixed to 10%)		Values per iteration (Noise adjusted to reach Flush+Purge)			
				Encryptions	Memory Accesses (incl. noise)	Noise (%)	Memory Accesses (incl. noise)	Flush	Time (ns)
✗	✗	✗	stuck	-	-	-	-		-
✗	✗	✓	stuck	-	-	-	-		-
✗	✓	✗	not stuck	10027	3061125	80	2404	0	166043
✗	✓	✓	not stuck	10353	3154048	80	2412	0	166120
✓	✗	✗	not stuck	2361	326440	0	119	17	5038
✓	✗	✓	stuck	-	-	-	-		-
✓	✓	✗	not stuck	2839	693831	10	451	17	25624
✓	✓	✓	not stuck	4561	1233768	10	544	22	30595

to break AES, but, at the same time, increasing the number of memory accesses.

To improve the attack, Song et al. [38] proposed flushing the eviction set after each probe step instead of using costly memory accesses. Our results confirm that, without invalid line prioritization, this strategy reduces the number of required encryptions by a factor of five and the total number of memory accesses by an order of magnitude. However, when invalid cache lines are filled first, the attack gets stuck, as described in Section 4.

Notably, applying both probabilistic eviction and flushing only doubles the number of memory accesses in the case where no cache lines are preferred. In the case where invalid cache lines are prioritized, the attack no longer gets stuck. However, we see a rise in complexity with 60% more encryptions and 77% more memory accesses required to break AES compared to only flushing the eviction set.

To verify our claim that Flush+Purge allows for a finer attack granularity than using only Purge, we looked at the number of accesses required to complete one attack round. Comparing the number of accesses per iteration requires the tested configurations to target the same success rate, i.e., the number of encryptions to succeed. Running all configurations with the same number of noise accesses would result in almost identical runtimes of one iteration, since the number of accesses (and flushes) per round is only determined by the eviction set size, and the number of noise accesses. In this case, less optimal configurations will simply take more iterations to succeed. To make them comparable, we chose Flush+Purge as baseline, which required around 4500 encryptions to succeed. Reaching this effectiveness using only Purge requires more noise accesses than the 10% of all available cache lines previously used. Specifically, we had to raise the percentage to around 80% to reach a comparable effectiveness to Flush+Purge. We then compared the average number of memory accesses, including noise, and the average number of flushes. We then used these values to establish a time estimate per iteration using LLC access cycles from a recent Intel 7 Ultra 265K, assuming a CPU frequency of 3GHz. Using our own benchmark we measured 60 cycles for an LLC hit and 225 cycles for an LLC miss. Flush cycles were measured with 400 for a hit and 200 for a miss, using the artifact of Rauscher et al. [33].

Evaluating the configuration on noise accesses (Purge) only, raising their number to around 80% of the total cache lines would result in 3276 noise accesses per iteration for our cache. However, the noise is only accessed when no victim access was detected, and thus the average number of noise accesses and overall number memory accesses, is slightly lower, at around 2400. We therefore estimate that a full attack round would take a median of 166.043μs. Flushing the eviction set without any noise significantly reduces the time per iteration to just 5μs. However, this strategy is only applicable, when invalid cache lines are not prioritized. Flushing and noise (Flush+Purge) results in 30.595μs in the case of invalid cache line prioritization. This confirms the better granularity of Flush+Purge compared to just using noise for the same effectiveness, lowering the runtime of an attack round by a factor of around five.

From these result, we can draw four conclusions:

- Flushing the eviction set after each probe step, as Song et al. [38] proposed, accelerates the attack.
- In a practical scenario, where invalid cache lines are filled first, the flush technique fails.
- Combining flushing and noise yields slightly worse performance than just flushing, but allows the attack to function when invalid cache lines are filled first.
- Although slightly slower than just flushing the eviction set, the combined approach still outperforms probabilistic eviction methods [10, 28].

6. Related Work

After the disclosure of the PRIME+PRUNE+PROBE attack [28], several randomized cache architectures that are secure against this type of profiling attack have been proposed. These architectures require additional complexity in the cache design. Mirage [34] divides the cache into a randomized tag-store and a data-store. Using this method, Mirage can replicate the behavior of a fully associative cache while maintaining decent complexity on the access path. ClepsydraCache [43] combines cache randomization with a decay mechanism, which makes observing cache conflict very challenging for an attacker. GuardCache [23] also implements a randomized cache architecture and introduces false cache misses and false cache hits to create noise and, thus, prevent the attacker from making useful observations.

Random Fill Cache [19] presents an alternative approach that is not based on randomization. Instead of caching an address on access, it caches an address close to the accessed address. Thus, the attacker cannot observe which exact address has been accessed. SassCache [11] is a skewed set-associative cache using a keyed index derivation function. Additionally, it splits the cache into partially overlapping security domains.

Several works have investigated the security of randomized cache architectures. Bodduna et al. [3] demonstrated that an insecure randomization function can lead to a trivial bypass of the randomization. In response, SCARF [4] proposes a cryptographically secure randomization function for 10-bit indexed caches focusing on low latency. The impact of the replacement policy on the security of randomized caches has been investigated in [27].

7. Conclusion

In this paper, we revisited a known but sidelined problem with PRIME+PRUNE+PROBE. In particularly, using the same eviction set over and over again to observe victim accesses leads to the eviction set and the victim address to be both cached at some point. This results in the eviction set becoming useless to the attacker, and they have to somehow either evict their eviction set or the victim. We first did a theoretical analysis of when and why this happens and confirmed our findings using the CacheFX [10] simulator. Next, we discussed three in previous literature proposed techniques to resolve the aforementioned problem. First, using multiple eviction sets increases the amount of time spend in the profiling phase while keeping a high attack granularity due to fast prime and probe steps. Utilizing purge accesses to probabilistically evict parts of the eviction set or the victim itself proofed as solid technique. Here the attacker needs only one eviction set, accelerating profiling, but reducing the attack granularity since the purge accesses delay following iterations. We found that the proposed technique by Song et al. [38] of flushing the eviction set after each probe does not function as intended when prioritizing invalid caches lines, as it is the case in common cache architectures. To this end we improved on their idea and propose a combined technique FLUSH+PURGE which re-enables flushing. All techniques were evaluated in CacheFX to see how they performed and set them into context attacking an AES T-table implementation. As a result, we show that using FLUSH+PURGE is almost twice as fast as only purge accesses and effectively enables an attacker to repeatedly use the same eviction set to observe victim accesses.

Acknowledgments

We would like to thank Daniel Gruss for the discussion, advice, and support.

This work was supported by an ARC Discovery Project number DP210102670; the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy - EXC 2092 CASA - 390781972 and Priority Program SPP 2253 Nano Security through the projects RAINCOAT II (440059533) and EMBOSOM (535695900); and the German Federal Ministry of Research, Technology and Space (BMFTR) through the project MANNHEIM-FlexKI (01IS22086I).

References

[1] Andreas Abel and Jan Reineke, "nanobench: A low-overhead tool for running microbenchmarks on x86 systems," in *IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2020, Boston, MA, USA, August 23-25, 2020*. IEEE, 2020, pp. 34–46. https://doi.org/10.1109/ISPASS48437.2020.00014. https://doi.org/10.1109/ISPASS48437.2020.00014

[2] Mohammad-Mahdi Bazm, Thibaut Sautereau, Marc Lacoste, Mario Südholt, and Jean-Marc Menaud, "Cache-based side-channel attacks detection through Intel cache monitoring technology and hardware performance counters," in *Fog and Mobile Edge Computing (FMEC 2018)*, 2018, pp. 7–12. https://doi.org/10.1109/FMEC.2018.8364038

[3] Rahul Bodduna, Vinod Ganesan, Patanjali SLPSK, Kamakoti Veezhinathan, and Chester Rebeiro, "Brutus: Refuting the security claims of the cache timing randomization countermeasure proposed in CEASER," *IEEE Comput. Archit. Lett.*, vol. 19, no. 1, pp. 9–12, 2020. https://doi.org/10.1109/LCA.2020.2964212

[4] Federico Canale, Tim Güneysu, Gregor Leander, Jan Philipp Thoma, Yosuke Todo, and Rei Ueno, "SCARF – a low-latency block cipher for secure cache-randomization," in *USENIX Sec*, 2023, pp. 1937–1954. https://www.usenix.org/conference/usenixsecurity23/presentation/canale

[5] Marco Chiappetta, Erkay Savas, and Cemal Yilmaz, "Real time detection of cache-based side-channel attacks using hardware performance counters," *Applied Soft Computing*, vol. 49, pp. 1162–1174, 2016. https://doi.org/10.1016/j.asoc.2016.09.014

[6] Sanjeev Das, Jan Werner, Manos Antonakakis, Michalis Polychronakis, and Fabian Monrose, "SoK: The challenges, pitfalls, and perils of using hardware performance counters for security," in *IEEE SP*, 2019, pp. 20–38.

[7] John Demme, Matthew Maycock, Jared Schmitz, Adrian Tang, Adam Waksman, Simha Sethumadhavan, and Salvatore J. Stolfo, "On the feasibility of online malware detection with performance counters," in *ISCA*, 2013, pp. 559–570. https://doi.org/10.1145/2485922.2485970

[8] Ghada Dessouky, Tommaso Frassetto, and Ahmad-Reza Sadeghi, "HybCache: Hybrid side-channel-resilient caches for trusted execution environments," in *USENIX Sec*, 2020,

pp. 451–468. https://www.usenix.org/conference/usenixsecurity20/presentation/dessouky

[9] Craig Disselkoen, David Kohlbrenner, Leo Porter, and Dean M. Tullsen, "Prime+Abort: A timer-free high-precision L3 cache attack using Intel TSX," in *USENIX Sec*, 2017, pp. 51–67. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/disselkoen

[10] Daniel Genkin, William Kosasih, Fangfei Liu, Anna Trikalinou, Thomas Unterluggauer, and Yuval Yarom, "CacheFX: A framework for evaluating cache security," in *AsiaCCS*, 2023, pp. 163–176. https://doi.org/10.1145/3579856.3595794

[11] Lukas Giner, Stefan Steinegger, Antoon Purnal, Maria Eichlseder, Thomas Unterluggauer, Stefan Mangard, and Daniel Gruss, "Scatter and split securely: Defeating cache contention and occupancy attacks," in *IEEE SP*, 2023, pp. 2273–2287. https://doi.org/10.1109/SP46215.2023.10179440

[12] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard, "Flush+Flush: A fast and stealthy cache attack," in *DIMVA*, 2016, pp. 279–299. https://doi.org/10.1007/978-3-319-40667-1_14

[13] Zecheng He and Ruby B. Lee, "How secure is your cache against side-channel attacks?" in *MICRO*, 2017, pp. 341–353. https://doi.org/10.1145/3123939.3124546

[14] Ralf Hund, Carsten Willems, and Thorsten Holz, "Practical timing side channel attacks against kernel space ASLR," in *IEEE SP*, 2013, pp. 191–205. https://doi.org/10.1109/SP.2013.23

[15] Tom Kessous and Niv Gilboa, "Prune+PlumTree - finding eviction sets at scale," in *IEEE SP*, 2024.

[16] 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 *IEEE SP*, 2019, pp. 1–19. https://doi.org/10.1109/SP.2019.00002

[17] William Kosasih, Yusi Feng, Chitchanok Chuengsatiansup, Yuval Yarom, and Ziyuan Zhu, "SoK: Can we really detect cache side-channel attacks by monitoring performance counters?" in *AsiaCCS*. ACM, 2024. https://doi.org/10.1145/3634737.3637649

[18] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg, "Meltdown: Reading kernel memory from user space," in *USENIX Sec*, 2018, pp. 973–990. https://www.usenix.org/conference/usenixsecurity18/presentation/lipp

[19] Fangfei Liu and Ruby B. Lee, "Random fill cache architecture," in *MICRO*, 2014, pp. 203–215. https://doi.org/10.1109/MICRO.2014.28

[20] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee, "Last-level cache side-channel attacks are practical," in *IEEE SP*, 2015, pp. 605–622. https://doi.org/10.1109/SP.2015.43

[21] Fangfei Liu, Hao Wu, Kenneth Mai, and Ruby B. Lee, "Newcache: Secure cache architecture thwarting cache side-channel attacks," *IEEE Micro*, vol. 36, no. 5, pp. 8–16, 2016. https://doi.org/10.1109/MM.2016.85

[22] Xiaoxuan Lou, Tianwei Zhang, Jun Jiang, and Yinqian Zhang, "A survey of microarchitectural side-channel vulnerabilities, attacks, and defenses in cryptography," *ACM Comput. Surv.*, vol. 54, no. 6, pp. 122:1–122:37, 2022. https://doi.org/10.1145/3456629

[23] Fernando Mosquera, Krishna Kavi, Gayatri Mehta, and Lizy K. John, "Guard cache: Creating noisy side-channels," *IEEE Comput. Archit. Lett.*, vol. 22, no. 2, pp. 97–100, 2023. https://doi.org/10.1109/LCA.2023.3289710

[24] Maria Mushtaq, Ayaz Akram, Muhammad Khurram Bhatti, Maham Chaudhry, Vianney Lapotre, and Guy Gogniat, "NIGHTs-WATCH: a cache-based side-channel intrusion detector using hardware performance counters," in *HASP@ISCA*, 2018, pp. 1:1–1:8. https://doi.org/10.1145/3214292.3214293

[25] Dag Arne Osvik, Adi Shamir, and Eran Tromer, "Cache attacks and countermeasures: the case of AES," in *CT-RSA*, 2006, pp. 1–20. https://doi.org/10.1007/11605805_1

[26] Dan Page, "Partitioned cache architecture as a side-channel defence mechanism," IACR Cryptol. ePrint Arch. 2005/280, 2005. http://eprint.iacr.org/2005/280

[27] Moritz Peters, Nicolas Gaudin, Jan Philipp Thoma, Vianney Lapôtre, Pascal Cotret, Guy Gogniat, and Tim Güneysu, "On the effect of replacement policies on the security of randomized cache architectures," *ASIACCS 2024*, 2024.

[28] Antoon Purnal, Lukas Giner, Daniel Gruss, and Ingrid Verbauwhede, "Systematic analysis of randomization-based protected cache architectures," in *IEEE SP*, 2021, pp. 987–1002.

[29] Antoon Purnal, Furkan Turan, and Ingrid Verbauwhede, "Prime+Scope: Overcoming the observer effect for high-precision cache contention attacks," in *CCS*, 2021, pp. 2906–2920.

[30] Moinuddin K. Qureshi, "New attacks and defense for encrypted-address cache," in *ISCA*, 2019, pp. 360–371. https://doi.org/10.1145/3307650.3322246

[31] Moinuddin K. Qureshi, "CEASER: mitigating conflict-based cache attacks via encrypted-address and remapping," in *MICRO*, 2018, pp. 775–787. https://doi.org/10.1109/MICRO.2018.00068

[32] Moinuddin K. Qureshi and Yale N. Patt, "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches," in *MICRO*, 2006, pp. 423–432. https://doi.org/10.1109/MICRO.2006.49

[33] Fabian Rauscher, Luca Wilke, Hannes Weissteiner, Thomas Eisenbarth, and Daniel Gruss, "TDXploit: Novel techniques for Single-Stepping and cache attacks on intel TDX," in *34th USENIX Security Symposium (USENIX Security 25)*, 2025, pp. 1207–1222.

[34] Gururaj Saileshwar and Moinuddin K. Qureshi,

"MIRAGE: mitigating conflict-based cache attacks with a practical fully-associative design," in *USENIX Sec*, 2021, pp. 1379–1396. https://www.usenix.org/conference/usenixsecurity21/presentation/saileshwar

[35] Daniel Sánchez and Christos Kozyrakis, "Scalable and efficient fine-grained cache partitioning with Vantage," *IEEE Micro*, vol. 32, no. 3, pp. 26–37, 2012. https://doi.org/10.1109/MM.2012.19

[36] Sercan Sari, Onur Demir, and Gurhan Kucuk, "FairSDP: Fair and secure dynamic cache partitioning," in *Conference on Computer Science and Engineering (UBMK)*, 2019, pp. 469–474.

[37] Wei Song and Peng Liu, "Dynamically finding minimal eviction sets can be quicker than you think for side-channel attacks against the LLC," in *RAID*, 2019, pp. 427–442. https://www.usenix.org/conference/raid2019/presentation/song

[38] Wei Song, Boya Li, Zihan Xue, Zhenzhen Li, Wenhao Wang, and Peng Liu, "Randomized last-level caches are still vulnerable to cache side-channel attacks! but we can fix it," in *IEEE SP*, 2021, pp. 955–969. https://doi.org/10.1109/SP40001.2021.00050

[39] Chao Su and Qingkai Zeng, "Survey of CPU cache-based side-channel attacks: Systematic analysis, security models, and countermeasures," *Secur. Commun. Networks*, vol. 2021, pp. 5 559 552:1–5 559 552:15, 2021. https://doi.org/10.1155/2021/5559552

[40] Jakub Szefer, "Survey of microarchitectural side and covert channels, attacks, and defenses," *J. Hardw. Syst. Secur.*, vol. 3, no. 3, pp. 219–234, 2019. https://doi.org/10.1007/S41635-018-0046-1

[41] Qinhan Tan, Zhihua Zeng, Kai Bu, and Kui Ren, "PhantomCache: Obfuscating cache conflicts with localized randomization," in *NDSS*, 2020. https://www.ndss-symposium.org/ndss-paper/phantomcache-obfuscating-cache-conflicts-with-localized-randomization/

[42] Jan Philipp Thoma and Tim Güneysu, "Write me and I'll tell you secrets - write-after-write effects on Intel CPUs," in *RAID*, 2022, pp. 72–85. https://doi.org/10.1145/3545948.3545987

[43] Jan Philipp Thoma, Christian Niesler, Dominic A. Funke, Gregor Leander, Pierre Mayr, Nils Pohl, Lucas Davi, and Tim Güneysu, "ClepsydraCache - preventing cache attacks with time-based evictions," in *USENIX Sec*, 2023. https://www.usenix.org/conference/usenixsecurity23/presentation/thoma

[44] Eran Tromer, Dag Arne Osvik, and Adi Shamir, "Efficient cache attacks on AES, and countermeasures," *J. Cryptol.*, vol. 23, no. 1, pp. 37–71, 2010. https://doi.org/10.1007/s00145-009-9049-y

[45] Pepe Vila, Boris Köpf, and José F. Morales, "Theory and practice of finding eviction sets," in *IEEE SP*, 2019, pp. 39–54. https://doi.org/10.1109/SP.2019.00042

[46] Zhenghong Wang and Ruby B. Lee, "New cache designs for thwarting software cache-based side channel attacks," in *ISCA*, 2007, pp. 494–505. https://doi.org/10.1145/1250662.1250723

[47] Mario Werner, Thomas Unterluggauer, Lukas Giner, Michael Schwarz, Daniel Gruss, and Stefan Mangard, "ScatterCache: Thwarting cache attacks via cache set randomization," in *USENIX Sec*, 2019, pp. 675–692. https://www.usenix.org/conference/usenixsecurity19/presentation/werner

[48] Yuejian Xie and Gabriel H. Loh, "PIPP: promotion/insertion pseudo-partitioning of multi-core shared caches," in *ISCA*, Stephen W. Keckler and Luiz André Barroso, Eds., 2009, pp. 174–183. https://doi.org/10.1145/1555754.1555778

[49] Zihan Xue, Jinchi Han, and Wei Song, "CTPP: a fast and stealth algorithm for searching eviction sets on Intel processors," in *RAID*, 2023, pp. 151–163. https://doi.org/10.1145/3607199.3607202

[50] Yuval Yarom and Katrina Falkner, "Flush+Reload: a high resolution, low noise, L3 cache side-channel attack," in *USENIX Sec*, 2014, pp. 719–732. https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom

[51] Yuval Yarom, Qian Ge, Fangfei Liu, Ruby B. Lee, and Gernot Heiser, "Mapping the intel last-level cache," IACR Cryptol. ePrint Arch. 2015/905. http://eprint.iacr.org/2015/905

[52] Tianwei Zhang, Yinqian Zhang, and Ruby B. Lee, "CloudRadar: A real-time side-channel attack detection system in clouds," in *RAID*, 2016, pp. 118–140. https://doi.org/10.1007/978-3-319-45719-2_6

[53] Zirui Neil Zhao, Adam Morrison, Christopher W. Fletcher, and Josep Torrellas, "Last-level cache side-channel attacks are feasible in the modern public cloud," in *ASPLOS*, 2024, pp. 582–600. https://doi.org/10.1145/3620665.3640403

[54] Boyou Zhou, Anmol Gupta, Rasoul Jahanshahi, Manuel Egele, and Ajay Joshi, "Hardware performance counters can detect malware: Myth or fact?" in *AsiaCCS*, 2018, p. 457–468. https://doi.org/10.1145/3196494.3196515

Appendix

This appendix is a collection of measurements for the other variants of CEASER-S and PhantomCache.

Figure 6 shows the results measurement results for CEASER-S with only one partition. It shows a detection rate of 100% for all ways and amounts of rotating eviction sets since it behaves like a non-randomized set-associative cache.

The same can be observed for PhantomCache with one random set in Figure 10, as it behaves identically.

Figure 7 shows the results for CEASER-S with four partitions. The only difference to CEASER-S with two ways is a higher detection rate using only 64 eviction sets and an overall lower detection rate when adding a `purge` step and utilizing `Flush+Purge`.

For CEASER-S with eight and sixteen partitions shown in Figure 8 and Figure 9, respectively, the difference repeats

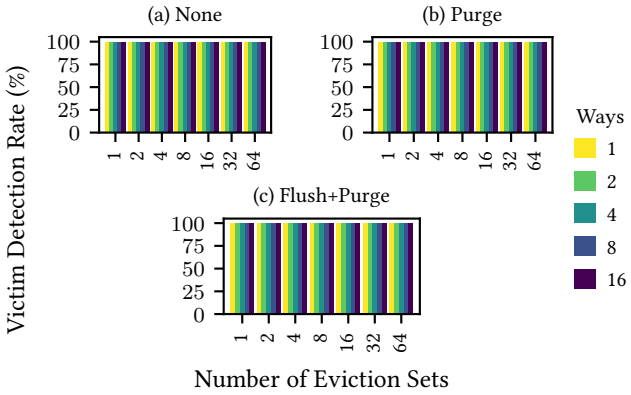


Figure 6: Victim detection rates for CEASER-S 1 over the number of ways for different solutions (None, Purge, Flush+Purge) and with different amounts of rotating eviction sets.

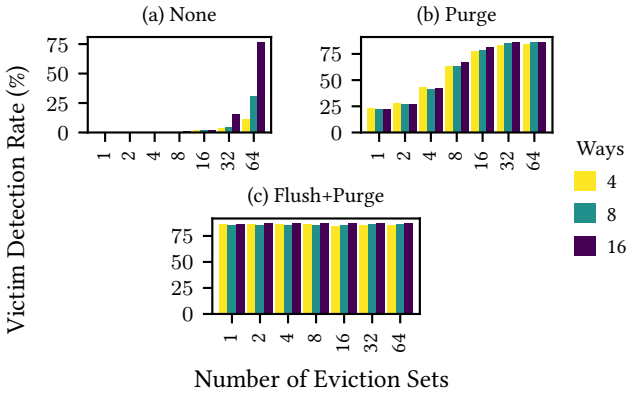


Figure 7: Victim detection rates for CEASER-S 4 over the number of ways for different solutions (None, Purge, Flush+Purge) and with different amounts of rotating eviction sets.

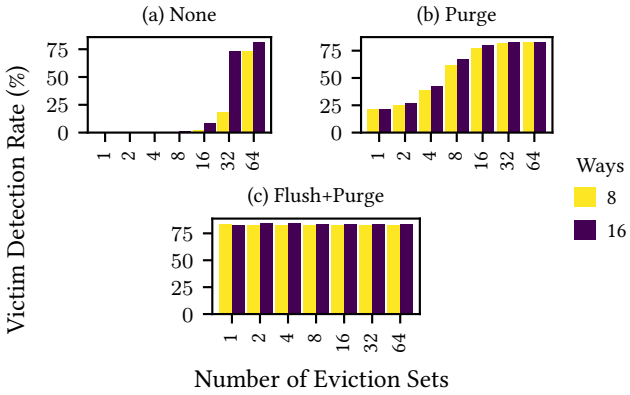


Figure 8: Victim detection rates for CEASER-S 8 over the number of ways for different solutions (None, Purge, Flush+Purge) and with different amounts of rotating eviction sets.

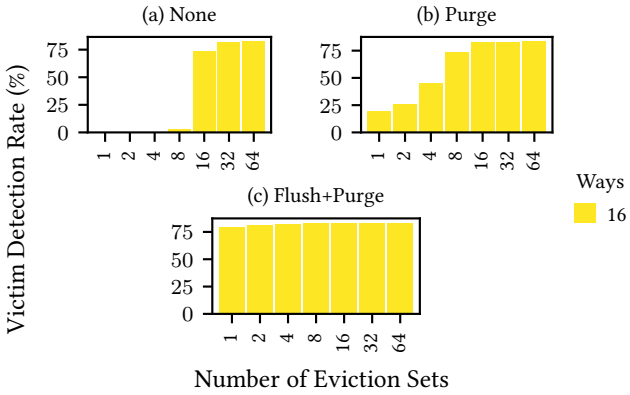


Figure 9: Victim detection rates for CEASER-S 16 over the number of ways for different solutions (None, Purge, Flush+Purge) and with different amounts of rotating eviction sets.

with higher rates using only eviction sets and slightly lower rates for the other two techniques.
In Figure 10 one can find the results for PhantomCache with only one random set. As mentioned, this behaves like a non-randomized set-associative cache with a 100% detection rate.

For PhantomCache with two random sets, as shown in Figure 11, one can observe lower detection rates for smaller associativities and amounts of eviction sets without any purging compared to PhantomCache with eight sets. Combining `purging` with a `flush` increases the detection rate for smaller associativities compared to PhantomCache with eight sets.

Figure 12 shows the results for PhantomCache with four random sets. In this scenario, the detection rate is increased for higher associativities and amounts of eviction sets without any purging. At the same time, it is lower when adding `purging` and a `flush` compared to PhantomCache with two sets.

Finally, in Figure 13, the results for PhantomCache with sixteen ways are presented. Here, we observe an overall increased detection rate, mostly due to the huge size of an eviction set for high associativities. For sixteen ways, PhantomCache with sixteen random sets behaves almost like a fully-associative cache. Therefore, the size of one eviction set alone exceeds the total size of the cache, leading to an almost 100% detection rate.

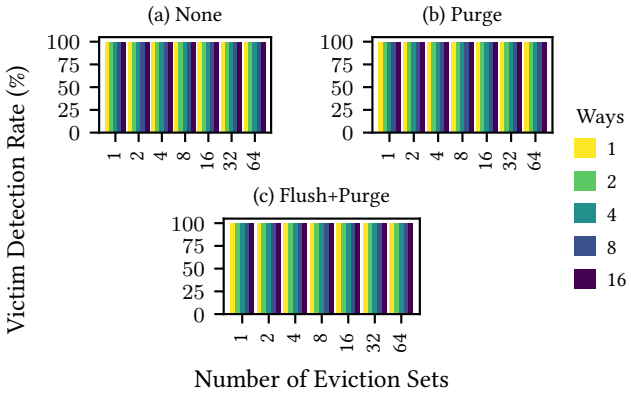


Figure 10: Victim detection rates for PhantomCache 1 over the number of ways for different solutions (None, Purge, Flush+Purge) and with different amounts of rotating eviction sets.

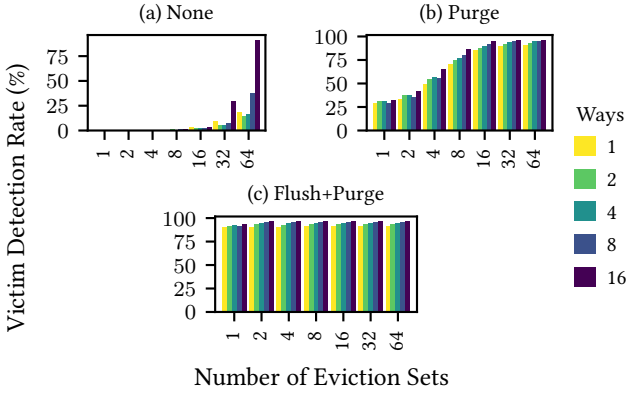


Figure 11: Victim detection rates for PhantomCache 2 over the number of ways for different solutions (None, Purge, Flush+Purge) and with different amounts of rotating eviction sets.

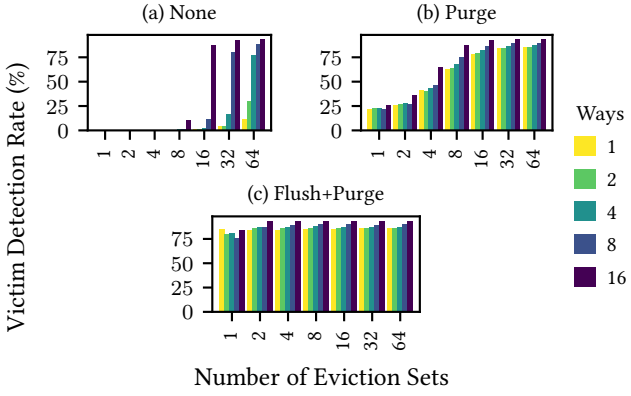


Figure 12: Victim detection rates for PhantomCache 4 over the number of ways for different solutions (None, Purge, Flush+Purge) and with different amounts of rotating eviction sets.

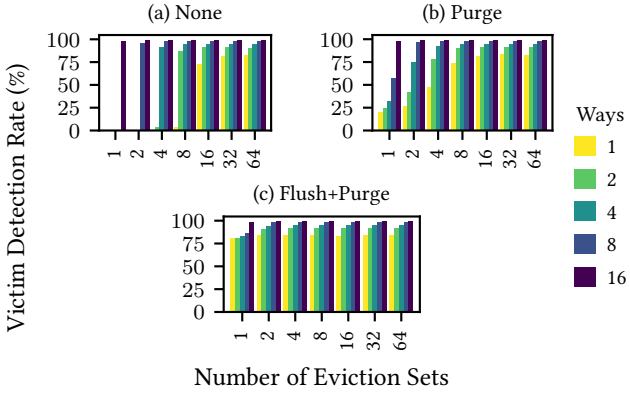


Figure 13: Victim detection rates for PhantomCache 16 over the number of ways for different solutions (None, Purge, Flush+Purge) and with different amounts of rotating eviction sets.