A Theoretical Framework For Modeling Cache-based Side-Channel Attacks and Countermeasures

Nivedita Shrivastava[1] and Smruti R. Sarangi[2]
Electrical Engineering Department
Indian Institute of Technology, Delhi, India
Email: nivedita.shrivastava@ee.iitd.ac.in[1], srsarangi@cse.iitd.ac.in[2]

***Abstract*—Cache-based side-channel attacks are a critical security threat – they have two distinct phases: *(i)* identify the security-sensitive addresses (eviction sets) and *(ii)* reconstruct the cryptographic key by monitoring victim accesses. Although extensive research has focused on complicating the first phase, the second phase, namely key reconstruction, has received less attention. State-of-the-art approaches either take a long time or address a highly reduced version of this problem. The best solution for the AES algorithm requires at least 5 billion attempts, which is impractical. Thus, researchers have focused on simpler algorithms, such as the ECDSA algorithm.**

We introduce a novel theoretical framework, the 4R model, to classify countermeasures (CMs) based on their impact on address mappings. We observe that reconstructing the key is quite difficult, as heuristic- and ML-based methods fail under realistic address obfuscation scenarios. To address this challenge, we propose a two-phase attack framework that dynamically adapts to address remapping and noise injection. It employs real-time state validation to overcome dynamic obfuscation mechanisms and retrieve the secret key. Our algorithm is noise-tolerant and efficiently recovers the AES key in roughly 50k attempts with the most sophisticated CMs, demonstrating its robustness and practicality.

***Index Terms*—Side-channel attacks, AES key recovery, Hardware security, Key reconstruction**

I. Introduction

Cache-based side-channel attacks (CSCA) represent a class of critical attack vectors in modern computing systems; they exploit the shared cache memory to infer sensitive information from victim processes. By analyzing cache access patterns [1], timing differences [2], [3], and eviction behaviors [4], [5], attackers can extract private data such as cryptographic keys or other sensitive information, posing a significant risk to secure computing.

These attacks comprise two steps: ① identify the sensitive target addresses (and their eviction sets) and ② monitor victim accesses to these addresses to reconstruct the secret key. In response, numerous countermeasures (CMs) [6]–[13] have been proposed. They focus on complicating step one. Specifically, they make the discovery of eviction sets challenging, where an eviction set is defined as the set of addresses that need to be accessed to evict the target address from the cache. All addresses in the eviction set map to the same set.

The second step has not received a lot of attention up till now, where attackers retrieve the key after some computation. The prevailing opinion among researchers is to focus on "stopping the enemy at the gates" – assume that the first step fails and the attacker never moves to the second step. This has led to an overemphasis on the first step, while neglecting the challenges posed by the second step.

The barrier created by the first step is not insurmountable. With some effort, eviction sets can be effectively and reliably found out [14], [15]. There is thus a need to focus on the second step and complicate the process of key recovery, even if the accessed addresses are known either exactly or probabilistically. Understanding the limitations of the second step helps us choose the right countermeasure for the first step given that there is a trade-off between the degree of obfuscation of the eviction sets and performance overheads.

To the best of our knowledge, Shrivastava et al. [16] and Zhao et al. [17] are among the few most influential works that attempt to address step two. The methodology proposed by Shrivastava et al. [16] faces two major limitations. ① The bounds focus solely on address randomization, ignoring the other three possible obfuscation techniques, which we shall discuss later. ② For a T-table based AES implementation [18] with 1024 target addresses, even with their $n^{O(\log(\log(n)))}$ bound, their approach requires billions of encryptions under ideal noise-free conditions, making the attack impractical in real-world scenarios where noise is prevalent. Similarly, Zhao et al.'s work [17] that focuses on the relatively simple ECDSA cipher [19], relies on frequency-domain analysis, which assumes periodicity in access patterns. This assumption fails for AES, where random keys along with sophisticated obfuscations result in highly irregular T-table access patterns, rendering such techniques ineffective.

In this work, we focus on the standard T-table-based AES implementation. We define a *virtual node (VN)* as the result of an XOR operation between a plaintext byte and a corresponding key byte. It corresponds to the row of the T-table that is accessed. These VNs (T-table accesses) are subsequently mapped to specific memory addresses, referred to as physical nodes (PNs). Our approach introduces a novel theoretical framework,

hereby named the $4R$-model, which classifies existing CMs into four distinct categories based on their impact on the VN-to-PN mapping: redundancy [12], [13], access removal [20], randomization [9], [11], and periodic remapping [8], [10]. This classification provides a comprehensive method to evaluate the effectiveness of CMs.

We assume that the attacker has successfully completed step one, in the presence of sophisticated CMs, and retrieved some "noisy" eviction sets. With such noisy eviction sets, isolating target addresses and completing step two becomes significantly more challenging due to the lack of information [9]–[11], [13], [20]. We model the VN-to-PN mapping using hypercubes, as proposed by Shrivastava et al. [16], and employ several heuristic-based as well as ML-based algorithms to reconstruct this mapping. Our findings reveal that these attacks do not reliably recover the original mapping, highlighting the robustness of CMs in practical scenarios.

Building on these insights, we propose a two-phase attack framework capable of bypassing all known categories of CMs. In the first phase, the adversary iteratively guesses a VN-to-PN mapping using the first key byte. In the second phase, she generates a proxy mapping for the unknown key bytes and creates VNs. By combining multiple VN-to-PN mapping snippets with proxy mappings, the adversary recovers the secret key regardless of the countermeasures. The full key is ultimately validated through a full-scale encryption process. Until the validation is successful, this process keeps repeating.

For scenarios with dynamic remapping, where the VN-to-PN mapping evolves over time, the attack adapts through a real-time verification mechanism. The adversary continuously validates the extracted data against the current state, ensuring alignment. Despite the increased effort required, this adaptive approach successfully overcomes the additional challenges, demonstrating the resilience and precision of the framework.

The contributions of the paper are as follows. ❶ We propose a theoretical framework (4R model) to model all known CMs for CSCAs. The paper proposes a general taxonomy for classifying all such work using 4 axes: Redundancy, access Removal, Randomness and Remapping (4R). ❷ We propose a sophisticated attack against known CMs in this space, which is based on a mapping between virtual and physical nodes (VNs and PNs). We look at problems with increasing difficulty. The most basic version of the problem that does not have remapping, redundancy and randomness was solved in $n^{O(log(log(n)))}$ time by Shrivastava et al. We propose an algorithm that runs in $O(n^2)$ time. ❸ Next, we propose a family of problems that are at different points of the 4R spectrum. We observed that all of them are in general hard to solve using conventional hypercube-based approaches and other heuristic-based approaches. However, using our method, they can be solved quite easily. The hardest problem in this space is when all 4 Rs (4R) are enabled, and there is no sparsity. We show that it too can be solved with regular verification in roughly 50k attempts.

The paper is organized as follows. §II provides the relevant background. §III presents the threat model and §IV shows the classification of the CMs. §V discusses the design details, §VI shows the results, and we finally conclude in §VII.

II. Background and Related Work

A. Software Implementation of AES

AES is a symmetric-key block cipher operating on a 128-bit data block, referred to as the *state*. This state is represented as a matrix of 16 bytes organized in a 4×4 matrix structure, where each byte is denoted as $s = \{s_0, \ldots, s_{15}\}$. The AES encryption process consists of an initial *AddRoundKey* transformation followed by a sequence of rounds, each comprising four main transformations: *SubBytes*, *ShiftRows*, *MixColumns*, and *AddRoundKey*. The computation requires nr rounds (depends on the key length); for AES-128, $nr = 10$. In the final round, the *MixColumns* transformation is omitted. Note that the initial *AddRoundKey* transformation is referred to as the 0^{th} round.

To enhance the efficiency of software implementations, precomputed *T-tables* are used, which combine the roles of the *SubBytes*, *ShiftRows* and *MixColumns* transformations. These T-tables allow the AES round operations to be performed through simple look-up and XOR operations, thereby bypassing the computational complexity of the transformations themselves. AES encryption primarily uses four T-tables T_0, T_1, T_2 and T_3, each containing 256 4-byte entries. The size of each such T-table is 1 KB.

Notations- In the notation used, the superscript indicates the round number, whereas the subscript denotes the byte position within the state, key or plaintext.

Let the initial round key be denoted by k^0 and the plaintext input be p (both 16 bytes). k^0 comprises 16 individual bytes: $k = \{k_0^0, k_1^0, \ldots, k_{15}^0\}$. The plaintext can be represented as $p = \{p_0, p_1, \ldots, p_{15}\}$.

The initial state byte s_i^0 represents each i^{th} byte of the state matrix s^0 after the XOR operation between the corresponding plaintext byte and the initial round key byte. This operation is defined as: $s_i^0 = p_i \oplus k_i^0$ where $\oplus$ denotes the bitwise XOR (exclusive OR) operation.

This initial transformation produces the input for subsequent rounds. At each step, the T-tables are accessed based on these intermediate state bytes. The state bytes after the final round represent the ciphertext. The AES transformation for the state values across r rounds is represented as follows.

$$
\begin{aligned}
(s_0^{r+1}, s_1^{r+1}, s_2^{r+1}, s_3^{r+1}) &= T_0[s_0^r] \oplus T_1[s_5^r] \oplus T_2[s_{10}^r] \oplus \\
&\quad T_3[s_{15}^r] \oplus \{k_0^{r+1}, k_1^{r+1}, k_2^{r+1}, k_3^{r+1}\}, \\
(s_4^{r+1}, s_5^{r+1}, s_6^{r+1}, s_7^{r+1}) &= T_0[s_4^r] \oplus T_1[s_9^r] \oplus T_2[s_{14}^r] \oplus \\
&\quad T_3[s_3^r] \oplus \{k_4^{r+1}, k_5^{r+1}, k_6^{r+1}, k_7^{r+1}\}, \\
(s_8^{r+1}, s_9^{r+1}, s_{10}^{r+1}, s_{11}^{r+1}) &= T_0[s_8^r] \oplus T_1[s_{13}^r] \oplus T_2[s_2^r] \oplus \\
&\quad T_3[s_7^r] \oplus \{k_8^{r+1}, k_9^{r+1}, k_{10}^{r+1}, k_{11}^{r+1}\}, \\
(s_{12}^{r+1}, s_{13}^{r+1}, s_{14}^{r+1}, s_{15}^{r+1}) &= T_0[s_{12}^r] \oplus T_1[s_1^r] \oplus T_2[s_6^r] \oplus \\
&\quad T_3[s_{11}^r] \oplus \{k_{12}^{r+1}, k_{13}^{r+1}, k_{14}^{r+1}, k_{15}^{r+1}\}.
\end{aligned}
$$

We can see that the T-table accesses are inherently dependent on both the secret key and the plaintext. Cache-based

side-channel attacks exploit this key-dependent behavior of T-table accesses to infer the secret key.

For the purpose of analysis, we refer to the entries of the T-table as *virtual nodes* (VNs), which are mapped to specific cache lines. The memory addresses of these cache lines are referred to as *physical nodes* (PNs).

B. Related Work

Shrivastava et al. [16] highlight the vulnerability of address-based CMs, and prove that the search space for reconstructing address mappings can be reduced from $n!$ to $n^{O(\log(\log(n)))}$. This reduction leverages the fact that a 1-bit Hamming distance in plaintext bytes (for the same position) results in a 1-bit Hamming distance of the corresponding state bytes after a round of encryption. We shall improve this result to $O(n^2)$. using a much simpler EDCSA cipher. They highlight the challenges posed by noise and dynamic remapping. To enhance the attack, they propose techniques like parallel probing for victim access monitoring and frequency-domain analysis using power spectral density to identify target cache sets efficiently. AES accesses lack repetitive patterns, hence, frequency-domain analysis is not very useful here.

III. Threat Model

We consider a cross-core side-channel attack in a multi-tenant, shared computing environment. In accordance with standard threat models [7], [16], [17], the attacker and the victim are assumed to run concurrently on separate physical cores of the same processor, but they share the Last-Level Cache (LLC). The victim performs AES encryption using T-tables. The attacker's objective is to recover the victim's secret key by exploiting the side-channel information leaked via shared cache accesses while accessing the T-tables.

Attacker's Capability- The attacker can influence the victim's T-table accesses by sending carefully crafted plaintexts to the victim. In a realistic scenario, the victim might be running a cryptographic service that the attacker can interact with. The LLC is shared between the cores, meaning that cache lines accessed by the victim during AES encryption will also be visible to the attacker (albeit indirectly). The attacker can monitor the victim's cache access patterns. We assume that the attacker is aware of all the *noisy* eviction sets (not precise), even in the presence of sophisticated cache access obfuscation techniques.

IV. Classification of Countermeasures (CMs)

We classify various CMs based on their impact on the VN-to-PN mapping, as summarized in Table I. These CMs are classified as follows:

A. Redundancy-Based Techniques

Li et al. [12] propose *Prefender*, a prefetching-based defense that introduces extraneous cache activity to mislead attackers. It comprises the Data Scale Tracker (DST) and the Access Pattern Tracker (APT). DST prefetches cache lines expected to be accessed by the victim, mimicking secret-dependent behavior, while APT prefetches cache lines the attacker is likely to access (probe), creating misleading cache hits.

Similarly, Mosquera et al. [13] propose to introduce false cache hits using a novel cache structure – *Guard* cache. During execution, when data is evicted from the primary cache, it is redirected to the *Guard* cache. Since the access time of the *Guard* cache is comparable to that of the primary cache, any missing data retrieved from it gives the impression that it was originally present in the primary cache. They also propose to introduce false cache misses by randomly evicting cache lines from the primary cache. The likelihood of utilizing the *Guard* cache, along with the rate at which data is evicted from the primary cache (resulting in false misses) can be adjusted.

Li et al. proposed [20] to introduce a dedicated, small eviction-hiding buffer – *Treasure* cache. This buffer temporarily holds evicted LLC entries, which can be reloaded directly back into the LLC upon an LLC miss that matches an entry in the buffer.

► *Implications:* A cache-side channel attack typically unfolds in three steps as follows. ① the attacker evicts a set of suspected target addresses from the cache; ② the attacker allows the victim to execute and access memory; ③ the attacker measures which addresses were accessed by measuring cache access times (and inferring hits).

When CMs such as Prefender or Guard cache are active, the attack process is disrupted, since the attacker no longer observes a single cache hit corresponding to the victim's secret-dependent access, but instead observes multiple cache hits at different target addresses. For example, in *Prefender*, the APT predicts the target address the attacker will measure next (in the third step) and prefetches it in advance, making it appear as if the victim accessed it. Simultaneously, the DST anticipates the cache lines the victim will access in the *next* encryption and prefetches them. This causes multiple cache lines from different encryptions (different plaintexts) to be mapped to the same plaintext or VN, injecting *redundancy* into the attacker's observations. This results in a one-to-many VN-to-PN mapping as shown in Figure 1(b). Figure 1(a) illustrates the actual VN-to-PN mapping, where multiple VNs can be mapped to the same PN since multiple entries in the T-table can reside within a single cache line.

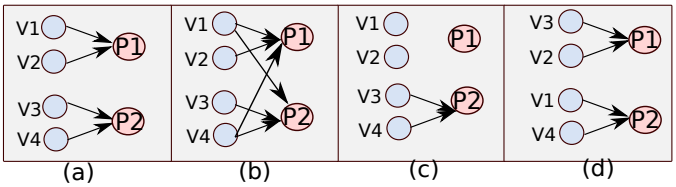


Fig. 1: Classification of Different Countermeasures. **(a)** Actual VN-to-PN mapping **(b)** Mapping after redundancies **(c)** Mapping after removal **(d)** Mapping after randomization

B. Access- Removal-Based Techniques

Thoma et al. [11] propose *ClepsydraCache* to mitigate the effect of state-of-the-art cache attacks using a combination of

Work	Venue	Insight	Impact
Prefender [12]	TC'24	Prefetches the victim's and adversary's data to introduce fake cache hits	Redundancy
Guard Cache [13]	CAL'23	Introduces false misses by randomly evicting data. Introduces false cache hits by storing evicted data in the *Guard* cache.	Access removal + Redundancy
Treasure Cache [20]	TDSC'24	Stores the evicted cache lines, introduces false cache hits.	Redundancy
Clepsydra Cache [11]	USENIX'23	Random address to index mapping. Random cache evictions.	Randomized mapping + Access removal
Scarf [9]	USENIX'23	Random address to cache index mapping.	Randomized mapping
CEASER [8]	ISCA'19	Periodically remaps the address mapping	Remapping
SCATTERCACHE [10]	USENIX'19	Periodically remaps the address mapping	Remapping

TABLE I: Summary of the Countermeasures

cache decay and index randomization. Each cache entry is linked with a Time- To-Live (TTL) value. The TTL is steadily reduced and, when it expires, the entry is evicted from the cache (known as *cache decay*). The authors also randomize the process of mapping block addresses to cache lines.

In Guard Cache [13], cache lines are randomly evicted from the primary cache, introducing artificial cache misses. These unpredictable evictions disrupt the attacker's ability to track the victim's memory accesses accurately.

► *Implications:* Such CMs that introduce false cache misses disrupt the attacker's measurement process. After the attacker evicts the target addresses and the victim reloads the secret-dependent address, these CMs may randomly evict the secret-dependent address again, converting a potential cache hit into a cache miss. Consequently, the adversary may not observe a cache hit (PN) associated with the VN, as illustrated in Figure 1(c). This behavior results in missed VN-to-PN connections.

C. Randomization-Based Techniques

Canale et al. [9] propose *SCARF*, which is a cryptographically sound, tailor-made cache cipher, that randomizes the address-to-cache-index mapping and makes it difficult to construct the eviction sets. Basically, it is a tweakable block cipher with a 48-bit tweak and 10-bit block size. *SCARF* uses a 240-bit secret key. Similarly, *ClepsydraCache* [11] also randomizes the mapping of the address to the cache index.

► *Implications:* In a conventional system, the mapping of a memory address to a cache line (cache index) is typically fixed and deterministic. Randomization techniques alter the deterministic address-to-cache mapping, disrupting an attacker's ability to infer memory access patterns. As a result, the VN no longer corresponds to a single PN (the cache line). This technique randomizes the mapping between VNs and PNs, as shown in Figure 1 (d).

D. Remapping-Based Techniques

Qureshi et al. [21] proposed CEASER, which employs periodic remapping of cache sets to disrupt eviction sets and counter side-channel attacks. By dynamically changing the mappings at regular intervals, it ensures that attackers cannot rely on static mappings to exploit cache behavior. Similarly, Werner et al. propose ScatterCache [10], which uses fine-grained randomization and dynamic remapping of cache lines to decouple addresses from cache indices. Its remapping occurs at runtime, preventing attackers from establishing predictable eviction sets, while maintaining low performance overheads.

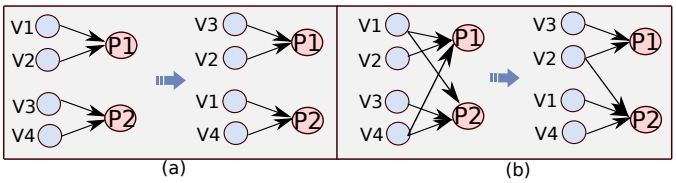


Fig. 2: Remapping **(a)** Without redundancies **(b)** With randomization and redundancies

► *Implications:* The mapping between VNs and PNs experiences periodic randomization over the course of time, as illustrated in Figure 2(a). Figure 2(b) depicts the scenario when redundancies and randomization are incorporated into the system.

V. Attack Methodology

We propose a two-phase attack strategy to infer secret key bytes, wherein the attacker iterates through all 256 possible values of the first key byte and estimates the remaining unknown key bytes until the complete key is recovered.

In the first phase, the attacker finds the mapping between the VNs and the PNs by using the first key byte. This mapping serves as the foundation for the second phase, where the attacker uses the derived mapping to infer the remaining unknown key bytes, as shown in Algorithm 1. Once the complete key is derived, the attacker verifies it by encrypting a plaintext and comparing the (estimated) ciphertext with the expected ciphertext. If the key is incorrect, the process repeats with a different first key byte until the correct key is found.

1) Phase-1: Deriving the VN-to-PN Mapping with the First Key Byte: The attacker begins by selecting a candidate for the first key byte and infers the VN-to-PN mapping as shown in Algorithm 2. She iterates over all possible values of plaintext bytes (0, 1 . . . 255). For each plaintext byte, the corresponding VN is computed as the XOR between the plaintext byte and the first key byte. The attacker then determines the corresponding PN associated with the VN.

In a real-world system, this process involves probing the cache to identify the PNs that consistently result in cache hits when a specific VN is accessed. This step is represented by the function FindPhysicalNode in Algorithm 2. The observed PNs are then mapped to the corresponding VN. Note that in a noisy system it will be a one-to-many mapping.

2) Phase-2: Inferring Unknown Key Bytes: In the second phase, the attacker targets a secret key byte. While the attacker can control the plaintext bytes, she lacks prior knowledge of the secret key bytes and, consequently, the corresponding VNs. ❶ The first step is to collect the plaintext-to-PN mapping for the unknown secret key byte. This process follows the same approach as Phase 1. Since the attacker lacks knowledge of VNs, she uses plaintext bytes instead (see the GENERATEMAPPING function in Algorithm 2). Internally, the system computes the VN as the XOR of the plaintext and the unknown key byte, but this VN is not known to the attacker.
❷ Next, the attacker identifies the candidate key bytes (a superset) by leveraging the VN-PN and plaintext-to-PN mappings as shown in Algorithm 1. Initially, the attacker identifies the plaintext byte values and VNs that map to the same PN in their respective mappings. For each physical node, the attacker computes the XOR between all pairs of mapped plaintext bytes and virtual nodes using the ALLPAIRXOR function. This results in a set of candidate key bytes at a specific position (no duplicates). As the attack progresses across multiple PNs (corresponding to the same byte position), the set of candidate key bytes keeps reducing and ultimately becomes 1 if the initial guess for $\mathbf{K}[0]$ is correct. The attack proceeds to find all key bytes at non-zero positions using this approach.

A. In the Presence of Remapping

If the victim dynamically remaps the VN-to-PN mapping after the completion of Phases 1 and 2, the attacker must adopt an adaptive verification-based strategy to ensure accurate key recovery.

In this scenario, after completing Phase 2, the attacker re-executes Phase 1 to verify whether the VN-to-PN mapping has changed. If the mapping remains consistent, the attacker retains the previously derived key byte and proceeds to target the next byte. However, if the mapping has changed, the attacker must remount the attack by re-executing both Phase 1 and Phase 2 to recompute the key byte.

This iterative process continues until the key byte is successfully recovered or the maximum iteration limit is reached. This approach ensures convergence in practice because no practical system remaps memory very quickly. It is thus quite unlikely that a remapping will always happen between Phases 1 and 2. This systematic cross-verification of the consistency of the mapping ensures robustness against dynamic remapping, making the attack methodology resilient.

VI. RESULTS

We use a machine with the following system configuration: Intel i7-8550U CPU (1.8 GHz) with 64 cores, 16 GB DRAM, and a three-level cache hierarchy (L1: 32 KB, L2: 256 KB, L3: 8 MB). The software environment includes Ubuntu 18.04 with the Linux kernel 5.4, and GCC 7.5. We developed a simulator in C++ to model both attack strategies and CMs, enabling a detailed analysis of their implications. The simulator is designed to incorporate all relevant parameters, including VN-to-PN mapping dynamics, remapping probabilities, and real-world noise injection levels as shown in Table II. These parameters are carefully chosen on the basis of extensive real-world analysis of the CMs, ensuring that the simulation accurately reflects the practical system behavior even in the presence of noise.

In the presence of randomness-based CMs, we mounted the attack while increasing the number of VNs per PN.

Algorithm 1 Two-Phase Key Inference Attack Algorithm

Input: Plaintext $\mathbf{PT}$, Known ciphertext $\mathbf{CT}$, N is the key length
Output: Key $\mathbf{K} = \{\mathbf{K}[0], \mathbf{K}[1] \dots \mathbf{K}[N-1]\}$ or $\langle$`failed-attack`$\rangle$

▷ Iterate over all the possible values of the first key byte
for all $b \in \{0, \dots, 255\}$ **do** ▷ Guess for $\mathbf{K}[0]$
  ▷ Phase 1: Establish the VN-to-PN Mapping
  $M_{\mathcal{VN}\mapsto\mathcal{PN}} \leftarrow$ GENERATEMAPPING$(1, b, 0)$ ▷ $isVN = 1$

  ▷ Phase 2: Infer Remaining Key Bytes
  for all $pos \in \{1, \dots, N-1\}$ **do**
    $M_{\mathcal{PT}\mapsto\mathcal{PN}} \leftarrow$ GENERATEMAPPING$(0, -, pos)$ ▷ $isVN = 0$

    $\mathcal{K}_{\text{candidates}} \leftarrow \{\ \}$
    `firstIter` $\leftarrow$ **true**
    $\mathcal{P} \leftarrow \bigcup_{0\le i\le 255} M_{\mathcal{PT}\mapsto\mathcal{PN}}[i]$ ▷ all mapped PNs

    for all $pn \in \mathcal{P}$ **do** ▷ iterate over the PNs
      ▷ Find all plaintext bytes and VNs that map to pn
      $\mathcal{S}_b \leftarrow M^{-1}_{\mathcal{PT}\mapsto\mathcal{PN}}(pn)$ ▷ preimage of pn (text bytes)
      $\mathcal{S}_v \leftarrow M^{-1}_{\mathcal{VN}\mapsto\mathcal{PN}}(pn)$ ▷ preimage of pn (VNs)

      if `firstIter` **then**
        `firstIter` $\leftarrow$ **false**
        $\mathcal{K}_{\text{candidates}} \leftarrow$ ALLPAIRXOR$(\mathcal{S}_b, \mathcal{S}_v)$ ▷ compute superset
      else
        ▷ Prune the set of candidate key bytes
        $\mathcal{K}_{\text{candidates}} \leftarrow \mathcal{K}_{\text{candidates}} \cap$ ALLPAIRXOR$(\mathcal{S}_b, \mathcal{S}_v)$
      end if

      if $|\mathcal{K}_{\text{candidates}}| = 1$ **then** ▷ Convergence reached
        $\mathbf{K}[pos] \leftarrow \mathcal{K}_{\text{candidates}}[0]$
        break
      end if
    end for
  end for

  ▷ Verify derived key candidate
  if ENCRYPT($\mathbf{PT}$,$\mathbf{K}$) = $\mathbf{CT}$ **then**
    return $\mathbf{K}$
  end if
end for
return $\langle$`failed-attack`$\rangle$

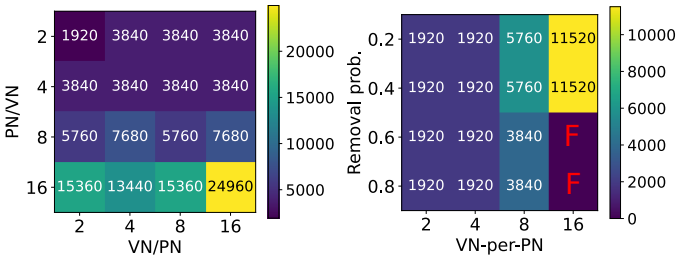
Algorithm 2 Generalized Mapping Extraction Algorithm

function GENERATEMAPPING($isVN$, b, pos)
  Input: Boolean $isVN$, key byte b (guess), key byte pos. pos
  Output: Mapping $M_{\mathcal{X}\mapsto\mathcal{PN}}$, $\mathcal{X}$ is either $\mathcal{PT}$ or $\mathcal{VN}$ (see $isVN$)
  $M_{\mathcal{X}\mapsto\mathcal{PN}} \leftarrow \{\ \}$; $\mathcal{P} \leftarrow \{\}$
  ▷ Iterate over all the possible plaintext bytes
  for all $pt_{byte} \in \{0, 1 \dots 255\}$ **do**
    if $isVN$ **then**
      $x \leftarrow pt_{byte} \oplus b$ ▷ For VN-PN mapping (Phase 1)
    else
      $x \leftarrow pt_{byte}$ ▷ For PT-PN mapping (Phase 2)
    end if
    $\mathcal{P} \leftarrow$ FINDPHYSICALNODES(pt_{byte}, pos) ▷ Set of PNs
    $M_{\mathcal{X}\mapsto\mathcal{PN}}[x] \leftarrow M_{\mathcal{X}\mapsto\mathcal{PN}}[x] \cup \mathcal{P}$
  end for
  return $M_{\mathcal{X}\mapsto\mathcal{PN}}$ ▷ Maps a VN/PT to a set of PNs
end function

Parameter	Significance	CM
VN/PN	# of VNs mapped to the same PN	Randomness
Remap Prob.	Probability of change in the mapping	Remapping
Removal Prob.	Prob. of missing PN-VN connection	Removal
PN/VN	Maximum # of PNs mapped to a VN	Redundancy

TABLE II: Simulation Parameters

The results indicate that as the VN/PN ratio increases, the number of iterations required to retrieve the key also increases. Specifically, for VN/PN ratios of 2, 4, 6, and 8, the number of tries required were 1920, 1920, 7680, and 11520, respectively. We then introduced redundancy-based CMs into the system also by increasing the number of PNs assigned to each VN. We present the results in Figure 3a.



(a) Redundancy (b) Removal

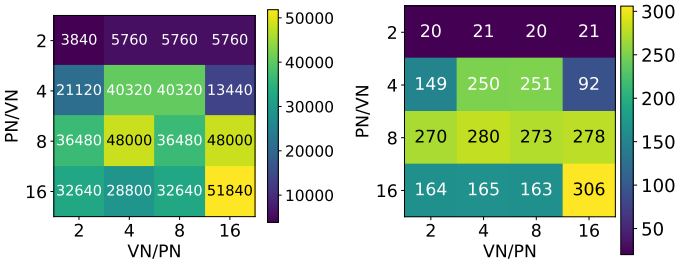
Fig. 3: Number of tries required in the presence of different CMs. $F \rightarrow$ not enough samples to mount the attack.

Subsequently, we evaluated the attack under removal-based countermeasures, where noise was introduced by randomly removing physical nodes according to the *removal probability*. As shown in Figure 3b, the key byte could still be estimated within a finite number of iterations. However, the attack failed (key candidates do not converge) when the noise became excessive, i.e., when the sample size was too small (probability ≥ 0.6).

Figures 4a and Figure 4b show the number of attempts and the attack time, respectively, required to retrieve the key in the presence of a remapping-based countermeasure with a probability of remap of 0.2, along with redundancy, removal (removal prob. 0.2) and randomness. We observe that both the number of attempts and the attack time are significantly higher compared to other CMs.

VII. Conclusion

Side-channel attacks pose a serious threat to secure systems as they exploit shared caches to extract sensitive data. While research has focused on disrupting target eviction set identification, the equally critical phase of key reconstruction has been largely neglected. Existing solutions are either computationally impractical or limited to simple scenarios, leaving the broader challenge unaddressed. In this work, we introduced a comprehensive theoretical framework, the 4R model, to classify CMs into four key categories: redundancy, removal, randomization and remapping. We observed that the presence of all 4Rs significantly complicates key reconstruction, rendering conventional approaches ineffective. To address this



(a) (b)

Fig. 4: Results in the presence of remapping, randomness, removal and redundancy. **(a)** Number of tries **(b)** Attack time (in seconds)

challenge, we proposed a dual-phase attack framework that adapts dynamically to sophisticated CMs to recover the key. We require just about 50k attempts to retrieve the key, far surpassing prior state-of-the-art methods that require billions of operations.

References

[1] D. A. Osvik *et al.*, "Cache attacks and countermeasures: the case of aes," in *Cryptographers' track at the RSA conf.*, 2006, pp. 1–20.

[2] Y. Yarom and K. Falkner, "Flush+ reload: A high resolution, low noise, l3 cache side-channel attack," in *USENIX*, 2014.

[3] D. Gruss *et al.*, "Flush+ flush: a fast and stealthy cache attack," in *DIMVA*, 2016.

[4] F. Liu *et al.*, "Last-level cache side-channel attacks are practical," in *S&P*, 2015.

[5] W. Song and P. Liu, "Dynamically finding minimal eviction sets can be quicker than you think for side-channel attacks against the {LLC}," in *RAID*, 2019.

[6] A. Bhatla *et al.*, "The maya cache: A storage-efficient and secure fully-associative last-level cache," in *ISCA*. IEEE, 2024, pp. 32–44.

[7] G. Saileshwar and M. Qureshi, "{MIRAGE}: Mitigating {Conflict-Based} cache attacks with a practical {Fully-Associative} design," in *USENIX*, 2021, pp. 1379–1396.

[8] M. K. Qureshi, "Ceaser: Mitigating conflict-based cache attacks via encrypted-address and remapping," in *MICRO*, 2018.

[9] F. Canale *et al.*, "{SCARF}–a {Low-Latency} block cipher for secure {Cache-Randomization}," in *USENIX Security*, 2023, pp. 1937–1954.

[10] M. Werner *et al.*, "Scattercache: Thwarting cache attacks via cache set randomization," in *USENIX*, 2019.

[11] J. P. Thoma *et al.*, "{ClepsydraCache}–preventing cache attacks with {Time-Based} evictions," in *USENIX Security*, 2023, pp. 1991–2008.

[12] L. Li *et al.*, "Prefender: A prefetching defender against cache side channel attacks as a pretender," *IEEE Transactions on Computers*, 2024.

[13] F. Mosquera *et al.*, "Guard cache: Creating noisy side-channels," *IEEE CAL*, vol. 22, no. 2, pp. 97–100, 2023.

[14] W. Song *et al.*, "Randomized last-level caches are still vulnerable to cache side-channel attacks! but we can fix it," in *S&P*, 2021.

[15] T. Kessous and N. Gilboa, "Prune+ plumtree-finding eviction sets at scale," in *SP*. IEEE Computer Society, 2024, pp. 173–173.

[16] N. Shrivastava and S. R. Sarangi, "Toward an optimal countermeasure for cache side-channel attacks," *ESL*, vol. 15, no. 3, pp. 141–144, 2022.

[17] Z. N. Zhao *et al.*, "Last-level cache side-channel attacks are feasible in the modern public cloud," in *ASPLOS, Volume 2*, 2024, pp. 582–600.

[18] V. Rijmen *et al.*, "Optimised ANSI C code for the Rijndael cipher (now AES)," 2000. [Online]. Available: https://www.esat.kuleuven.be/cosic/pulications/article-154.pdf

[19] S. Vanstone, "Elliptic curve digital signature algorithm," *Submission to NIST*, 1992.

[20] M. Li *et al.*, "Treasurecache: Hiding cache evictions against side-channel attacks," *IEEE TDSC*, 2024.

[21] M. K. Qureshi, "New attacks and defense for encrypted-address cache," in *ISCA*, 2019, pp. 360–371.