Yet Another Mirage of Breaking MIRAGE: Debunking Occupancy-based Side-Channel Attacks on Fully Associative Randomized Caches

Chris Cao
University of Toronto
chrisj.cao@mail.utoronto.ca

Gururaj Saileshwar
University of Toronto
gururaj@cs.toronto.edu

***Abstract*—Recent work presented at USENIX Security 2025 (SEC'25) claims that occupancy-based attacks can recover AES keys from the MIRAGE randomized cache. In this paper, we examine these claims and find that they arise from a modeling flaw in the SEC'25 paper. Most critically, the SEC'25 paper's simulation of MIRAGE uses a constant seed to initialize the random number generator used for global evictions in MIRAGE, causing every AES encryption they trace to evict the same deterministic sequence of cache lines. This artificially creates a highly repeatable timing pattern that is not representative of a realistic implementation of MIRAGE, where eviction sequences vary randomly between encryptions. When we instead randomize the eviction seed for each run, reflecting realistic operation, the correlation between AES T-table accesses and attacker runtimes disappears, and the attack fails. These findings show that the reported leakage is an artifact of incorrect modeling, and not an actual vulnerability in MIRAGE.**

I. Introduction

MIRAGE [1] is a randomized cache design, proposed in 2021, that emulates a fully associative cache with globally random evictions, eliminating set-conflict cache side channels. It builds on theoretical foundations such as multiple randomized set indexing functions using block ciphers, and power-of-two-choices [2] based load-balancing, guaranteeing that set-associative evictions are practically impossible in a system's lifetime. Given these strong guarantees, several works have examined whether MIRAGE's security holds in practice.

In 2023, "*Are Randomized Caches Truly Random*" (ARCTR) [3] claimed to induce set-conflicts in MIRAGE, breaking its security guarantees. However, subsequent work [4] showed that these were the result of incorrect modeling by ARCTR, caused by a buggy cipher implementation, and that MIRAGE's security guarantees remained intact.

More recently, the SEC 2025 paper, "*Systematic Evaluation of Randomized Cache Designs against Cache Occupancy*" (RCO) [5], claims that MIRAGE is vulnerable to cache-occupancy-based side-channel attacks that can leak secret AES keys. Specifically, RCO (in Section 7), claims that the AES T-Table implementation can leak the AES key on MIRAGE via the cache occupancy side-channel, and that MIRAGE's fully associative eviction policy makes it more susceptible than other randomized caches. A subsequent paper at SEC 2025, "*SoK: So, You Think You Know All About Secure Randomized Caches?*" [6] reiterates these claims in its Figure 17. This paper examines whether these claims hold up or whether they are artifacts of modeling flaws like the ARCTR paper.

First, we attempted to reproduce the AES key recovery results reported in the RCO paper using their artifact[1], specifically the guessing entropy (GE) for an unknown victim key (Figure 10 in the RCO paper [5]). Using the RCO artifact, we initially could not reproduce their results.

Bug-1: GE Analysis Bug in RCO. Figure 1 compares the guessing entropy (GE) for an AES key (higher is better) for MIRAGE using RCO's raw data (RCO – red line) and our reproduced data (Reproduced – blue line), along with other randomized caches. In the original analysis (Figure 1a), RCO reports GE dropping below 80 after just 300 AES traces. In contrast, our reproduced data shows the same GE reduction occurs after 1400 traces.

Upon investigation, we identified a *bug* in the RCO artifact's GE analysis code. The bug stems from an incorrect indexing of AES traces used for the analysis (see Appendix A), which are spread over *six* trace files in RCO's raw data, due to which their code uses roughly 6× more traces than intended to calculate GE. This results in an under-reporting of the number of traces required to reach a given GE in the RCO paper. After fixing this bug in RCO's artifact[2], the analysis using RCO's raw data matches our naive reproduction (Figure 1b), with GE for MIRAGE now dropping after 1400 traces.

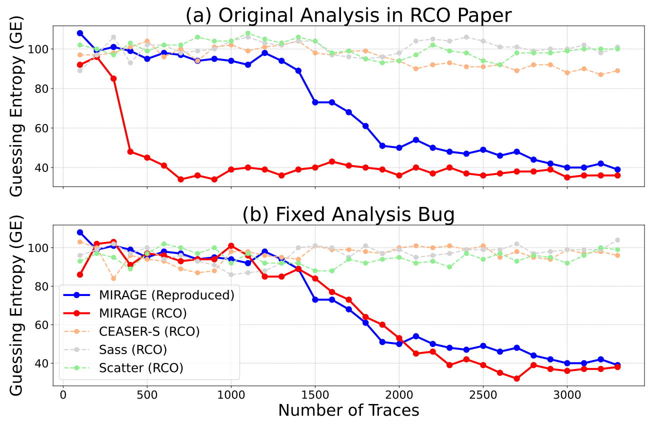


Fig. 1: **Guessing Entropy (GE) for an unknown AES key**, as the number of AES encryption traces increases. (a) Original analysis from RCO shows GE reduction for MIRAGE after 300 traces. (b) After fixing the analysis bug, GE reduction occurs after 1400 traces, consistent with our reproduction.

Next, we examine the validity of this leakage.

[1]We use the the code artifact with RCO's paper [5] - https://zenodo.org/records/14869981 (Github release v2.0, commit: fc07ea6).

[2]The code for our reproduction, with our bug fixes, is open-sourced at: https://github.com/sith-lab/yet-another-mirage-of-breaking-mirage

Bug-2: Global Eviction Modeling Bug in RCO. RCO attributes AES key leakage to last-round S-Box accesses in the AES encryption influencing cache occupancy. The attacker observes this by measuring access times to its own cached array, building templates for each key-byte value using a profiled key, and matching these templates against timings for an unknown key to recover it. However, this explanation ignores MIRAGE's **random global evictions**, which introduce noise in cache occupancy. MIRAGE randomly evicts an existing LLC line upon every new insertion. Hence, repeated AES encryptions of even the same plaintext and key evict different addresses and produce different occupancy (unrelated to the key), and the attacker's own accesses also evict its addresses randomly on each run, introducing further noise. Thus, an attacker would even find fingerprinting a key difficult (Figure 3), let alone leak the entire key byte-wise. Investigating this, we uncover a second *bug* in RCO's evaluation:

Modeling Flaw in RCO. RCO's simulations initialize the RNG used by MIRAGE's evictions with a **static seed** before each AES encryption, causing a **fixed** sequence of evictions on each AES run, not practical in a real attack.

RCO's simulations seed MIRAGE's RNG used for evictions with a **fixed value (42)** before each AES encryption, restarting each AES run from an identical state (see Appendix B). This produces a fixed eviction sequence on each AES encryption, not representative of MIRAGE. In a realistic modeling of MIRAGE with random global evictions, the final cache occupancy (**O**) after each AES encryption should be determined by both the victim's accesses (**V**) and the global random eviction sequence (**R**), $O \approx f(R, V)$. However, as RCO uses a constant RNG seed, producing a fixed eviction sequence in each run, the cache occupancy becomes a function of just victim accesses, $O \approx f(V)$, artificially creating correlations with the key.

To correctly model MIRAGE while restarting the simulation for each AES run, we **randomize the eviction RNG seed for each AES run** (e.g., using `std::random_device`[3] to seed the RNG). This models a realistic setting, where the attacker cannot reset the RNG to a constant state on each AES run. With this correction, correlations between attacker timings and keys disappear, and the victim AES key's guessing entropy remains high (above 90%), as shown in **Figure 4**.

Results with Correct Modeling. After randomizing the RNG seed for MIRAGE in each AES run, the victim key's guessing entropy (GE) remains high (>90%), see **Figure 4**. Thus, AES key leakage on MIRAGE is infeasible.

To summarize, we make the following contributions:

1) We show that the AES key guessing entropy remains high in MIRAGE, contradicting the RCO paper's claims.
2) We identify that RCO's usage of a fixed RNG seed for MIRAGE's evictions creates artificial correlations; randomizing the seed eliminates any observed leakage.

[3]Note that on some systems without a hardware-based entropy source, `std::random_device` can also result in a deterministic output [7]. Please validate that the source of randomness you use is truly random on your system.

II. BACKGROUND

A. The MIRAGE Cache

MIRAGE [1] is a randomized cache that prevents conflict-based side channels by emulating a fully-associative randomized cache: every eviction is global and selected uniformly at random from the entire cache. To support this, MIRAGE over-provisions invalid tags in each set and uses load-balancing via the Power-of-2-Choices to maintain available space, avoiding set-associative evictions. On a miss, the data-store victim is chosen randomly from all cache lines, its tag is located via a Reverse Pointer (RPTR) and removed, and the new tag is inserted via a Forward Pointer (FPTR). With an 8-way cache augmented with 6 extra ways (75% extra) in the tag-store, MIRAGE guarantees the probability of a set-associative eviction is once in 10^{34} cache installs, an event that takes 10^{17} years to occur, making set-conflicts practically impossible in a system's lifetime and eliminating conflict-based attacks.

B. Cache Occupancy Attacks on Mirage

Cache occupancy attacks measure changes in the overall occupancy of a shared cache, rather than targeting specific sets as in set-conflict attacks like Prime+Probe. Because they exploit aggregate cache usage, all randomized caches without explicit cache partitioning, such as CEASER, ScatterCache, and Mirage in principle, leak some information via cache occupancy. In fact, MIRAGE's threat model explicitly claims to not protect against such occupancy-based channels. While covert channels between two colluding processes can be naively constructed by modulating the cache occupancy, the RCO [5] paper claims the existence of a stronger side-channel on MIRAGE: leaking AES keys via cache occupancy effects.

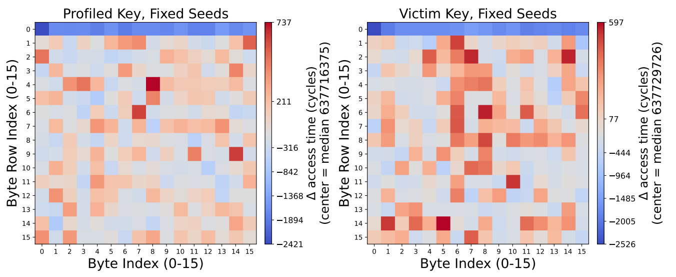
The RCO paper claims that a victim using an AES T-Table implementation, can be forced to leak the key in MIRAGE, by a spy first priming the LLC to a chosen occupancy, letting the victim run one encryption, then timing accesses to the attacker's own cache lines. By correlating these timings with last-round T-table accesses for guessed keys, they report low guessing entropy for AES keys (lower than 30%), and claim full 128-bit AES key recovery on MIRAGE within a few hours. This paper examines these claims of RCO.

III. ANALYZING CLAIMS OF OCCUPANCY-BASED SIDE-CHANNEL ATTACKS ON MIRAGE

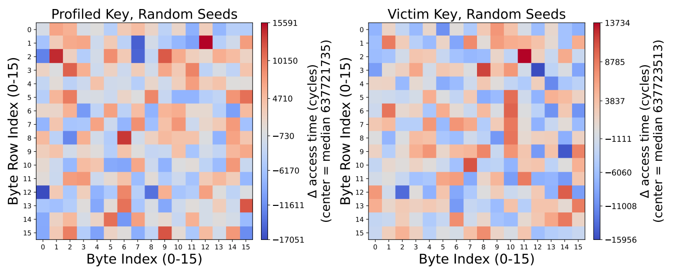
Using the authors' publicly released artifact, we first reproduced the results in Figure 10 of the RCO paper, as shown in Figure 1. Once we were able to reproduce the trend, that the guessing entropy for an unknown AES key reduces after 1400 encryptions, we analyze the root cause of the attack: that the execution time for an attacker accessing its own array is a function of the cache occupancy of the AES encryption, which is in turn a function of the victim's AES key.

A. Pitfall-1: Fixed Sequence of Evictions Modeled by RCO

RCO Attack Root Cause. The RCO [5] paper claims that the T-Table implementation of AES running on MIRAGE [1] can leak the secret key through a cache occupancy attack. The



(a) Fixed Seed for Global Evictions (RCO [5])



(b) Random Seed for Global Evictions (our fix)

Fig. 2: Heatmap of access times for the attacker to iterate through its array. We bin the access times, based on the 256 possible T-table entries accessed in the last round. The 256 bins are represented in the 16 x 16 matrix. (a) With Fixed Seed for Global Eviction, as used in RCO [5], there is strong correlation between the heatmaps for profiled key and victim key. (b) After our fix, with Random Seeds for Global Eviction, the correlations disappear, showing it is infeasible to guess victim AES keys.

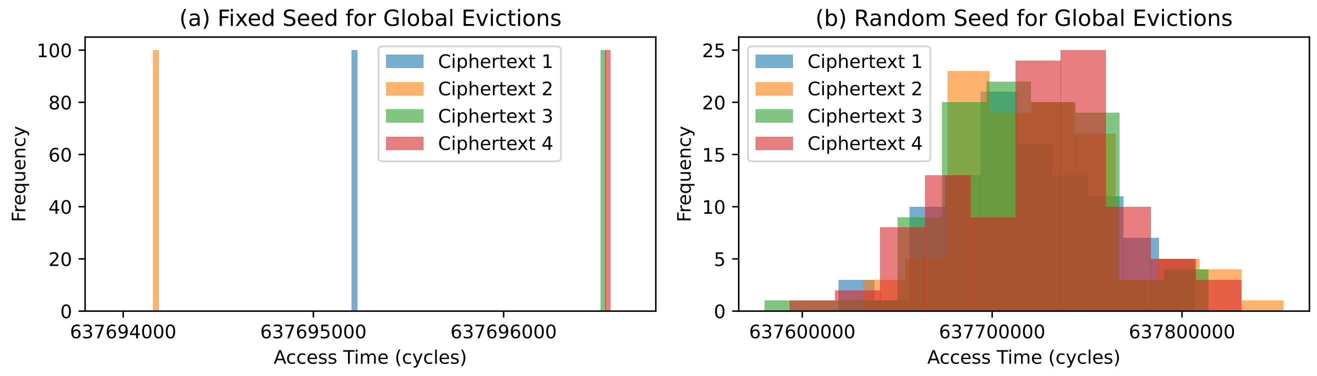


Fig. 3: Access Times for the attacker with (a) original RCO implementation and (b) our bug fix for four sample plaintext-ciphertext pairs with the same key. (a) The original implementation uses a Fixed Seed (42) for Global Evictions in each AES encryption. (b) Our bug fix initializes the RNG used for Global Evictions, with a random seed (e.g., using `time()`), for each AES encryption, mimicking a real system where the seed cannot be reset to a static value each time. Once we address this issue, the different encryptions are indistinguishable based on access times to the attacker's array.

root cause of the leakage claimed by RCO is that the sequence of T-Table accesses in the last round of AES encryption, which depends on the secret key, can impact the cache occupancy in MIRAGE. Therefore, the execution time for an attacker to access a large cached array, parts of which may have been evicted by the AES encryption, can leak the cache occupancy, and therefore the secret key.

RCO Attack Mechanism. To perform this attack, RCO [5] creates a template for the execution times with all possible T-table entries accessed in the last round (T_1 to T_{255}), by using a known key (*profiled key*). The attacker measures the execution time to access its own cached array that occupies 50% of the MIRAGE cache, using randomly generated plaintext-ciphertext pairs, and creates the template (T_1 to T_{255}) by averaging the access times for T_n= SBOX-Inv$(K \oplus CT)$, where K and CT are bytes of the last-round key and ciphertext. Such templates can be created for each of the bytes of the round-key (0 to 15). Later, for an unknown victim key, by creating a similar template using a guessed key and random plaintext-ciphertext pairs, the attacker identifies likely key values having the highest correlation with the profiled template.

Reproducing RCO's Root-Cause. To validate RCO's leakage, we try to reproduce its root cause, the templates with correlation between profiled and victim keys. Figure 2a shows a heatmap visualizing the template built by the attacker, where each entry represents one of the 256 possible T-table entries (arranged as a 16×16 matrix) and the cell color encodes the attacker's average access time, when that entry is accessed in the last round of AES by the victim. If the profiled-key heatmap (attacker's template) closely matches the heatmap with the guessed victim-key, the guess is likely to be the correct key. For simplicity, instead of T_n = SBOX-INV$(K \oplus CT)$, we use $T_n = K \oplus CT$ for our bins, since SBOX-INV is just a lookup table, and visualize a single heatmap, averaging the heatmaps of key bytes 0 to 15.

As shown in Figure 2(a), when we generate the heatmaps by using the RCO artifact [5] there is a clear correlation between the templates of the profiled key and the guessed victim key. This correlation can allow the attacker to leak the victim key since the templates will be highly correlated when the guessed key is actually the correct victim key.

RCO's Bug: Fixed Sequence of Global Evictions. To investigate the source of the observed correlation, we measured attacker access times after victim AES encryptions for four randomly chosen plaintext-ciphertext pairs, each repeated 100 times. Each encryption has a distinct last-round AES T-Table access sequence. Figure Figure 3(a) shows the histogram of attacker access times. CT1, CT2 and CT3 differ in access times (although CT3 and CT4 overlap), highlighting that the access times are indeed correlated with T-Table accesses. However, notably, all 100 repetitions of each encryption produce identical access-time measurements, despite global evictions being intended as random.

This reveals a bug in RCO: the global eviction RNG is seeded with a static value (42). As a result, each AES execution follows the same fixed eviction sequence, making MIRAGE's behavior deterministic. In a real implementation, the attacker cannot reset the RNG between runs, so the deterministic behavior is unrealistic and artificially creates correlations between access times and AES T-Table sequences.

Our Fix: Accurate Modeling with Random Seed. The cache occupancy (O) in MIRAGE is a function of both the Victim accesses (V) and the Global Eviction decisions (GE), i.e., $O \approx f(V, GE)$. RCO incorrectly assumes a fixed sequence of GE for each encryption, making the fingerprinting of V using O measurements possible. We fix this bug in RCO, by changing the fixed seed for global evictions to a randomly chosen seed for each AES encryption simulation, representative of real systems, where the Global Eviction is performed by a RNG whose state cannot be reset by the attacker. After this bug fix, we see that the O is now strongly impacted by the GE in addition to the V. As shown in Figure 3(b), with a random seed for each encryption, the attacker's access times show random variations of the order of 100,000 cycles, due to the global evictions unpredictably evicting the attacker's own lines during its measurement phase. These overwhelm minor occupancy differences caused by the victim which varied timings by few 1000 cycles in Figure 3(a).

After our fix, when we use a random seed for each AES encryption in Figure 2(b), the correlation between profiled and victim keys disappears, making the heatmaps virtually unrelated, eliminating the signal required for key leakage. This realistic modeling of MIRAGE's global evictions removes the attack's timing signals, preventing AES key leakage on MIRAGE via occupancy attacks. We provide the code for this bug fix in Appendix B.

B. *Pitfall-2: Unrealistic L1 Cache Configuration*

The RCO paper claims to use a 512 **kilobyte** L1 cache, as per Section 7.1 of their paper [5]. However, their code artifact[4] uses a 512 **byte** L1 cache by default which is not representative of real systems which typically possess atleast 64KB L1 Cache. Modeling a smaller L1 cache, such as 512 byte L1 cache, can inflate the L1 cache misses and LLC accesses, compared to a 64KB L1 Cache which can provide hits for *all* T-table accesses after the first round. Thus, a 512 byte L1 cache can overestimate the attack success. We confirmed that after updating the initialized seed to be random, regardless of the L1 cache size being 512-byte like in the RCO artifact or a more realistic 64KB, there is a lack of correlation between the heatmaps of the profiled and victim key templates, similar to Figure 2 (b), indicating that AES key leakage is impractical in MIRAGE.

[4]We refer to the code artifact version 2.0 released by the authors on Zenodo - https://zenodo.org/records/14869981

C. *Guessing AES Key after Fixing RCO's Modeling Issues*

We evaluate the Guessing Entropy (GE) for a victim's AES key, using the template for a profiled AES key similar to RCO, using the formula: $GE = \sum_{i=0}^{15} log_2(R_i)$, where R_i is the rank of the correct guess for key byte i. Figure 4 shows the guessing entropy (GE) for Mirage based on their artifact (**reproduced**), and with our fixes of RCO's modeling issues where we use a random seed to initialize the RNG for global evictions (**bug fix**). After our bug fixes, we see that **Mirage has high GE above 90%**, even after thousands of AES encryptions, demonstrating that it is **resilient to brute-force key guessing attacks on AES** when modeled correctly.

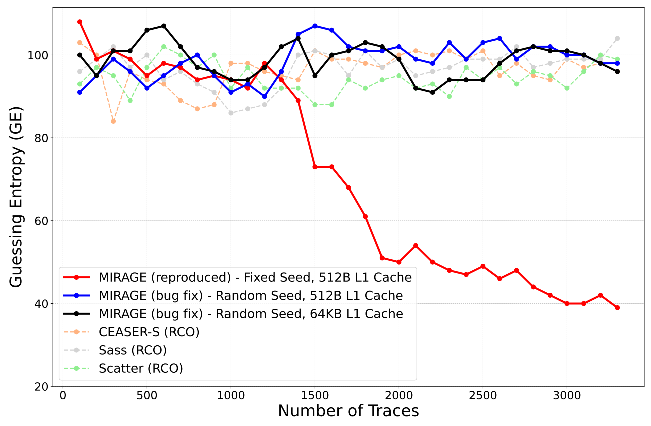


Fig. 4: **Guessing Entropy (GE) for an unknown AES key, after our fixes**, as number of AES encryptions increases. MIRAGE has a high GE above 90% even after thousands of AES encryptions, showing no leakage, once the Eviction RNG is initialized with a Random Seed, with both a 512B L1 Cache (like RCO's artifact) and a more realistic 64KB L1 Cache.

IV. Concluding Remarks

Our analysis shows that, when modeled faithfully, MIRAGE remains resilient to occupancy-based side-channel attacks aiming to recover AES keys. The results reported in the RCO paper [5] arise from unrealistic modeling, most notably the use of a deterministic sequence of global evictions, across multiple AES encryptions, which do not reflect MIRAGE's design. We encourage the authors of the RCO paper [5] and the SoK paper on randomized caches [6] to revisit their conclusions in light of these findings. Our code is open-sourced at https://github.com/sith-lab/yet-another-mirage-of-breaking-mirage.

V. Acknowledgment

We thank Moinuddin Qureshi and Tom Ristenpart for their helpful feedback on an earlier draft of this work.

References

[1] G. Saileshwar and M. Qureshi, “MIRAGE: Mitigating Conflict-Based Cache Attacks with a Practical Fully-Associative Design,” in *USENIX Security*, 2021.

[2] M. Mitzenmacher, “The power of two choices in randomized load balancing,” *IEEE Trans. on Parallel and Distributed Systems*, 2001.

[3] A. Chakraborty, S. Bhattacharya, S. Saha, and D. Mukhopadhyay, “Are Randomized Caches Truly Random? Formal Analysis of Randomized-Partitioned Caches,” in *HPCA*, 2023. [Online]. Available: https://github.com/SEAL-IIT-KGP/randCache

[4] G. Saileshwar and M. Qureshi, “The mirage of breaking mirage: Analyzing the modeling pitfalls in emerging “attacks” on mirage,” *IEEE Computer Architecture Letters*, vol. 22, no. 2, pp. 121–124, 2023.

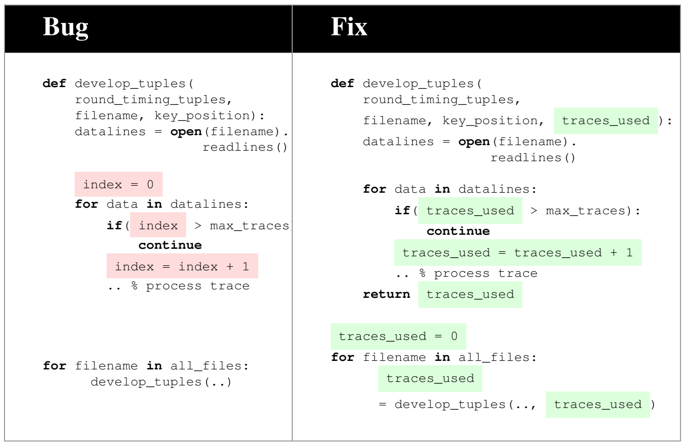
[5] A. Chakraborty, N. Mishra, S. Saha, S. Bhattacharya, and D. Mukhopadhyay, “Systematic evaluation of randomized cache designs against cache occupancy,” in *35th USENIX Security Symposium (USENIX Security 25)*, 2025.

[6] A. Bhatla, H. R. Bhavsar, S. Saha, and B. Panda, “Sok: So, you think you know all about secure randomized caches?” in *35th USENIX Security Symposium (USENIX Security 25)*, 2025.

[7] cppreference.com, “Random device,” https://en.cppreference.com/w/cpp/numeric/random/random_device.html, 2025.

Appendix

Appendix A
Bug-1: Guessing Entropy Analysis Bug in RCO



Bug

```
def develop_tuples(
    round_timing_tuples,
    filename, key_position):
    datalines = open(filename).
                readlines()

    index = 0
    for data in datalines:
        if( index  > max_traces
            continue
        index = index + 1
        .. % process trace



def filename in all_files:
    develop_tuples(..)
```

Fix

```
def develop_tuples(
    round_timing_tuples,
    filename, key_position,  traces_used ):
    datalines = open(filename).
                readlines()

    for data in datalines:
        if( traces_used  > max_traces):
            continue
        traces_used = traces_used + 1
        .. % process trace
    return  traces_used

traces_used = 0
for filename in all_files:
      traces_used
      = develop_tuples(..,  traces_used )
```

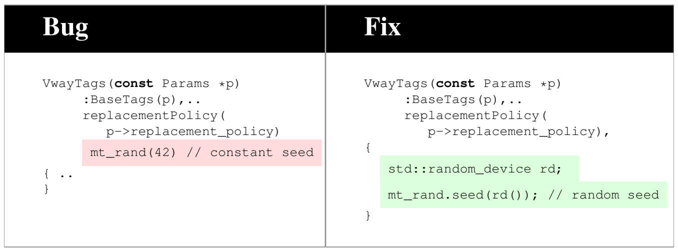
Fig. 5: Bug in RCO’s Guessing Entropy Code

Figure 5 illustrates the bug in RCO’s Guessing Entropy Analysis code. In the **original implementation (left)**, RCO processes multiple trace files (up to 6 AES trace files) to calculate the guessing entropy, creating tuples for computing guessing entropy. The bug arises because the check against `max_traces` is applied locally per trace file, rather than globally across all trace files. As a result, the code uses ends up using 6 × `max_traces`, instead of the intended `max_traces`, causing the number of AES traces needed for a given GE to be under-reported by a factor of six.

In our **bug fix (right)**, we introduce a `traces_used` variable that tracks the total number of traces used across all files. This ensures that the `max_traces` limit is enforced globally correcting this counting error.

Our reproduction stored all the traces in a single file and thus was not affected by this bug. The discrepancy between the reproduced results and RCO’s reported results helped us identify this issue. The code for this bug fix is available here.

Appendix B
Bug-2: Global Eviction RNG Seed Bug in RCO



Bug

```
VwayTags(const Params *p)
      :BaseTags(p),..
      replacementPolicy(
         p->replacementPolicy)
      mt_rand(42) // constant seed
{ ..
}
```

Fix

```
VwayTags(const Params *p)
      :BaseTags(p),..
      replacementPolicy(
         p->replacement_policy),
{
   std::random_device rd;
   mt_rand.seed(rd()); // random seed
}
```

Fig. 6: Bug in RCO’s modeling of MIRAGE’s Evictions

Figure 6 illustrates the bug in RCO’s modeling of MIRAGE’s evictions. In the **original code (left)**, the RNG used for global tag evictions is initialized with a **fixed seed** (42). Due to RCO’s attack methodology of restarting the simulation for each AES encryption, this causes every AES simulation to artificially produce the same eviction sequence. The **bug fix (right)** replaces the fixed seed with a random seed unpredictable to an attacker generated via `std::random_device`, producing a **different global eviction sequence** for each AES simulation. This models a realistic behavior of an attack on MIRAGE. Our patch for the bug fix is available at this link.