Systematic Analysis of Randomization-based Protected Cache Architectures

Antoon Purnal[*], Lukas Giner[†], Daniel Gruss[†], and Ingrid Verbauwhede[*]

[*]imec-COSIC, KU Leuven [†]Graz University of Technology

***Abstract*—Recent secure cache designs aim to mitigate side-channel attacks by randomizing the mapping from memory addresses to cache sets. As vendors investigate deployment of these caches, it is crucial to understand their actual security.**

In this paper, we consolidate existing randomization-based secure caches into a generic cache model. We then comprehensively analyze the security of existing designs, including Ceaser-S and ScatterCache, by mapping them to instances of this model. We tailor cache attacks for randomized caches using a novel Prime+Prune+Probe technique, and optimize it using burst accesses, bootstrapping, and multi-step profiling. Prime+Prune+Probe constructs probabilistic but reliable eviction sets, enabling attacks previously assumed to be computationally infeasible. We also simulate an end-to-end attack, leaking secrets from a vulnerable AES implementation. Finally, a case study of Ceaser-S reveals that cryptographic weaknesses in the randomization algorithm can lead to a complete security subversion.

Our systematic analysis yields more realistic and comparable security levels for randomized caches. As we quantify how design parameters influence the security level, our work leads to important conclusions for future work on secure cache designs.

I. Introduction

Caches reduce the latency for memory accesses with high locality. This is crucial for performance but also an inherent side channel that has been exploited in many microarchitectural attacks, e.g., on cryptographic implementations [3], [34], [54], [14], user input [40], [33], [12], [32], system secrets [13], [16], [9], covert channels [27], [11], [30], and transient-execution attacks like Spectre [19], [6], [4] and Meltdown [23], [47].

Due to the limited size of the cache, some addresses are bound to be allocated to the same cache set, *i.e.*, they are *congruent* and contend for the same resources. While some attacks are enabled by the attacker's capability to flush cache lines, others work purely with this *cache contention*. The basic building block for measuring cache contention is the *eviction set*, a set of congruent addresses. Accessing the addresses in this eviction set brings the cache into a known state. Measuring how long this takes, tells the attacker whether some process worked on congruent addresses since the last eviction.

To mitigate contention-based attacks, the cache hardware can be augmented to so-called protected cache architectures. One line of work reduces interference through better isolation [41], [17], [50], [55], [56], [24], [18], [42], or partial isolation (e.g., locking cache lines) [51], [8]. Another promising line of work is *randomized* cache architectures [51], [52], [21], [25], [26], [45], [38], [39], [53], which randomize the otherwise predictable mapping of memory addresses to cache sets. Several recently proposed randomized caches [45], [38], [39], [53] evaluate a dedicated hardware mapping to perform the randomization on the fly. Consequently, these architectures only slightly change the interface to the outside, and can maintain efficient and scalable sharing of caches. However, even if the randomized mapping is (cryptographically) unpredictable, there are cache collisions due to the limited size of the cache. Hence, existing proposals incorporate some notion of *rekeying*, *i.e.*, renewing randomization at runtime. This limits the temporal window in which eviction sets can be used for an attack.

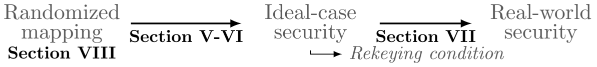


Fig. 1: Security argument for randomized caches.

While randomized cache architectures show promise to thwart eviction-based cache attacks with reasonable overhead, supporting them with quantified security claims (a default for cryptographic algorithms) is challenging. Figure 1 depicts the established security argument. The randomized mapping is used as a trust anchor for security in ideal attack conditions, yielding a (conservative) estimate for the rekeying condition. Currently, the security transfer from the randomization mapping to ideal-case security is not well-understood, which we highlight by improving state-of-the-art attacks by orders of magnitude. Assuming that system activity increases the attack complexity, ideal-case security implies real-world security. However, it is unclear to which extent the rekeying condition can be relaxed.

The high interest in these novel cache designs and their seeming relevance to mitigate a growing list of attacks motivates the following fundamental questions of this paper:

Can we accurately compare security levels for randomized caches? How realistic are security levels reported for secure randomized caches? Do secure randomized caches provide substantially higher security levels than regular caches?

In this paper, we systematically cover the attack surface of randomization-based protected caches. We consolidate existing proposals into a generic randomized cache model, and identify attacker objectives in such caches. We then analyze this model, resulting in a comprehensive and parametrized analysis, serving as a baseline for future secure caches and their analysis.

We present Prime+Prune+Probe (PPP), a technique to find probabilistic but reliable eviction sets in randomized caches. Improving the approach by Werner et al. [53], PPP dramatically outperforms traditional eviction, turning infeasible attacks (e.g.,

$>10^{30}$ accesses) into feasible ones (e.g., $<10^{7}$ accesses).

We also analyze security under complicating system effects, e.g., noise and multiple victim accesses, culminating with successful key recovery from a vulnerable AES implementation.

Latency constraints associated with the cache hierarchy have inspired designers to invent new [38], [45] or repurpose existing [45] low-latency structures for the randomization mapping. Security arguments then rely on their alleged unpredictability. We falsify this assumption for CEASER-S, and propose that future designs use mappings that resist extensive cryptanalysis.

Contributions. In summary, our main contributions are:

- We consolidate existing proposals into a generic randomized cache architecture model.
- We derive a comprehensive and parametrized analysis of all computation-based randomized cache architectures. We improve noise-free attacks by several orders of magnitude.
- We analyze non-ideal effects in profiling on randomized caches, and demonstrate the first end-to-end attack.
- We study the security requirements of the core randomized mapping and show that the security of CEASER-S can be completely subverted, even with frequent rekeying.

Outline. Section II provides background. Section III presents our generalized cache model. Section IV generalizes contention-based attacks for randomized caches. Section V presents ideal-case eviction set construction, Section VI describes optimizations, and Section VII considers aggravating system effects. Section VIII shows how exploiting internals can completely subvert security guarantees. Section IX discusses results and compares existing proposals. Section X concludes.

II. BACKGROUND

A. Caches and Cache Hierarchies

CPUs hide memory latency using caches to buffer data expected to be used in the near future. Caches are organized in cache lines. In a directly mapped cache, each memory address can be cached by exactly one of the cache lines, determined by a fixed address-based mapping. If a memory address can be cached in any cache line, the cache is called fully-associative. If a memory address can only be cached in a (fixed) subset of cache lines, the cache is called set-associative. Addresses mapping to the same set are called *congruent*. Upon a cache line fill request, a replacement policy determines which cache line in the set is replaced. The so-called cache line tag uniquely identifies a cached address. CPU caches can be virtually or physically indexed and tagged, *i.e.*, cache (set) index and the cache line tag are derived from the virtual or physical address.

CPUs have multiple cache levels, with the lower levels being faster and smaller than the higher levels. If all cache lines from a cache A are required to be also present in a cache B, cache B is called *inclusive* with respect to cache A. If a cache line can only reside in one of two cache levels at the same time, the caches are called *exclusive*. If the cache is neither inclusive nor exclusive, it is called *non-inclusive*. The last-level cache (LLC) is often inclusive to lower-level caches and shared across cores to enhance the performance upon transitioning threads between cores and to simplify cache coherency and lookups.

The L1 cache is often considered the lowest level cache. It is usually virtually indexed and physically tagged. All higher-level caches are usually physically indexed and physically tagged.

Again for performance, the last-level cache today is typically composed of multiple independent *slices*, e.g., one slice per physical or logical core. Each (physical) address maps to one of the slices. After selecting the slice, the cache (set) index is selected as described before. The slices are interconnected, e.g., by a ring bus, allowing all cores to access all last-level cache lines. The mapping from physical addresses to slices has been reverse-engineered for certain microarchitectures [28]. In this work, we focus on the complete mapping function which combines the mapping from addresses to slices, sets, and lines.

B. Cache Attacks

Caches reduce the latency of memory accesses with temporal or spatial locality, e.g., recent memory accesses. An attacker can observe the latency and make deductions, e.g., on other recent memory accesses. The first cache attacks deduced cryptographic secrets by observing the execution time [20], [35], [46], [3]. The best techniques today are FLUSH+RELOAD [54] and PRIME+PROBE [34]. FLUSH+RELOAD flushes an address, then waits, and by reloading determines whether the victim accessed it in the meantime. While FLUSH+RELOAD requires a flush instruction to remove a cache line from all cache levels, EVICT+RELOAD [12] uses cache contention. Both FLUSH+RELOAD and EVICT+RELOAD only work on (read-only) memory shared between attacker and victim. PRIME+PROBE [34] overcomes this limitation. PRIME+PROBE measures cache contention instead of memory latency. The attacker fills (primes) a subset of the cache (e.g., a slice, a set, a line) and measures (probes) how long it takes. The time to fill the subset is higher if a victim replaces an attacker cache line with a congruent address.

Mounting PRIME+PROBE requires information about how addresses map to cache lines, which can be gained implicitly in certain scenarios. This is trivial for the L1 cache and, hence, the first PRIME+PROBE attacks targeted the L1 cache [36], [34]. More recently, PRIME+PROBE attacks were mounted on last-level caches [27], [33], [29], [22], [30].

Cache attacks based on cache contention generally consist of two phases. In the *profiling phase*, the attacker finds a so-called *eviction set*, a set of addresses with a high degree of contention in a subset of the cache. In the *exploitation phase*, the attacker accesses this eviction set to bring the cache into a known state. For EVICT+RELOAD, the attacker uses it to evict an entire cache set (including a target address) and to later on reload the target address to determine whether it has been accessed in the meantime. PRIME+PROBE works similarly, except that it does not reload the target address but accesses the eviction set again to measure contention caused by victim memory accesses.

Early approaches for *finding* eviction sets were based on knowing addresses and their congruence, and simply collected a set of such addresses. With address information unavailable, the attacker instead starts with a set of addresses, large enough to be a superset of an eviction set with high probability. Elements are removed from this set until it has minimal size. Recently, this eviction set reduction has been improved from quadratic to linear complexity in the size of the initial set [49], [39].

C. Randomized Cache Architectures

State-of-the-art randomized cache architectures replace predictable address-to-index mappings with deterministic but *random-looking* mappings. The original proposals consider a software-managed look-up table, whereas newer designs compute the randomized mapping on-the-fly in hardware.

1) Table-based architectures: RPCache [51] uses a permutation table to randomize the mapping from memory addresses to cache lines. Occasionally updating the permutation aims to mitigate statistical attacks. Random-fill cache [25] issues cache fill requests to random addresses in spatial proximity instead of the accessed ones. Table-based architectures face scalability issues, which are especially prohibitive for last-level caches.

2) Computation-based architectures: Recent designs (TIME-SECURE CACHE [45], CEASER [38], [39], SCATTERCACHE [53]) cope with this scalability problem by computing the mapping in hardware instead of storing it. This computation should have very low latency. Given their flexibility and scalability, computation-based designs are proposed for last-level caches, which have the largest latency budget and are important to protect as they are usually shared across cores.

3) Cache partitions: Algorithmic advances in eviction set construction [49], [39] have shown that *only* randomizing the memory address is insufficient to protect against contention-based cache attacks. As a key insight, CEASER-S and SCATTERCACHE partition the cache and use the randomized mapping to derive a different cache-set index in each of these partitions. Not only does this significantly raise the bar for *finding* eviction sets, but it also hinders *using* them in the exploitation phase.

4) Rekeying: Even if the mapping from address to cache set in each partition is unpredictable, the attacker can, over time, still identify sets of addresses contending in the cache. Thus, randomized caches rely on *rekeying*, *i.e.*, sampling a new key to refresh the randomization. Selecting an appropriate rekeying condition marks an important security-performance trade-off.

5) Security analysis: Computation-based randomized caches show promise to mitigate cache-based side-channel attacks. Although all proposals come with first-party security analyses, they currently lack a systematic and complete analysis (that we rely on and know, e.g., for cryptographic schemes).

III. GENERIC RANDOMIZED CACHE MODEL

In this section, we present a generic randomized cache model that covers all proposed computation-based randomized caches to this date. We use it to cover the attack surface of randomized caches systematically. In later sections, we will quantify the influence of each parameter on the residual attack complexity.

A. Randomization-based Protected Cache Model

Although some protected cache designs fix the cache configuration, we consider a generic n_w-way set-associative cache with 2^b sets (*i.e.*, b index bits). Then, let $N = n_w \cdot 2^b$ denote the number of cache lines. As with traditional caches, the atomic unit of the mapping from addresses to cache sets is the cache line, for which we assume a generic size of 2^o bytes (*i.e.*, o line offset bits). The model makes abstraction of the line offset bits, as they do not contribute to the randomization.

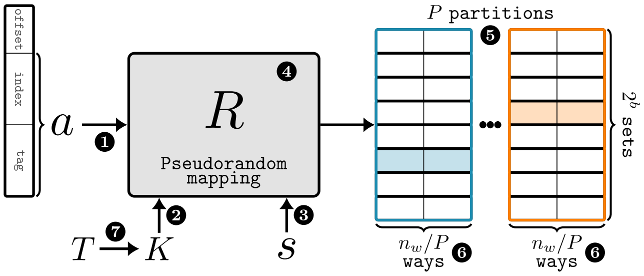


Fig. 2: Computation-based randomized cache model

In accordance with traditional caches, processes cannot monitor the data in the cache directly, nor can they infer to which cache way a certain memory address is allocated. The only interface available is the access latency when reading specific addresses, *i.e.*, it is *low* in case of a cache hit and *high* in case of a miss. In some practical cases, an attacker might also have access to flush semantics. However, our attacks do not rely on it and we thus assume it to be disabled architecturally.

1) Generic model: Figure 2 depicts our generic computation-based randomized cache, featuring the following components:

❶ The **memory address *a*** is the primary input to the randomization design. a is either a physical or virtual address, impacting the degree of control an attacker has over a.

❷ The **key *K*** captures the design's entropy (unpredictability).

❸ The **security domain separator** s optionally differentiates the randomization for processes in different threat domains.

❹ The **randomized mapping** $R_K(a, s)$ is the core of the architecture. It is a *pseudorandom* mapping, *i.e.*, deterministic but random-looking, for which the algorithmic description is publicly known, but the key K is not (Kerckhoff's principle). The LLC slicing function can be encapsulated in R (*i.e.*, one randomized cache), or not (*i.e.*, per-slice randomized caches).

❺ The randomized cache is divided into ***P* partitions**, where $1 \leq P \leq n_w$. An input address a has, in general, a different index in each of these partitions. To accommodate this, R has to supply $P \cdot b$ pseudorandom bits. We assume P divides n_w.

❻ When caching a, one of the partitions is *truly randomly* selected, and the corresponding cache-set index in this partition is determined based on the *pseudorandom* output of R. Then, one of the cache lines in this set is replaced by a, adhering to the **replacement policy within the partition**. We consider random replacement (RAND) and least-recently used (LRU). Under attack, several stateful policies can degenerate to LRU [10].

❼ The **rekeying period** T denotes the condition for entropy renewal. It should be strict enough to maintain high security, and loose enough to maintain high performance.

2) Instantiating Caches: Table I shows how existing designs instantiate this model. The key K can be a cryptographic key (CEASER-S, SCATTERCACHE), a set of cryptographic keys, or selection of a random permutation (TIME-SECURE CACHE). TIME-SECURE CACHE (TSC) implements domain separation

TABLE I: Instantiating the generic model for existing cache designs.

Design	K	s	P	R
Unprotected	$\varnothing$	$\varnothing$	1	slice + bits
TSC [45]	keys / select	$R_{K_s}(a)$	1	HashRP / RM
CEASER [38]	key	$\varnothing$	1	LLBC
CEASER-S [39]	key	?	2-4	LLBC
SCATTERCACHE [53]	key	$R_K(a, s)$	n_w	QARMA [1]

with a per-process key, SCATTERCACHE via additional input to the mapping, and CEASER-S mentions it without implementation details. Traditional unprotected caches, CEASER, and TSC all have one single partition. In SCATTERCACHE, $P = n_w$ (the maximum), whereas CEASER-S recommends $2 \leq P \leq 4$. The rekeying condition T can use, e.g., the wall-clock time, the number of accesses to the cache, or more complex policies.

3) Software Simulator: We implement our model as a C++ randomized LLC simulator, which we parametrize and use to obtain all experimental results in this work. For simulation purposes, many well-analyzed cryptographic primitives can be used for R_K. We use AES because of its hardware support.

B. Attacker Models

We now systematically cover the attack surface of randomized caches and define relevant attacker models in such caches.

Leveraging a provable security methodology from cryptography, we propose to analyze the randomized mapping R (❹) separately from how it is used. On the one hand, we consider *black-box attacks*, which assume that R behaves ideally. In this case, processes cannot efficiently recover K, find inputs to R_K that produce output collisions, or infer any information about cache set indices in one partition based on observations in another. On the other hand, we also consider *shortcut attacks* that exploit R directly. Physical side-channel attacks on R (e.g., using power consumption) are out of scope for this work but can be addressed orthogonally with established approaches [7].

We further assume full attacker control over input address a (❶) as the mapping R should dissolve any attacker control regardless of the input. The key K (❷) is considered full entropy (e.g., generated by a TRNG). If security domains (❸) are supported, we assume that an attacker cannot obtain the same identifier s as the victim. The attacker cannot observe the output of R (❺) directly, but only gather metadata about it by measuring cache contention. Finally, the attacker cannot modify the rekeying condition (❼) (e.g., it is enforced in hardware).

In line with Figure 1, we consider the following three attacks:

$\mathcal{A}_{ideal}$ In an **ideal black-box attack**, the mapping R is considered to behave ideally, and the system is completely noise-free. The victim performs only a single memory access, exactly the one the attacker wants to observe later (cf. Sections V and VI).

$\mathcal{A}_{nonid}$ In a **non-ideal black-box attack**, $\mathcal{A}_{ideal}$ is extended with aggravating system assumptions, and serves to study the increase in attack complexity with respect to $\mathcal{A}_{ideal}$, e.g., noise and multiple victim accesses (cf. Section VII).

$\mathcal{A}_{short}$ In a **shortcut attack**, internals of the mapping R are exploited to find eviction sets much faster than in the black-box case, *i.e.*, a shortcut is found (cf. Section VIII).

Existing analyses [39], [53] study attacker $\mathcal{A}_{ideal}$, as it describes the transfer of security properties from the mapping R_K to the cache architecture (cf. Figure 1). It allows selecting a conservative rekeying condition for a specific design. Besides its general applicability, it also covers some practical settings. For instance, trusted execution environments like Intel SGX are subject to precise control over victim execution [31], *i.e.*, precisely stepping to a single instruction (e.g., a memory access) and even repeating it an arbitrary number of times [48], [43].

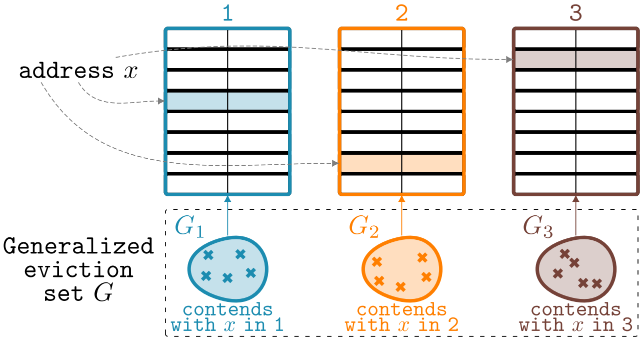


Fig. 3: Generalized eviction sets are based on partial congruence ($n_w = 6$, $P = 3$, $b = \log_2 8 = 3$)

IV. EXPLOITING CONTENTION ON RANDOMIZED CACHES

This section introduces generalized eviction to overcome the challenges introduced by randomized caches. Next, it generalizes traditional attacker objectives to randomized caches.

A. Generalizing Eviction

1) Full congruence: In an eviction set for a traditional cache, every address a_i in this set is *fully congruent* with x. Hence, if x is currently cached, each a_i has the potential to evict it.

In a randomized cache, an attacker can theoretically still find a set of addresses that collide with the target address x in *every partition*. However, the probability for a randomly selected address to be fully congruent with x is 2^{-bP}, *i.e.*, it plummets exponentially with P. Already for $P \geq 2$, relying on full congruence to construct eviction sets is highly impractical.

2) Partial congruence: To overcome the full congruence problem, one can also try to evict a target address x based on *partial congruence*. This approach, introduced by Werner et al. [53] for special case $P = n_w$, constructs an eviction set using addresses congruent with the target in *one partition only*.

To understand eviction with partial congruence in general, consider Figure 3, where the attacker wants to evict a target x in a toy randomized cache with 6 ways ($n_w = 6$), 8 sets ($b = 3$) and 3 partitions ($P = 3$). Assume the attacker has found sets of addresses G_1, G_2, G_3, satisfying that all elements in G_i are congruent with x in partition i but not in the other partitions.

Eviction based on partial congruence is probabilistic. If x is allocated to partition i, it *could* be evicted by G_i. An element in G_i can only contribute to evicting x when it is also assigned to partition i; this assignment is truly random (*i.e.*, not pseudorandom). In what follows, we let a *generalized* eviction set G for a target address x denote the superset of addresses that collide with x in one partition: $G = \bigcup_{i=1}^{P} G_i$.

TABLE II: Generalized eviction set size for several instances.

RP	p_e	$n_w=4$		$n_w=8$			$n_w=16$		
	[%]	$P=2$	$P=4$	$P=2$	$P=4$	$P=8$	$P=2$	$P=4$	$P=16$
RAND	50	6	12	12	24	48	22	44	176
	90	18	36	36	72	144	72	144	576
	95	22	44	46	92	184	94	188	752
LRU	50	6	12	14	28	48	30	60	176
	90	14	36	24	60	144	42	100	576
	95	16	44	26	72	184	46	116	752

3) Eviction probability: Given a target x to evict, we now derive the eviction probability p_e as a function of the size $|G|$ of the generalized eviction set G. We assume that G contains an equal share for every partition, *i.e.*, $|G_i| = \frac{|G|}{P}$, $(1 \leq i \leq P)$. This assumption holds probabilistically in practice, and we will show how it can be met deterministically in Section VI-B.

For replacement policy RAND, the eviction probability generalizes the expression by Werner et al. [53]. Regardless of the partition in which x resides, $\frac{|G|}{P}$ addresses in G could evict it, each with probability n_w^{-1}. Consequently, we have:

$$p_{e,\texttt{RAND}}\left(|G|\right) = 1 - \left(1 - \frac{1}{n_w}\right)^{\frac{|G|}{P}}$$

For LRU, evicting x requires the attacker to evict the full set in the partition in which x currently resides. This corresponds to the event that at least $\frac{n_w}{P}$ out of the $\frac{|G|}{P}$ addresses for the designated partition are actually mapped to this partition. It is described by the complement of the cumulative binomial with $\frac{|G|}{p}$ trials, $\frac{n_w}{P} - 1$ successes and success probability $\frac{1}{P}$:

$$\begin{aligned} p_{e,\texttt{LRU}}\left(|G|\right) &= 1 - \texttt{binom}\left(\frac{|G|}{P}, \frac{n_w}{P} - 1, \frac{1}{P}\right) \\ &= 1 - \sum_{i=0}^{\frac{n_w}{P}-1} \binom{\frac{|G|}{P}}{i} \left(\frac{1}{P}\right)^i \cdot \left(1 - \frac{1}{P}\right)^{\frac{|G|}{P} - i} \end{aligned}$$

Conversely, selecting the eviction probability p_e fixes $|G|$, presented in Table II for different cache configurations and p_e.

B. Generalizing Attacker Objectives

We now generalize eviction set objectives from traditional to randomized caches and evaluate their utility.

A *targeted eviction set* for an address x is a set of addresses that, when accessed, evicts x from the cache with high probability. The complexity and utility of this objective depends on the capability of the attacker to access the target address x.

The attacker can access x if it is an in-process address or resides in memory shared between attacker and victim. By accessing x directly, the attacker can measure its access latency. This objective is useful even in randomized caches, e.g., for EVICT+RELOAD side- and covert channels or to trigger direct DRAM accesses for eviction-based Rowhammer [2], [10].

In the other case, the attacker does not learn the access latency of x, and victim accesses to x are needed for constructing eviction sets. It is the primary attack vector for randomized caches, as it represents the general scenario where x is not accessible by the attacker (unshared memory), *or* accessible to the attacker but decoupled in the cache for different security domains. In addition to the previous objectives, generalized eviction sets in this setting are useful, e.g., for PRIME+PROBE side- and covert channels, or to extend transient execution windows by evicting branch condition values from the cache.

An *arbitrary eviction set* (normally the easiest to construct [49]) is a set of memory addresses that, when accessed, has a high probability that at least one of its elements is evicted from the cache. Although this objective has proven to be useful in traditional caches, e.g., for covert channels [30], its generalization to randomized caches with $P > 1$ and security domains does not seem to map to any known adversarial goals.

Takeaway: Generalize eviction to avoid full congruence. Rely on partially congruent addresses to efficiently (but probabilistically) measure contention in randomized caches.

V. CONSTRUCTING GENERALIZED EVICTION SETS

The generalized eviction set G is the primitive at hand for attacking randomized caches. Once G has been constructed, contention-based attacks like PRIME+PROBE are also possible in randomized caches, although with a larger set and lower success probability (cf. Section IV-A). The major hurdle is the *profiling* attack stage, *i.e.*, constructing G itself. Purnal and Verbauwhede [37] performed an initial study of this problem.

This section is concerned with the construction of G for a target address x, using the capabilities of the black-box attack $\mathcal{A}_{ideal}$ (cf. Section III-B). We focus on the general case of a target x that is not attacker-accessible (cf. Section IV-B), as the security domain separator s lifts most attack objectives from the accessible to the non-accessible case. We will later show the optimizations that can be applied *should* they be accessible. Our novel profiling approach is generically applicable and efficient, improving state-of-the-art methods by orders of magnitude.

Conventionally, eviction sets are constructed by reducing a large set of addresses to a smaller set while maintaining a high eviction rate. This traditional *top-down* approach is highly effective for $P = 1$, but both the size of its initial set *and* its reduction step are strongly hindered by partitioning the cache ($P > 1$). We cope with the sheer infeasibility of reducing the initial set by adopting a new *bottom-up* approach: The attacker starts from an empty set and incrementally adds addresses for which cache contention with the target address was observed.

When measuring contention with a target x that is not attacker-accessible, the only available procedure is to prepare the cache state, wait for victim execution, and observe changes in the cache state. Finding a generalized eviction set G then comprises several iterations of this procedure. A successful iteration is one that *catches* an access to x, and the success probability of an iteration is the *catching probability* p_c.

Takeaway: Adopt a bottom-up strategy to construct G. In partitioned randomized caches, detecting contention is much more efficient than detecting *absence* of contention.

A. Generic Prime+Prune+Probe

To maximize the probability of catching a victim access to x in a given iteration, we develop a specialized PRIME+PROBE, tailored for finding eviction sets in randomized caches.

TABLE III: Catching probability p_c as a function of cache and attack instance, and whether the target address is cached or not.

RP	**Catching probability**	
	$p_{c,n}$ (not cached)	$p_{c,c}$ (cached)
RAND	$\frac{k'}{N}$	$\sum_{i=1}^{n_w} \binom{n_w}{i} \frac{i^2 \cdot k'^i \cdot (N-k')^{n_w-i}}{n_w^2 \cdot N^{n_w}}$
LRU	$\approx 1 - \texttt{binom}(k, \frac{n_w}{P}-1, \frac{1}{P \cdot 2^b})$	$\approx p_{c,n} \cdot \frac{p_{c,n}(P-1)+1}{P}$

1) Prime+Prune+Probe: An iteration begins with a **prime** step, where the attacker accesses a set of k addresses, loading them into the cache. For $k > 1$, there can be cache contention *within* this set. Thus, as a key step to eliminate false positives, the **prune** step iteratively re-accesses the set. This forces all self-evicted addresses to be cached again, at a potentially different location than before. The **prune** step terminates as soon as no more self-evictions occur when accessing the set.

If there are still self-evictions after a few iterations, pruning becomes more aggressive and additionally discards all addresses with high access latency (*i.e.*, those evicted by another attacker address). Upon termination of the **prune** step, the attacker has a set of $k' \leq k$ known addresses guaranteed to reside in the cache. Let m_{pr} denote the total number of pruning iterations.

Now, the attacker triggers the victim to perform the access of interest (*i.e.*, access x, as in conventional PRIME+PROBE). This memory access evicts one attacker address with probability p_c, which depends both on the attack parameter k and the randomized cache parameters (cf. Section V-B). In the **probe** step, the attacker accesses the set of k' addresses again, adding addresses with high latency to G (*i.e.*, victim evicted them).

In PRIME+PRUNE+PROBE (PPP), the **prune** step is crucial and noise-absorbing. Without it, the attacker cannot distinguish evictions due to victim accesses from those by the priming set. By pruning, the attacker completely removes these false positives. Appendix A experimentally relates pruning parameters k, k' and m_{pr} for different cache configurations.

The PRIME+PRUNE+PROBE procedure is repeated until enough accesses are *caught* and added to G. This constitutes the bottom-up approach; G is not the result of shrinking a large initial set. Instead, it is built from the ground up.

2) Penalty for being cached: In case the target address x is already cached, a single PPP iteration must both *evict* x and *catch* the access to x when it is reloaded into the cache.

The attacker can either (1) first evict x probabilistically, by accessing many different addresses or other techniques; (2) apply PPP as-is, tolerating a suboptimal catching probability p_c. These strategies trade off the success probability of one iteration (p_c) with its execution time (number of accesses). In what follows, we consider both a cached and uncached x. Any profiling strategy then has higher p_c than when the target is *always* cached, and lower p_c than when it is *never* cached.

B. Catching Probability p_c

The catching probability p_c is the success rate of one PRIME+PRUNE+PROBE iteration and depends on the randomized cache (n_w, b, P, policy RP) and attack parameter k'. Table III establishes p_c for several configurations. We distinguish whether x is cached (denoted $p_{c,c}$), vs. not cached (denoted $p_{c,n}$).

1) Target is not cached ($p_{c,n}$): After **prime** and **prune**, the victim access to x caches it in a random partition, and R_K pseudorandomly determines the cache set *within* this partition.

For RAND, x evicts an attacker address with probability equal to the coverage of the cache after pruning (*i.e.*, $p_{c,n} = k'/N$).

For LRU, x evicts an attacker address if there are at least $\frac{n_w}{P}$ addresses in the attacker set that were mapped to the same cache partition *and* set of x during **prime** and **prune**.

It can be approximated (and lower-bounded) by the complement of the cumulative binomial with k trials, $\frac{n_w}{P} - 1$ successes and binomial success probability $(P \cdot 2^b)^{-1}$, *i.e.*, $p_{c,n} = 1 - \texttt{binom}(k, \frac{n_w}{P}-1, \frac{1}{P \cdot 2^b})$. In practice, due to self-evictions during pruning, the actual number of binomial trials is slightly higher than k, resulting in increased $p_{c,n}$.

2) Target is cached ($p_{c,c}$): Catching an access to a cached target x requires *both* evicting x and detecting its reintroduction in the cache, resulting in a penalty on p_c. The probabilities $p_{c,c}$ (exact for RAND, approximate for LRU) are derived in Appendix B and collected in Table III. The penalty is maximal for $k' = 1$, being n_w (RAND) or P (LRU), and decreases with k' as **prime/prune** implicitly evict an increasing cache portion.

Appendix C complements the theoretical analysis with empirical validation. It also explores the relation between p_c and k', and the penalty on p_c for a cached target.

Takeaway: Add pruning to PRIME+PROBE profiling.
Pruning enables testing more than one guess per iteration. It improves profiling for RAND and is essential for LRU.

VI. OPTIMIZATIONS FOR PRIME+PRUNE+PROBE

This section describes optimizations of PRIME+PRUNE+PROBE for (A) total cache accesses and (B) victim invocations. We then evaluate PPP strategies on a range of cache instances.

A. Optimizing for total cache accesses

1) Burst Accesses: As derived, the catching probability $p_{c,c}$ (target already cached) holds at the start of constructing the generalized eviction set G. As the elements of G have explicitly been observed to collide with x, they can be accessed in burst before the PPP iteration, essentially implementing a *targeted* eviction of x. As profiling progresses and G grows, the burst becomes more successful, and the penalty for a cached target shrinks, hence $p_{c,c} \rightarrow p_{c,n}$ asymptotically. The burst access optimization thus hides the caching penalty. It applies to both RAND and LRU, but the latter can be accelerated even more.

2) Bootstrapping: A PPP iteration for LRU succeeds if **prime/prune** fill the full set for x in the designated partition. As G contends with x, we add G as bootstrapping elements to the PPP set. Thus, filling the full set becomes more likely.

However, if a victim access to x evicts a bootstrapping element instead of a PPP guess, the iteration is wasted: G was already known to contend with x. This issue can be resolved by relying on LRU statefulness. Adding G at the *end* of the PPP set ensures that PPP evictions precede bootstrapping evictions.

Bootstrapping implicitly implements burst accesses, and works very well for LRU. However, it is unattractive for RAND.

Takeaway: Use elements in G to accelerate finding more. *Burst accesses* hide caching penalty effectively as G grows. *Bootstrapping* increases p_c by helping to fill the LRU set.

B. Optimizing for victim invocations

We now explicitly minimize the required victim accesses A_v. This is relevant, e.g., for long victim programs or cases where victim runs are limited. We decouple it as $A_v = \frac{c}{p_c}$, relating it to accesses c needed to be *caught* (*i.e.*, successful iterations), and to p_c (*i.e.*, success probability of one iteration).

Section V already maximized denominator p_c with PRIME+PRUNE+PROBE. We now independently minimize numerator c, forming a flexible profiling framework to globally optimize A_v. It first preselects candidate addresses that have higher catching probabilities. The framework comprises three steps:

`Step 1.` Use PRIME+PRUNE+PROBE to find, for every partition i, *one* address a_i that collides with x in that partition.

`Step 2.` For each a_i, construct a candidate pool with addresses that collide with it in at least one partition.

`Step 3.` Resume PRIME+PRUNE+PROBE with the obtained candidate pools instead of randomly selected addresses.

The first step simply constructs a smaller G with PPP. Assume it needs to continue until G contains at least one element for every partition. The expected accesses to catch is then given by the coupon collector problem in statistics, with one set of P coupons: $E[c] = P(1 + 1/2 + \cdots + 1/P)$.

The second step finds addresses that contend with the a_i obtained in Step 1, instead of profiling x directly. As the a_i are attacker-accessible, their access latency can be measured, and no victim accesses are required. Addresses that contend with a_i also contend with x with probability $\geq P^{-1}$, which is much more likely than a randomly selected address ($\approx 2^{-b}$).

The third step resumes PPP for target x with candidate pools for the a_i. Every iteration accesses the pools, prunes, triggers access to x, and probes. For sufficiently large candidate pools, $p_c \approx 1$, significantly reducing A_v as compared to Step 1.

Conceptually, the first and third step are similar in nature. They can also be independently accelerated, as in Section VI-A.

We now explore the complexity and acceleration opportunities of Step 2. As the access latency of the *targets* a_i can be measured, catching probabilities can increase, and there is no penalty if the a_i are already cached. We again distinguish between replacement policies, and measure the complexity in attacker accesses A_a (as there are no victim accesses).

1) Optimizing Step 2 for RAND: We propose to construct the candidate pool through *reverse* PRIME+PRUNE+PROBE. Let S $= \{a_1, a_2, \ldots, a_c\}$ be the starting set obtained in Step 1. The elements of S are now the *targets* instead of the victim address x. Every iteration tries *one* random address guess g.

PRIME+PRUNE+PROBE (PPP) primes the cache with k guesses and observes eviction by the target. REVERSE PPP instead primes the cache with the *targets* S, prunes, accesses the *guess* g, then probes S. If accessing an element of S is slow, say a_k, we add g to the candidate pool for a_k. Every iteration has $p_c = \frac{c}{N}$, and there are $\approx c + 1$ attacker accesses per iteration, (*i.e.*, very little pruning, and **`probe`** overlaps with the next **`prime`**). The expected number of attacker accesses to obtain one element for the candidate pool hence is $E[A_a] \approx N$.

2) Optimizing Step 2 for LRU: For LRU, reverse PPP is even more effective. Again, let S $= \{a_1, a_2, \ldots, a_c\}$ be the set from Step 1, and let g denote a random address guess.

Assume the attacker primes the cache with S, prunes it, and observes self-evictions. For LRU, this implies that S filled a full cache set ($\frac{n_w}{P}$ lines). In this case, the attacker does reverse PRIME+PRUNE+PROBE, where one iteration consists of **`prime`** and **`prune`** with S, accessing g, and **`probe`** with S. If accessing an element of S is slow in the **`probe`** step, say a_k, we add g to the candidate pool for a_k. This approach has $p_c = \frac{1}{P \cdot 2^b}$, and there are $\approx c{+}1$ accesses per iteration, resulting in expected number of attacker accesses $E[A_a] \approx (c{+}1) \cdot P \cdot 2^b$.

Importantly, as g collides with multiple a_k in S, it very likely collides with x and can directly be added to S. Thus, it immediately grows eviction set G without accesses by the victim, bypassing Step 3. However, it can only be started if priming S has observed self-evictions. Interleaving it with Step 1 implicitly generates new attempts at this precondition.

3) Flexibility of the Framework: The three-step framework flexibly instantiates randomized caches and attack scenarios. If the victim program is tiny and executes continuously, all profiling time is spent in Step 1. The shares of Step 2 and Step 3 grow as soon as the victim program becomes the bottleneck in any way. Finally, if x is attacker-accessible, reverse PPP from Step 2 is used immediately. The framework also enables splitting G based on the partition of contention with x, making the eviction probabilities (Section IV-A) exact.

Takeaway: Use elements in G to reduce victim accesses. Filtering candidate addresses based on contention with G allows to (partially) refrain from victim invocations.

C. Evaluation of profiling strategies

Figure 4 depicts victim and total cache accesses for the presented profiling strategies, obtained from simulated profiling runs (cf. Section III-A3). We observe a mostly linear progression in constructing G. One exception is reverse PPP, where the construction of the candidate pools does not grow G immediately (jump), but accelerates the profiling that follows.

Optimizations like burst accesses and bootstrapping improve both total and victim accesses. In contrast, probabilistic full cache evictions and three-step profiling incur a trade-off between total accesses and victim invocations. Of course, one can freely interpolate between these extreme strategies.

D. Influence of randomized cache instance

1) Sets, ways and partitions: Both profiling and exploitation in randomized caches are influenced by the parameters of the instance. We investigate the effectiveness of PPP on several

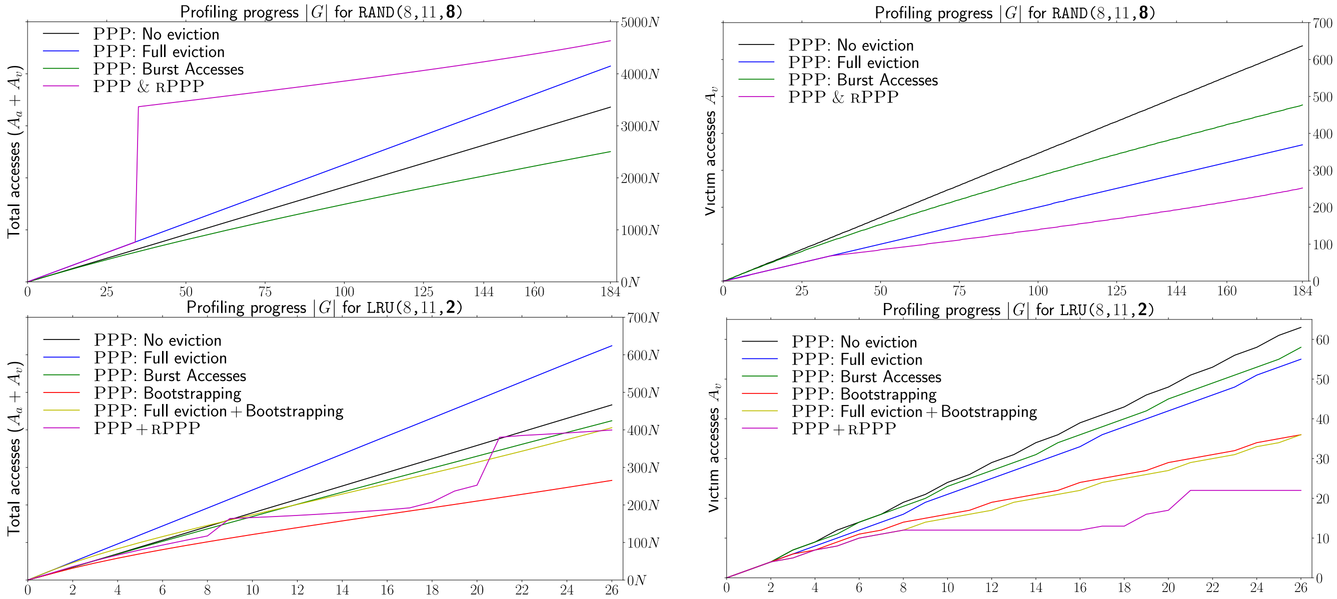


Fig. 4: Effort of profiling strategies for RAND **(top)** and LRU **(bottom)**, measured as total **(left)** and victim **(right)** cache accesses, averaged for 10^4 simulated profiling runs. Cache instances are denoted $\mathtt{RP}(n_w, b, \boldsymbol{P})$. k is fixed to $\frac{N}{2}$ (RAND) and $\frac{3N}{4}$ (LRU) to isolate the influence of the strategy. Pruning becomes aggressive from the sixth iteration, if not already terminated. Full evictions between PPP iterations, if performed, use $2N$ addresses for LRU and $3N$ for RAND.

instances for RAND and LRU. Figure 5 captures our findings, again based on simulation (cf. Section III-A3).

Larger caches resist better against PPP. Increasing cache ways (n_w) seems to compare favorably to increasing sets (2^b). While the latter only proportionally prolongs profiling and does not affect exploitation, the former inhibits both profiling and exploitation. In particular, $|G|$ increases for the same exploitation p_e, and profiling is prolonged as $|G|$ increases while the accesses per element of G stay roughly the same.

Similarly, for the same cache dimensions (n_w, b), both PPP profiling and exploitation suffer from increased partitioning P. Especially for RAND, there is no indication from our ideal-case analysis why one should not opt for maximal partitioning.

In general, we find that PPP can be hindered by tuning cache sets, ways, and partitions, but not to the point where it becomes infeasible. What really works is limiting the cache access budget for the attacker (*i.e.*, a strict rekeying condition).

2) Rekeying period: The difference between the profiling state of the art and rekeying period T is the design's security margin. Although tempting, setting T just low enough to thwart known techniques does not account for potential improvements.

As an example to obtain (very) conservative rekeying periods, we now leverage the security of R_K to derive minimal complexities to construct generalized eviction sets *of certain quality*, *i.e.*, with a lower bound on eviction probability p_e (e.g., $p_e \geq 90\%$). We use the following central assumptions:

🅐 R_K is indistinguishable from a random function.
🅑 Victim addresses of interest are not attacker-accessible.
🅒 The eviction probability p_e for G is lower-bounded.

As the target is not accessible to the attacker (🅑), she can only infer accesses with PPP (cf. Section V-A1): bring cache in known state, wait for victim execution, and probe.

To achieve an eviction rate p_e (🅒), the profiling needs to catch at the very least $m \geq p_e P$ victim accesses to different partitions. Indeed, an attack with $p_e > \frac{m}{P}$ has inferred information about partitions for which no memory access has been caught. By contradiction with 🅐, it cannot exist.

TABLE IV: Rekeying periods T to ensure that the success rate to construct G with $p_e \geq 95\%$ is upper-bounded by $1/2^{\{8,12,16,24,32\}}$. The cache instance is RAND(16, 13, **16**) and all accesses are counted as cache hits (e.g., 10 ns)

	T for profiling		$T/2$ for profiling	
Success Rate	T	time	T	time
2^{-8}	$40N$	$\approx$ 10 sec	$80N$	$\approx$ 20 sec
2^{-12}	$29N$	$\approx$ 2.5 min	$58N$	$\approx$ 5 min
2^{-16}	$22N$	$\approx$ 30 min	$44N$	$\approx$ 60 min
2^{-32}	$9N$	$\approx$ 2 years	$18N$	$\approx$ 4 years

Beyond $\mathcal{A}_{ideal}$, we further contrive the setting in favor of the attacker. We consider strongly idealized pruning (*i.e.*, $k' = k$ and $m_{pr} = 1$), and a permanently uncached target (*i.e.*, $p_c = p_{c,n}$). Furthermore, we scope the algorithm as catching a single access in m partitions, neglecting the necessary expansion to full G.

A perfectly ideal PRIME+PRUNE+PROBE iteration then requires k accesses for **prime**, k for **prune**, 1 for the victim access, and k to **probe**. Assuming the attacker somehow manages to combine **probe** of one iteration with **prime** of the next, we use $2k+1$ accesses per iteration as lower bound.

We outline the idea for a randomized cache with random replacement. The only degree of freedom in the idealized PPP is the number of addresses k in the **prime** step. Indeed, their order or frequency does not impact the cache coverage $\frac{k}{N}$.

Given a rekeying period of T cache accesses, the probability of observing at least one access in at least m distinct partitions is (using a generalization of the birthday problem in statistics):

$$\max_k \left[\sum_{i=m}^{\frac{T}{2k+1}} \binom{\frac{T}{2k+1}}{i} \frac{k^i (N-k)^{\frac{T}{2k+1}-i}}{N^{\frac{T}{2k+1}}} \sum_{l=m}^{P} \binom{P}{l} \sum_{r=0}^{l} (-1)^r \binom{l}{r} \left(\frac{l-r}{P}\right)^i \right]$$

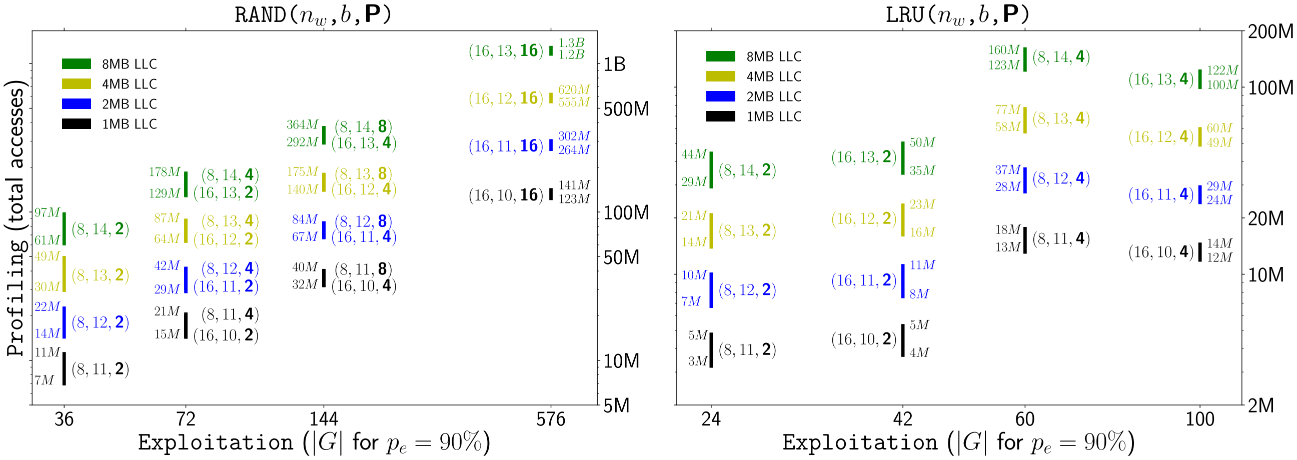


Fig. 5: Influence of randomized cache parameters, for RAND **(left)** and LRU **(right)**. To isolate the influence of the instance, profiling strategies are fixed to burst accesses and $k = \frac{N}{2}$ for RAND, and bootstrapping and $k = \frac{3N}{4}$ for LRU. Instances are indicated as $(n_w, b, \boldsymbol{P})$, and positioned for mean profiling effort (y-axis, log scale), and eviction set size for exploitation (x-axis, log scale). Vertical lines span the 5-95th percentiles (ranges indicated) over 10^3 simulated runs.

Conversely, Table IV captures rekeying periods T that upper bound the fraction of successful rekeying periods. Pessimistically assuming that every memory access is a cache hit, it gives an expected continuous profiling time of having *one* successful construction of G within the rekeying period. As the obtained G is only useful for one period, Table IV also includes the case where half of it is used for exploitation. Note that these minimal efforts strongly depend on m, and hence on the quality of G that can be tolerated for exploitation (🅒).

VII. Lifting Idealizing Assumptions

In this section, we explore for the first time the more challenging attack $\mathcal{A}_{nonid}$ with complicating system activity (cf. Section IV), as opposed to the commonly assumed $\mathcal{A}_{ideal}$. We start with a victim program performing more memory accesses than of interest to the attacker and present an end-to-end attack on a vulnerable AES implementation. We then quantify the influence of random noise on Prime+Prune+Probe.

The central assumption is that the attacker wants to profile specific addresses of the victim and that the access probability of said addresses can be changed via inputs to the victim.

A. Multiple Victim Accesses

In the profiling phase, the attacker identifies addresses of interest in a victim program and distinguishes between them if there are multiple, requiring disjoint eviction sets for each target. From this perspective, we model the execution of victim code as a set of *static* and *dynamic* memory accesses. Static accesses are performed regardless of the attacker's input, *i.e.*, code and data accesses performed in all victim executions. Dynamic accesses do not always occur, e.g., state- or input-dependent code or data accesses.

The attack targets are one or more addresses that are accessed upon a certain *event* the attacker wants to spy on [12]. Like Gruss et al. [12], we cannot distinguish addresses in the static set, as the cause-effect relationship is the same for all of them. Hence, for our attack, all targets are in the dynamic set.

To profile the cache addresses of interest, we propose a two-phase approach. First, we collect a superset of addresses containing colliding addresses for all static and dynamic cache lines. Second, we obtain disjoint sets of addresses from the superset, each with colliding addresses for one target.

The attacker distinguishes static and dynamic accesses by the property that dynamic accesses are statistically performed *less often* than static accesses, which are *always* performed. With the assumption from the beginning of this section, we consider a scenario where an attacker controls, e.g., via input, which dynamic accesses the victim performs in any given execution. This control can be exerted positively (*i.e.*, a dynamic access is always performed for a specific input) as well as negatively (*i.e.*, a dynamic access is never performed for a specific input). The latter scenario repeatedly calls the victim with inputs that cause it to access all but one address. Thus, it can be separated from the superset, as all other addresses in it are accessed eventually. In general, any manipulation of access probabilities in the victim can be observed. This approach describes a stronger attacker, as targeted addresses can be distinguished from others in both the dynamic and static set in the same step.

1) Implementation: In the following, we focus on maximum partitioning $P = n_w$, as non-random replacement policies like *LRU* generally require special treatment but behave predictably. We employ catching with intermediate full eviction. The analysis of Section V-B1 applies. To generate distinct and large eviction sets for our n_{target} target addresses, we slightly modify the three-step approach described in Section VI-B. All experiments are obtained in simulation (cf. Section III-A3).

To find sets of addresses a_i, in `Step 1`, we first construct the previously described superset using Prime+Prune+Probe (Section VI-B). Instead of only one victim memory access, all n_{stat} static and n_{dyn} dynamic victim accesses are now observed by the attacker. To identify a enough colliding addresses for all targets, we construct a superset of at least $n_w \cdot (n_{stat} + n_{dyn})$ addresses. The expected amount of memory required to find a collision in a specific

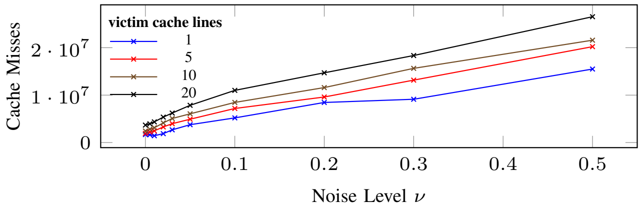


Fig. 6: Cache misses for creating a superset with $3 \cdot n_w$ addresses per victim cache line, as a function of noise ν for different numbers of total victim cache lines, $k = 1000$ (avg. over 100 runs). Instance is $\texttt{RAND}(8, 9, \mathbf{8})$

TABLE V: End-to-end attack on T-table AES for different configurations (means over 100 runs). $n_{slices} = 8$, $b = 11$. Where not shown, standard deviations are $< 0.5\%$ of the mean.

n_w	P	policy	misses [10^9]	hits [10^9]	#AES	∅ collisions/addr.	correct nibbles	est. t [min]
8	8	n/a	12.03	3.59	56663	20.47 ± 3.61	15.90 ± 0.33	1.58
8	2	RAND	2.78	2.25	23682	15.52 ± 3.37	16.00 ± 0.00	0.63
8	2	LRU	3.21	1.75	26060	17.74 ± 7.53	15.94 ± 0.28	0.78
16	16	n/a	46.27	9.25	157072	37.91 ± 5.65	15.77 ± 0.45	6.89
16	2	RAND	4.69	3.91	39192	26.62 ± 5.85	15.93 ± 0.26	1.32
16	2	LRU	7.85	2.63	66640	26.99 ± 11.66	15.75 ± 0.51	2.60

way is $\frac{cachesize}{n_w}$, though higher confidence requires more. We can apply the coupon collector's problem (cf. Section VI-B) for an estimated factor of $\frac{coupon(n_w)}{n_w}$, but as more addresses need to be profiled, the probability to catch enough addresses for all targets decreases. Consequently, this step requires a number of repetitions, depending on the **`prime`** parameter k.

Next, we separate unwanted addresses from target addresses within the superset. To this end, we call the victim with inputs that exclude exactly one of the n_{target} cache lines. By repeatedly evicting the cache, calling the victim with the required parameters, and measuring accesses in the superset, we generate a histogram for all target addresses. After a certain number of repetitions, addresses that are *never* evicted by the victim are very likely to collide with the targeted address.

Repeating this process n_{target} times, we get disjoint sets of addresses for each target cache line. `Step 2` and `Step 3` can be applied to these sets of addresses (a_i) to construct the final generalized eviction sets like in the single-access case.

From our experiments (cf. Figure 6), we estimate that the number of cache misses (the largest factor of the execution time) increases sub-linearly with the total amount of accesses by the victim $(n_{stat} + n_{dyn})$. This is because the catching probability p_c increases with n_{target}. The superset's separation depends linearly on n_{target} and the overall size of the superset.

2) End-to-end Attack on AES T-Tables: We choose the 10 round T-tables implementation of AES in OpenSSL 1.1.0g as an example, as it is a well-known target for cache attacks [3], [34], [44], [12]. We perform the One-Round Attack, described by Osvik et al. [34], and thus recover 64 bits in the 16 upper nibbles of the 16-byte key (see Appendix D).

The parameters for this attack are $n_{stat} = 27$ and $n_{dyn} = 65$. With $n_{target} = 64$, the 4 T-tables are a difficult attack target, as the profiling time scales linearly with n_{target}.

For profiling, we require AES runs that access all but the target address, for each target. We can prepare 64 such key/plaintext pairs offline. All AES runs are recorded as memory access traces with the Intel PIN Tool [15] and injected into the simulator (cf. Section III-A3) at the appropriate times. Lacking more efficient eviction methods, we rely on probabilistic full cache eviction. In total, eviction accounts for $\approx 90\%$ of all accesses during the attack, which in turn makes the superset-splitting step of the profiling the largest contributor to the overall runtime. Because we assume no restriction on the number of encryptions, we do not perform `Step 2` for this attack, as pruning the generated candidate pools would also require the costly splitting phase. Instead, we see that using fewer colliding addresses for each target (cf. Table V) still performs well. We can compensate for the lower detection probability by increasing the number of encryptions during the exploitation phase.

We use cache parameters from modern Intel processors: 8 slices (with a slicing function [28]) of 1 MB each, so each slice is a randomized cache with $n_w = 8/16$ and $b = 11$. We run the same attack for $P = n_w = 8/16$ and $P = 2$, with replacement policies random and `LRU`. As seen in Table V, the attack is generic enough for all configurations, without special considerations for `LRU`. The variance in the number of addresses found per target increases for $P = 2$, especially for `LRU`, but since this specific attack sums over the hits on different addresses, this effect is mitigated for the end result (see Appendix D). For $P = 2$, we speed up the attack by reducing the cache lines used for full cache eviction from $2N$ to $1.5N$, as well as reducing the superset size (cf. Section VI-D1).

This end-to-end implementation is not optimal, as there are many parameters that could be optimized. Nonetheless, we can see that cache attacks can still be executed in a reasonable time frame. If we model the attack as a mixture of sequential accesses for full cache evictions and timed random accesses for the sets, we can calculate the average attack times shown in Table V. For this rough estimate, we use access times measured on a real system with the same miss rates (i7-8700K @ 3.60GHz, sequential access: ≈ 11.4c, timed (`rdtsc`) random miss: ≈ 235c, hit: ≈ 222c).

Takeaway: Unpredictability requires key agility.
Frequent rekeying is essential to maintain the benefits of randomization, even in non-ideal conditions.

B. Influence of Noise

In the ideal case ($\mathcal{A}_{ideal}$), there is no noise from memory accesses by the attacker process itself, nor the victim, or any other process in the system (including the operating system). Section VII-A already implicitly includes noise generated by the victim's code execution. We now additionally consider noise introduced by other system activity. We make the simplifying assumption that noise accesses are random and occur at a rate of ν random accesses for every attacker access.

Multiple steps of the (unmodified) profiling algorithm from Section VI-B are affected by noise. Spurious memory accesses during the **`prune`** step increase the number of pruning iterations m_{pr} significantly and reduce the size k' of the

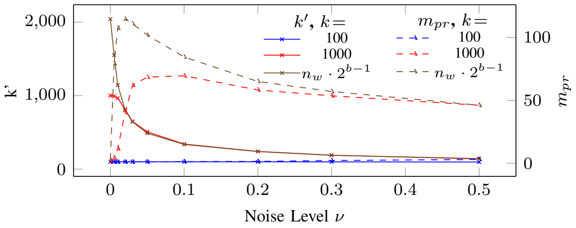


Fig. 7: Pruning m_{pr} and k' as a function of noise ν, for various k (average over 100 runs). Instance is RAND $(8, 9, \mathbf{8})$

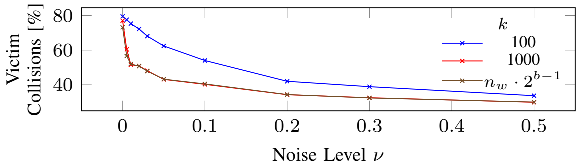


Fig. 8: Percentage of caught addresses in the superset that genuinely collide with victim addresses in exactly one way, as a function of ν for various k (avg. over 100 runs). Instance is RAND $(8, 9, \mathbf{8})$

resulting set. The **probe** step samples noise in addition to the collisions with the targeted victim cache line.

Figure 7 and Figure 8 show both effects. Though the unmodified **prune** step terminates, the resulting set size k' can be seen to decrease quickly with ν, while the number of pruning iterations m_{pr} increases. Hence, with noise, the attacker could explore the PPP parameter space in favor of a smaller k. Figure 8 also shows a faster decrease in correct collisions for higher k, while the cost of pruning grows.

Alternatively, the attacker can consider *early-aborting pruning*, *i.e.*, terminating **prune** before it is *entirely* free of misses. Indeed, a large part of the pruning iterations are no longer due to self-evictions, but due to sampling noise. The false positives introduced by the early-abort are then removed in a later stage.

The separation phase from Section VII-A is effective at filtering false positives caused by noise during PPP, since static victim accesses, dynamic victim accesses, and false positives exhibit different behavior in the separation phase. In contrast to static or dynamic accesses, false positives occur only in some runs, leading to multiple runs with 0 accesses. Hence, they appear in more than one set in the end and can be removed.

Figure 6 shows the total number of cache misses for the generation of supersets for victims of different total sizes $(n_{stat}+n_{dyn})$. These supersets contain exactly $3 \cdot n_w \cdot (n_{stat}+n_{dyn})$ addresses that collide with victim cache lines in exactly one way. They additionally contain non-colliding addresses introduced by noise and self-eviction in the proportion shown in Figure 8, which is removed during separation. We can see that the number of cache misses (and by extension, the runtime) for this step grows approximately linearly with noise.

C. Infrequent victim events

In the case where an event in the victim happens only *once* or a limited number of times (e.g., user input), the probe set G needs to be large enough to achieve a very high detection probability, which places more weight on accurate profiling compared to VII-A2. On the other hand, when events trigger accesses to multiple cache lines, all of them can be used for detection. Attacks will mostly need to be asynchronous, which necessitates some form of continuous monitoring. We leave an investigation of practical implementations for future work.

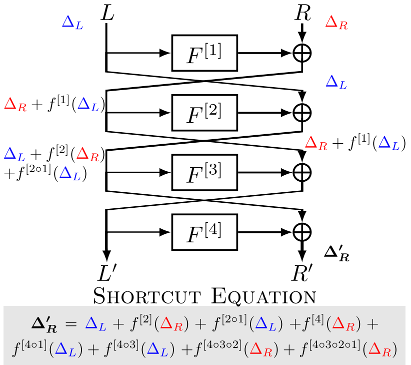

Δ_L
L
R
Δ_R
$F^{[1]}$
Δ_L
$\Delta_R + f^{[1]}(\Delta_L)$
$F^{[2]}$
$\Delta_L + f^{[2]}(\Delta_R)$
$+f^{[2\circ1]}(\Delta_L)$
$\Delta_R + f^{[1]}(\Delta_L)$
$F^{[3]}$
$F^{[4]}$
$\boldsymbol{\Delta'_R}$
L'
R'
Shortcut Equation
$\boldsymbol{\Delta'_R} = \Delta_L + f^{[2]}(\Delta_R) + f^{[2\circ1]}(\Delta_L) + f^{[4]}(\Delta_R) +$
$f^{[4\circ1]}(\Delta_L) + f^{[4\circ3]}(\Delta_L) + f^{[4\circ3\circ2]}(\Delta_R) + f^{[4\circ3\circ2\circ1]}(\Delta_R)$

Fig. 9: Differential propagation through CEASER's LLBC. For brevity, we introduce $f^{[j\circ i]}(\cdot)$ as shorthand for $f^{[j]}(f^{[i]}(\cdot))$.

VIII. Shortcut Attacks

In this section, we consider attack $\mathcal{A}_{short}$ and draw attention to the soundness of the randomized mapping by achieving *shortcuts* during a case study on CEASER and CEASER-S.

In particular, we demonstrate how weaknesses in their common randomized mapping allow us to reliably construct eviction sets without any memory accesses. We first describe Low-Latency Block Cipher (LLBC), the CEASER-specific implementation of the mapping R_K. Drawing inspiration from differential cryptanalysis, we show how input differences propagate through the LLBC, and we derive an expression for precomputing address differences that systematically yield cache set collisions, *independent of key, partition, and address*.

We describe the attack first for CEASER [38] before tackling the generalized and improved CEASER-S [39].

A. Low-Latency Block Cipher in CEASER(-S)

CEASER instantiates R_K by encrypting the input address a with a custom LLBC with 40-bit blocks and 80-bit key. In particular, it divides the input address in two equally sized (left-right) chunks $a = (\mathtt{L} \mid\mid \mathtt{R})$ and produces an output encrypted address $R_K(a) = (\mathtt{L'} \mid\mid \mathtt{R'})$. From this output, the lowermost b bits determine the cache set index: $s = \lfloor R_K(a) \rfloor_b = \lfloor \mathtt{R'} \rfloor_b$.

The encryption proceeds as a keyed four-stage Feistel network (depicted in Figure 9). Each stage instantiates a round function $F(X, K)$, taking 40-bit input (20 bit X and 20 bit K) and producing 20-bit output (Y). In each round function, 20 intermediary bits W_i are first computed as $W_i = S_i(X, K)$, where S_i defines exclusive or ($\mathtt{xor}$) of 20 input bits (out of 40). The W_i are shifted with a bit-permutation P to obtain Y.

In CEASER, the round functions are randomly sampled, fixed at design time, and explicitly different in every stage. Let $F^{[r]}$ denote the round function for stage r, and $K^{[r]}$ the 20-bit subkey for this stage. Describe the bit-permutation with $i \leftarrow P(i)$, *i.e.*, a bit at position $P(i)$ moves to position i. Next, let $\mathcal{X}_i$ and $\mathcal{K}_i$ denote the indices from resp. X and $K^{[r]}$

that are xored to obtain intermediary bit $W_i = S_i(X, K)$. The round function output is $Y = (Y_0||Y_1||...||Y_{19}) = (W_{P(0)}||W_{P(1)}||...||W_{P(19)})$. The round function $F^{[r]}$ thus comprises 20 functions $F_i^{[r]}(X, K^{[r]})$ each computing one Y_i:

$$Y_i = F_i^{[r]}(X, K^{[r]}) = \sum_{j \in \mathcal{X}_{P(i)}} X_j + \sum_{k \in \mathcal{K}_{P(i)}} K_k^{[r]} \tag{1}$$

Observing the linearity in the entire cipher (particularly in the SBoxes S_i, supposed to be non-linear), we draw inspiration from differential cryptanalysis to bypass R_K altogether.

B. Constructing and Using the Shortcut

The outcome of the shortcut is a set of addresses a_i that collides in the cache with a target address a, *i.e.*, $R_K(a_i) = R_K(a)$. The attacker could attempt this shortcut by recovering the mapping key K, granting the shortcut for the lifetime of the key. Our approach, in contrast, is fully key-independent. It is a restricted take on *chosen-plaintext attacks*, where the restriction stems from being embedded in a cache. Specifically, the adversary can *choose* a set of plaintexts to R_K (*i.e.*, input addresses a_i), but does not observe any cryptographic output.

We rephrase the shortcut as a differential problem, *i.e.*, to finding a set of Δa satisfying $R_K(a+\Delta a) = R_K(a)$. Matching with the Feistel topology, we denote the input difference $\Delta a = (\Delta_L||\Delta_R)$ and the output difference $(\Delta_{L'}||\Delta_{R'})$. Achieving the shortcut is then equivalent to finding pairs Δ_L and Δ_R, not both zero, that result in the same set index bits: $\lfloor \Delta_{R'} \rfloor_b = 0^b$.

1) $\Delta-$Propagation: We derive the propagation first through the round function $F^{[r]}$, then the full LLBC. Let $+$ denote *GF*(2) addition (bitwise xor). As a well-known cryptanalytic fact, differences propagate unaffected through addition. Let Δ_X and Δ_Y denote differences at the input and output of $F^{[r]}$. Stated differently, if $Y = F^{[r]}(X, K^{[r]})$ and $Y' = F^{[r]}(X+\Delta_X, K^{[r]})$, then $\Delta_Y = Y'+Y$. Now compute the i-th output bit $\Delta_{Y,i}$:

$$\begin{aligned}\Delta_{Y,i} = Y_i' + Y_i &= F_i^{[r]}(X + \Delta_X, K^{[r]}) + F_i^{[r]}(X, K^{[r]}) \\ &= \sum_{j \in \mathcal{X}_{P(i)}} (X_j + \Delta_{X,j} + X_j) + \sum_{k \in \mathcal{K}_{P(i)}} (K_k^{[r]} + K_k^{[r]}) \\ &= \sum_{j \in \mathcal{X}_{P(i)}} \Delta_{X,j} = f_i^{[r]}(\Delta_X)\end{aligned}$$

If we let $\Delta_Y = f^{[r]}(\Delta_X)$, then $f^{[r]}$ captures the effect of round function $F^{[r]}$ on an input difference Δ_X. Similar to $F^{[r]}$ before, $f^{[r]}$ is an umbrella for 20 functions:

$$\Delta_Y = f^{[r]}(\Delta_X) = (f_0^{[r]}(\Delta_X) \; || \; f_1^{[r]}(\Delta_X) \; || \; ... \; || \; f_{19}^{[r]}(\Delta_X))$$

Note that $f^{[r]}$ only depends on the input difference Δ_X. Crucially, it is *independent of both X itself and the key K.*

2) Shortcut Equation: Armed with the Δ-propagation through round functions $F^{[r]}$, Figure 9 shows our probability 1 differential trail through CEASER's full LLBC, yielding an expression for output difference Δ_R'. This expression, which we dub the SHORTCUT EQUATION, describes $\Delta a = (\Delta_L||\Delta_R)$ satisfying output collision: $\lfloor \Delta_R' \rfloor_b = 0^b \Rightarrow R_K(a) = R_K(a+\Delta a)$.

A straightforward way to find solutions to this equation fixes (say) Δ_L and tests variable Δ_R for equality. The expected *offline* complexity for each $\Delta a = (\Delta_L, \Delta_R)$ is 2^{b-1} evaluations of the shortcut equation. Since very often $b < 20$, the naïve computation is very practical. As the shortcut equation describes twenty linear equations over *GF*(2), one could also algebraically determine a compact expression for Δ_L and Δ_R.

3) Implications: The shortcut does not require *knowledge* of key K, and is even completely *independent* of K. Furthermore, it is also independent of the input a. Although in general $R_K(a) \neq R_{K'}(a)$, eviction sets constructed for key K' and input a' are still eviction sets for *any other* (K, a) pair:

$$R_K(a) = R_K(a + \Delta a) \Rightarrow R_{K'}(a' + \Delta a) = R_{K'}(a') \tag{2}$$

This follows from the key-independence of the SHORTCUT EQUATION. Hence, rekeying does not invalidate eviction sets constructed using the shortcut. This has the devastating consequence that, as soon as the Δa have been precomputed offline, the attacker can construct arbitrary eviction sets for any target a with zero cache accesses, completely bypassing R_K.

4) Extension to CEASER-S*:* CEASER-S implements partitions with P parallel LLBC instances with different keys. By Equation (2), collision in one partition implies collision in all partitions. Thus, our shortcut equally impacts CEASER-S, allowing easy construction of *fully congruent* eviction sets.

5) Mitigation: At the very least, the LLBC rounds should incorporate non-linear SBox layers. This spot mitigation thwarts the presented shortcut, but more subtle attacks could remain.

Takeaway: Do not overestimate the mapping's security. Shortcut attacks can be fundamentally eliminated by a randomization mapping that resists formal cryptanalysis.

IX. DISCUSSION

In this section, we relate and compare the contributions in this paper to the most closely related work, as well as provide specific recommendations and directions for future work.

A. Prime+Prune+Probe on specific designs

Our generic model for computation-based randomized caches permits to instantiate existing designs, extend their security analysis, and compare them in terms of profiling effort.

We consider an 8 MB cache with 16 ways (n_w) and 13 index bits (b) (*i.e.*, $N = 131\,072$). We assume a non-accessible target address (e.g., by enabling security domain separation s). Although we consider $\mathcal{A}_{ideal}$ (cf. Section III-B), *i.e.*, we are able to pinpoint one target access of interest, victim execution time cannot be neglected. Therefore, we assume a modestly-sized victim program, performing 1 000 accesses per invocation.

Figure 10 shows total cache accesses to profile a generalized eviction set G with $p_e = 90\%$. For each instance we use PRIME+PRUNE+PROBE and optimize for total cache accesses.

1) Single-partition caches: Randomized caches with $P = 1$ (CEASER, TSC) can be treated as traditional caches without adversary control over physical addresses. They require extremely frequent rekeying, given that fully congruent eviction sets (*i.e.*, $p_e = 100\%$) can be obtained with the efficient top-down profiling approach [49], [39].

2) CEASER-S*:* First-party CEASER-S analysis [39] only considers fully congruent eviction. As fully congruent addresses are extremely scarce, it is completely infeasible for larger P.

We instantiate the model to CEASER-S2 (resp. CEASER-S4) by setting $P = 2$ (resp. $P = 4$) and replacement policy

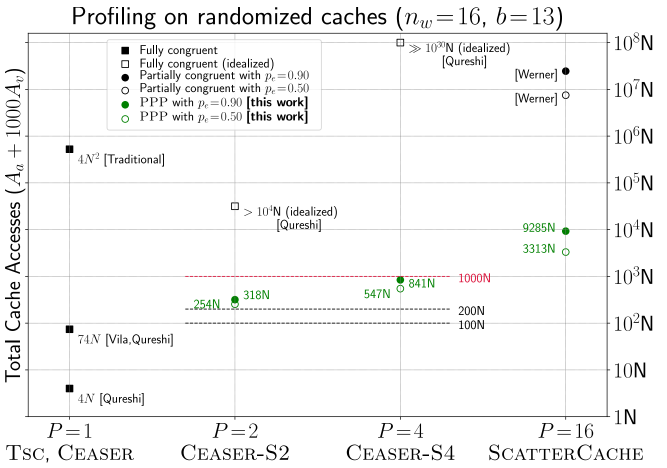


Fig. 10: Complexity ($\mathcal{A}_{ideal}$) to construct a fully congruent or generalized ($p_e = 90\%$ or $p_e = 50\%$) eviction set. Randomized caches are monolithic 8 MB ($n_w = 16$, $b = 13$, $N = 131\,072$ lines). Cost metric is total cache accesses; victim runtime is modeled with 1 000 accesses. Fully congruent eviction assumes initial set (before reduction) of $2N$. PPP uses the best-performing strategy (cf. Section VI-C), with $k = \frac{N}{2}$ (RAND) or $k = \frac{3N}{4}$ (LRU).

LRU. While CEASER-S could accomodate several policies (e.g., LRU, RRIP, ...) [39], we believe LRU leads to an accurate security assessment. Indeed, many stateful replacement policies can be degraded to LRU with some repeated accesses [10].

In what follows, we assume the problems from Section VIII to be fixed. There are three proposed CEASER-S instances, with rekeying periods resp. $100N$, $200N$ and $1000N$. We observe that PRIME+PRUNE+PROBE consistently obtains high-quality generalized eviction sets within the rekeying period of the $1000N$-instance. While prior profiling techniques succeed on average once every 68 years [39], PPP on CEASER-S2 has average complexity of $\approx 320N$, leaving on average 68% of *every* rekeying period available for exploitation. The more conservative designs ($100N, 200N$) resist PPP for the majority of rekeying periods, though with considerably reduced security margin. We observe an extreme gap between PPP and previous idealized estimates, easily exceeding 20 orders of magnitude for $P = 4$ and 50 orders of magnitude for $P = 8$ (not displayed).

3) SCATTERCACHE: First-party analysis [53] already considers generalized eviction. Their approach can be seen as a corner case of PPP, *i.e.*, using $k = 1$ (cf. Section V).

We instantiate SCATTERCACHE by setting $P = n_w = 16$, implicitly with replacement RAND. Optimized for total accesses, PPP improves profiling with three orders of magnitude for the considered configuration. The main contribution of PPP is that it requires much fewer victim invocations, as it permits to test many addresses in parallel ($k \gg 1$). While SCATTERCACHE does not specify a rekeying frequency, our results indicate that it should be determined more conservatively than expected.

4) Shortcuts: With a case study on CEASER-S, we show with devastating consequences that the security of the randomization should not be taken for granted, even if its output is not directly observable. A similar study was conducted in concurrent work [5]. Instantiating a sound cryptographic algorithm thwarts all shortcuts but affects performance. Though not investigated, TSC risks shortcuts due to absence of cryptographic structure. Shortcuts in SCATTERCACHE are only possible by significant cryptanalytic advances for QARMA [1].

B. Future Work

Our work serves as a baseline to which future secure cache designs, and their analysis, could be compared. This paper also shows the importance of cryptanalytic resistance of the core randomization mapping. Stringent latency constraints could inspire new designs in the space of low-latency cryptography.

The rekeying period may be varied for different security levels. This can be transparently implemented through frequently and unpredictably updating s for high-security processes (e.g., enclaves), while refreshing K in larger intervals for regular processes. We also propose heuristic-based rekeying, invalidating eviction sets upon observation of certain microarchitectural events (e.g., many LLC cache misses or PPP signatures). It should be noted that rapid rekeying only mitigates attacks in scope for randomized caches, *i.e.*, potential cache-contention channels that do not target set contention might remain.

The gap between our conservative rekeying periods (Section VI-D2) and PPP profiling in practice is quite large. Future work could explore closing this gap by improving profiling, relaxing theoretical bounds, or a combination of both.

X. CONCLUSION

Analyzing the residual attack surface of randomized cache architectures is a complex undertaking. In this work, we have established a generic framework to jointly analyze all existing computation-based randomized caches. We showed that, similar to cryptanalysis, randomized cache designs must be subjected to systematic analysis to gain confidence in their security. In this effort, we have contributed on three main fronts.

First, we have advanced the profiling state of the art for randomization-based secure caches. We developed novel attack techniques for such caches, including PRIME+PRUNE+PROBE and optimizations like bootstrapping and multi-step profiling.

Second, we have started bridging the gap between the usually assumed ideal attack and complicating effects like noise and multiple victim accesses. We have simulated an end-to-end attack, leaking AES keys from a vulnerable implementation.

Finally, we have falsified the implicit assumption that any randomized mapping successfully results in a secure cache.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers and our shepherd, David Kohlbrenner, for their valuable feedback. This work was supported in part by the European Research Council (ERC) under the EU Horizon 2020 research and innovation programme (grant agreements No 681402 and No 695305). It was also supported by the CyberSecurity Research Flanders VR20192203 and the Research Council KU Leuven C16/15/058. Antoon Purnal is funded by an FWO fellowship. Additional funding was provided by a generous gift from 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] R. Avanzi, "The QARMA block cipher family. almost MDS matrices over rings with zero divisors, nearly symmetric even-mansour constructions with non-involutory central rounds, and search heuristics for low-latency s-boxes," in *IACR ToSC*, 2017.

[2] Z. B. Aweke, S. F. Yitbarek, R. Qiao, R. Das, M. Hicks, Y. Oren, and T. Austin, "ANVIL: Software-based protection against next-generation Rowhammer attacks," *ACM SIGPLAN Notices*, 2016.

[3] D. J. Bernstein, "Cache-timing attacks on aes," 2005.

[4] A. Bhattacharyya, A. Sandulescu, M. Neugschwandtner, A. Sorniotti, B. Falsafi, M. Payer, and A. Kurmus, "SMoTherSpectre: exploiting speculative execution through port contention," in *CCS*, 2019.

[5] R. Bodduna, V. Ganesan, P. SLPSK, K. Veezhinathan, and C. Rebeiro, "Brutus: Refuting the security claims of the cache timing randomization countermeasure proposed in CEASER," in *IEEE CA Letters*, 2020.

[6] C. Canella, J. Van Bulck, M. Schwarz, M. Lipp, B. Von Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss, "A Systematic Evaluation of Transient Execution Attacks and Defenses," in *USENIX Security Symposium*, 2019.

[7] S. Chari, C. S. Jutla, J. R. Rao, and P. Rohatgi, "Towards sound approaches to counteract power-analysis attacks," in *CRYPTO*, 1999.

[8] M. Green, L. Rodrigues-Lima, A. Zankl, G. Irazoqui, J. Heyszl, and T. Eisenbarth, "AutoLock: Why Cache Attacks on ARM Are Harder Than You Think," in *USENIX Security Symposium*, 2017.

[9] D. Gruss, C. Maurice, A. Fogh, M. Lipp, and S. Mangard, "Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR," in *CCS*, 2016.

[10] D. Gruss, C. Maurice, and S. Mangard, "Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript," in *DIMVA*, 2016.

[11] D. Gruss, C. Maurice, K. Wagner, and S. Mangard, "Flush+Flush: A Fast and Stealthy Cache Attack," in *DIMVA*, 2016.

[12] D. Gruss, R. Spreitzer, and S. Mangard, "Cache Template Attacks: Automating Attacks on Inclusive Last-level Caches," in *USENIX Security Symposium*, 2015.

[13] R. Hund, C. Willems, and T. Holz, "Practical Timing Side Channel Attacks against Kernel Space ASLR," in *S&P*, 2013.

[14] M. S. Inci, B. Gulmezoglu, G. Irazoqui, T. Eisenbarth, and B. Sunar, "Cache Attacks Enable Bulk Key Recovery on the Cloud," in *CHES*, 2016.

[15] Intel Corporation, "Pin - A Dynamic Binary Instrumentation Tool," https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool.

[16] Y. Jang, S. Lee, and T. Kim, "Breaking Kernel Address Space Layout Randomization with Intel TSX," in *CCS*, 2016.

[17] T. Kim, M. Peinado, and G. Mainar-Ruiz, "StealthMem: system-level protection against cache-based side channel attacks in the cloud," in *USENIX Security Symposium*, 2012.

[18] V. Kiriansky, I. Lebedev, S. Amarasinghe, S. Devadas, and J. Emer, "DAWG: A Defense Against Cache Timing Attacks in Speculative Execution Processors," *MICRO*, 2018.

[19] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre Attacks: Exploiting Speculative Execution," in *S&P*, 2019.

[20] P. C. Kocher, "Timing Attacks on Implementations of Diffe-Hellman, RSA, DSS, and Other Systems," in *CRYPTO*, 1996.

[21] J. Kong, O. Acıiçmez, J.-P. Seifert, and H. Zhou, "Hardware-software integrated approaches to defend against software cache-based side channel attacks," in *HPCA*, 2009.

[22] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard, "ARMageddon: Cache Attacks on Mobile Devices," in *USENIX Security Symposium*, 2016.

[23] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading Kernel Memory from User Space," in *USENIX Security Symposium*, 2018.

[24] F. Liu, Q. Ge, Y. Yarom, F. Mckeen, C. Rozas, G. Heiser, and R. B. Lee, "Catalyst: Defeating last-level cache side channel attacks in cloud computing," in *HPCA*, 2016.

[25] F. Liu and R. B. Lee, "Random Fill Cache Architecture," in *MICRO*, 2014.

[26] F. Liu, H. Wu, K. Mai, and R. B. Lee, "Newcache: Secure cache architecture thwarting cache side-channel attacks," *IEEE Micro*, vol. 36, no. 5, pp. 8–16, Sep. 2016. [Online]. Available: https://doi.org/10.1109/MM.2016.85

[27] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-Level Cache Side-Channel Attacks Are Practical," in *S&P*, 2015.

[28] C. Maurice, N. Le Scouarnec, C. Neumann, O. Heen, and A. Francillon, "Reverse Engineering Intel Complex Addressing Using Performance Counters," in *RAID*, 2015.

[29] C. Maurice, C. Neumann, O. Heen, and A. Francillon, "C5: Cross-Cores Cache Covert Channel," in *DIMVA*, 2015.

[30] C. Maurice, M. Weber, M. Schwarz, L. Giner, D. Gruss, C. A. Boano, S. Mangard, and K. Römer, "Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud," in *NDSS*, 2017.

[31] A. Moghimi, G. Irazoqui, and T. Eisenbarth, "Cachezoom: How SGX amplifies the power of cache attacks," in *CHES*, 2017.

[32] J. Monaco, "SoK: Keylogging Side Channels," in *S&P*, 2018.

[33] Y. Oren, V. P. Kemerlis, S. Sethumadhavan, and A. D. Keromytis, "The Spy in the Sandbox: Practical Cache Attacks in JavaScript and Their Implications," in *CCS*, 2015.

[34] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: The case of aes," in *CT-RSA*, 2006.

[35] D. Page, "Theoretical use of cache memory as a cryptanalytic side-channel," *Cryptology ePrint Archive, Report 2002/169*, 2002.

[36] C. Percival, "Cache missing for fun and profit," in *BSDCan*, 2005.

[37] A. Purnal and I. Verbauwhede, "Advanced Profiling for Probabilistic Prime+Probe Attacks and Covert Channels in ScatterCache," in *arXiv 1908.03383*, 2019.

[38] M. K. Qureshi, "CEASER: Mitigating Conflict-based Cache Attacks via Encrypted-address and Remapping," in *MICRO*, 2018.

[39] ——, "New Attacks and Defense for Encrypted-address Cache," in *ISCA*, 2019.

[40] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, "Hey, You, Get off of My Cloud: Exploring Information Leakage in Third-party Compute Clouds," in *CCS*, 2009.

[41] D. Sanchez and C. Kozyrakis, "Vantage: scalable and efficient fine-grain cache partitioning," in *ISCA*, 2011.

[42] S. Sari, O. Demir, and G. Kucuk, "Fairsdp: Fair and secure dynamic cache partitioning," in *International Conference on Computer Science and Engineering (UBMK)*, 2019.

[43] D. Skarlatos, M. Yan, B. Gopireddy, R. Sprabery, J. Torrellas, and C. W. Fletcher, "MicroScope: Enabling Microarchitectural Replay Attacks," in *ISCA*, 2019.

[44] R. Spreitzer and T. Plos, "Cache-access pattern attack on disaligned aes t-tables," in *COSADE*, 2013.

[45] D. Trilla, C. Hernandez, J. Abella, and F. J. Cazorla, "Cache Side-channel Attacks and Time-predictability in High-performance Critical Real-time Systems," in *DAC*, 2018.

[46] Y. Tsunoo, T. Saito, and T. Suzaki, "Cryptanalysis of DES implemented on computers with cache," in *CHES*, 2003.

[47] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-order Execution," in *USENIX Security Symposium*, 2018.

[48] J. Van Bulck, F. Piessens, and R. Strackx, "SGX-Step: A Practical Attack Framework for Precise Enclave Execution Control," in *SysTEX*, 2017.

[49] P. Vila, B. Köpf, and J. F. Morales, "Theory and Practice of Finding Eviction Sets," in *S&P*, 2019.

[50] R. Wang and L. Chen, "Futility scaling: High-associativity cache partitioning," in *MICRO*, 2014.

[51] Z. Wang and R. B. Lee, "New cache designs for thwarting software cache-based side channel attacks," in *ISCA*, 2007.

[52] ——, "A novel cache architecture with enhanced performance and security," in *MICRO*, 2008.

[53] M. Werner, T. Unterluggauer, L. Giner, M. Schwarz, D. Gruss, and S. Mangard, "SCATTERCACHE: Thwarting Cache Attacks via Cache Set Randomization," in *USENIX Security Symposium*, 2019.

[54] Y. Yarom and K. Falkner, "FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-channel Attack," in *USENIX Security Symposium*, 2014.

[55] D. Zhang, Y. Wang, G. E. Suh, and A. C. Myers, "A hardware design language for timing-sensitive information-flow security," *ACM Sigplan Notices*, vol. 50, no. 4, pp. 503–516, 2015.

[56] Z. Zhou, M. K. Reiter, and Y. Zhang, "A software approach to defeating side channels in last-level caches," in *CCS*, 2016.

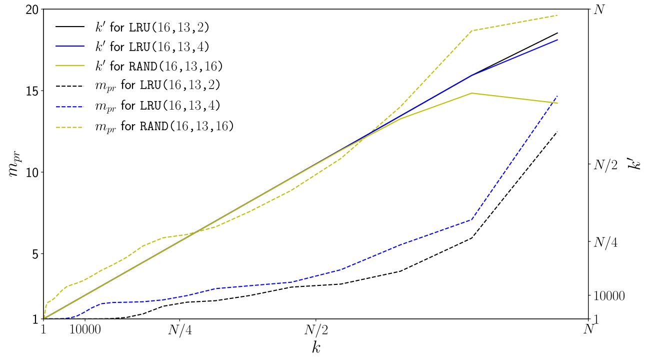


Fig. 11: Empirical k' and m_{pr} (means over 10^5 runs) for different k. Cache instances are denoted $\texttt{RP}(n_w, b, P)$. The **prune** step becomes aggressive from the sixth iteration, if not already terminated.

Appendix

A. Relation between k. k' and m_{pr}

Figure 11 experimentally (cf. Section III-A3) relates pruning parameters k, k' and m_{pr} for different cache instances.

It shows that the **prune** step does not shrink the initial set that much ($k' \approx k$), unless k is very large.

We also observe that instances with stateful replacement policy (LRU) require fewer pruning iterations m_{pr} than those with random replacement. Furthermore, they generally end up with a larger set after pruning (k').

B. Penalty for a cached target

We now derive the catching probability for a target address x that is already cached. Recall that, to detect an access to x, the attacker must first *evict* it from the cache, and *detect* its reintroduction in the cache. Let E denote the event that the Prime+Prune+Probe iteration successfully evicts x from the cache, and D the event that **probe** detects an access to x when it is reloaded. Given that $P[D, \widetilde{E}] = 0$ we have that

$$p_{c,c} = P[D] = P[D, E] = P[D|E] \cdot P[E]$$

1) Random replacement: We have that $P[E] = p_{c,n}(k')$. Recall that for RAND, the target x has n_w potential cache lines to which it can be mapped. Let I denote the number of those locations that the Prime+Prune+Probe iteration occupies.

$$\begin{aligned} P[D|E] &= \sum_{i=0}^{n_w} P[D, I = i|E] \quad \text{(law of total probability)} \\ &= \sum_{i=0}^{n_w} P[D|E, I = i] \cdot P[I = i|E]) \quad \text{(conditional)} \\ &= \sum_{i=0}^{n_w} \frac{i}{n_w} \cdot \frac{P[E|I = i] \cdot P[I = i]}{P[E]} \quad \text{(Bayes' rule)} \\ &= \sum_{i=0}^{n_w} \frac{i}{n_w} \cdot \frac{\frac{i}{n_w} \cdot \binom{n_w}{i} \frac{k'^i (N-k')^{n_w - i}}{N^{n_w}}}{P[E]} \quad \text{(binomial pmf)} \end{aligned}$$

Consequently, we obtain the expression from Table III:

$$p_{c,c} = P[D] = \sum_{i=1}^{n_w} \binom{n_w}{i} \frac{i^2 \cdot k'^i \cdot (N - k')^{n_w - i}}{n_w^2 \cdot N^{n_w}}$$

2) Least recently used (LRU): For LRU, we derive an approximation for $p_{c,c}$. We have that $P[E] = p_{c,n}$. Assume that x is cached in partition P_l, and let P_R be the random variable denoting the partition to which x is reloaded.

$$\begin{aligned} P[D|E] &= \sum_{i=1}^{n_w} P[D, P_R = P_i|E] \quad \text{(law of total probability)} \\ &= P[D, P_R = P_l|E] + \sum_{i \neq l} P[D, P_R = P_i|E] \\ &= P[D|P_R = P_l, E] \cdot P[P_l|E] + \\ &\quad \sum_{i \neq l} P[D|P_R = P_i, E] \cdot P[P_R = P_i|E] \quad \text{(conditional)} \\ &\approx 1 \cdot \frac{1}{P} + \sum_{i \neq l} p_{c,n} \cdot \frac{1}{P} \end{aligned}$$

Consequently, we obtain the expression from Table III:

$$p_{c,c} = P[D] \approx p_{c,n} \cdot \frac{p_{c,n}(P-1)+1}{P}$$

C. Theoretical vs. experimental catching probabilities

Figure 12 presents experimental and visual support for the analysis in Section V-B. Since the attacker cannot directly set k', we choose k as the independent variable in Figure 12. However, recall that in Table III, the catching probabilities p_c are described in terms of k' for RAND, *i.e.*, *after* pruning, and in terms of k for LRU, *i.e.*, *before* pruning. The reason is that the **prune** step does not allow concise statistical modeling.

Hence, for RAND, we approximate k' with k, noting that the exact p_c in terms of k' becomes an upper bound in terms of k. We observe theory to match practice very well for $k \leq 0.6N$, as $k \approx k'$ (cf. Appendix A). The upper bound becomes noticeable for very large k, exactly because k' and k diverge.

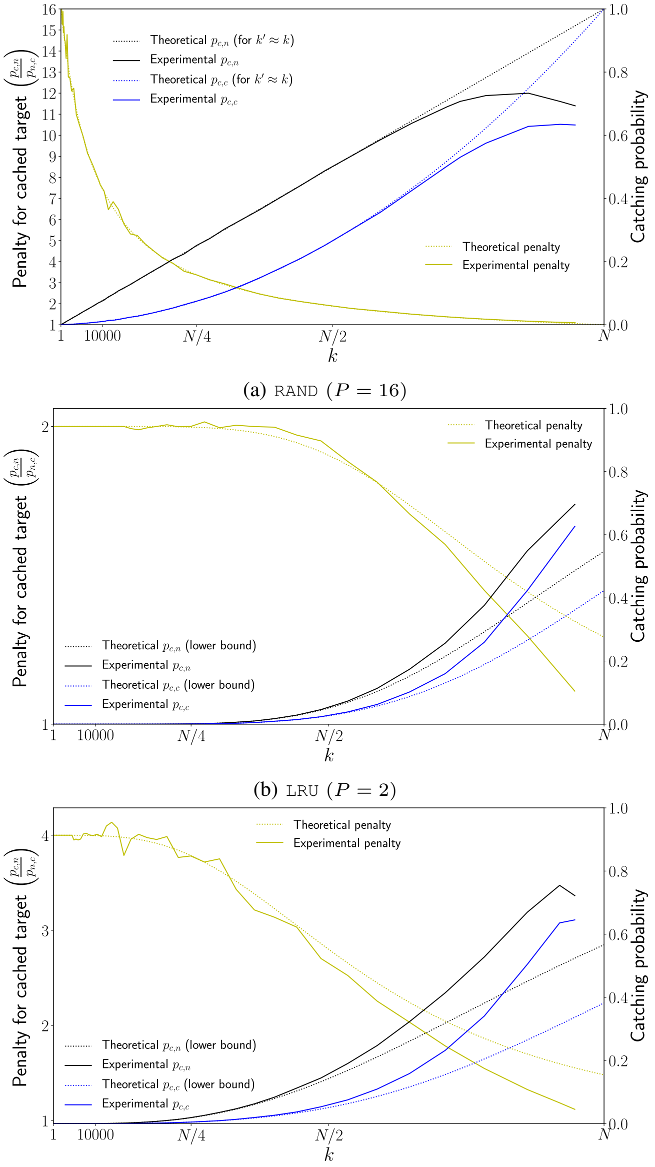
For LRU, the theoretical expressions were derived as a lower bound (cf. Section V-B1). It is a good approximation for low k. For larger k, although **prune** does not eliminate a lot of elements (cf. Appendix A), it increases p_c significantly because addresses that were evicted and reintroduced have another attempt at hitting the correct cache set and partition.

In principle, one can also choose $k > N$ to increase p_c beyond those depicted in Figure 12, all the way to $p_c \approx 1$. However, for such a configuration, the **prune** step becomes excessive, both for RAND and LRU. It will require a large number of non-aggressive iterations to avoid shrinking k' too fast (cf. Appendix A), followed by many aggressive iterations until there are no more self-evictions. As a result, the cache accesses needed to terminate the **prune** step are significant.

D. Known-Plaintext AES T-Tables Attack

We implement the "One-Round Attack" proposed by Osvik et al. [34]. Because their work already describes the attack in detail, we only add a high-level overview here and discuss implications for our cache model.

In the T-tables implementation of 128-bit AES, each of the first 9 rounds accesses all 4 T-tables 4 times each. Per round, one access is made for each key byte and plaintext byte pair



(a) RAND ($P = 16$)

(b) LRU ($P = 2$)

(c) LRU ($P = 4$)

Fig. 12: Catching probabilities for cached and uncached targets in theory and practice. Experimental results obtained from simulating a standalone 8MB randomized cache ($n_w = 16$, $b = 13$), averaged over 10^5 runs. The **prune** step becomes aggressive from the sixth iteration, if not already terminated.

$p_i \oplus k_i$ to table T_j, $j \equiv i \; mod \; 4$, $i \in [0..15]$. T-tables consist of 256 entries of 4 bytes each, but as cache lines are typically 64 bytes in size, we can consider them arrays of 16 entries each, addressed by the upper nibble of $p_i \oplus k_i$. Thus entries $T_j[\lceil p_i \oplus k_i \rceil_4]$ will *always* be accessed in round 1, while other entries *may* be accessed in later rounds. This produces the statistical difference we exploit.

We construct a matrix M of 16×16 bins, where columns represent the key byte position i, and rows the upper key nibble candidate $\lceil k_i \rceil_4$. Because of the table association mentioned before, table T_j produces columns $M_{*,i}$. We now measure accesses to our probe sets $A_{j,l}, l \in [0..15]$ for all 16 addresses of each table, and simply add the resulting amount of misses

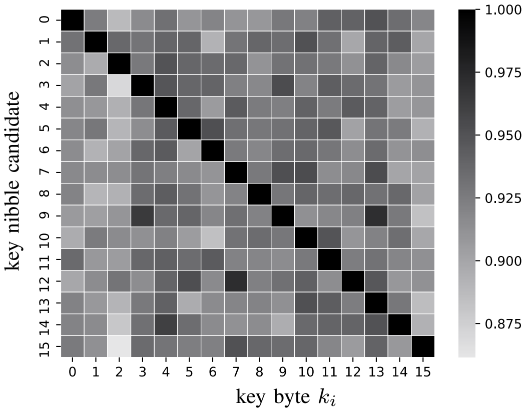


Fig. 13: Result of the AES T-tables attack. Columns normed to the highest value. The partial key can be read from left to right by the darkest field in each column. Pictured: Key=0x00102030405060708090a0b0c0d0e0f0

from each probe set to a bin for each plaintext byte: $M[l \oplus p_i, i] \mathrel{+}= misses(A_{j,l})$. The key nibbles are now the highest value in each column. Depending on the size and quality of the probe sets, this takes several thousand random plaintexts. Figure 13 shows the matrix for a successful attack.

The noteworthy part for our attack is that, for uniformly distributed random plaintexts, each table address contributes equally to each bin in the appropriate columns of M. For this reason, this attack absorbs differences in probe set sizes.