LeakyRand: An Efficient High-fidelity Covert Channel in Fully Associative Last-level Caches with Random Eviction

YASHIKA VERMA, DEBADATTA MISHRA, and MAINAK CHAUDHURI, Indian Institute of Technology, Kanpur, India

Recent studies on secure last-level cache (LLC) have advocated the fully associative organization to defend against conflict-based side-channel attacks. In a fully associative LLC, an attacker cannot extract any information about the cache location of the addresses evicted due to cross-core conflict. Use of the random replacement policy further guards against any deterministic eviction patterns. However, the fully associative LLC design remains vulnerable against timing-based covert channel attacks. In this paper, we present *LeakyRand*, a high-bandwidth covert communication mechanism that exploits the fully associative LLC with random replacement while guaranteeing an ultra-low bit error rate (BER). Our proposal is a union of three unique contributions. First, we present an efficient algorithm with strong analytical guarantees that enables the attacker to quickly occupy nearly the whole LLC with high probability, a prerequisite for achieving high bandwidth and high fidelity. Second, we present a novel covert communication protocol that allows the attacker to maintain high LLC occupancy. Third, our proposal detects error syndromes and efficiently takes corrective measures leading to an ultra-low expected BER. Exploiting a 2 MB fully associative LLC with random replacement policy to set up a covert channel, LeakyRand experiences an average BER of 10^{-4} or less while offering 3.3× to 5.7× higher channel bandwidth compared to the recently proposed Stochastic Prime+Probe attack. We also demonstrate that LeakyRand can be adopted to mount high-precision fine-grain fingerprinting attacks.

CCS Concepts: • **Security and privacy → Hardware attacks and countermeasures**.

Additional Key Words and Phrases: Fully associative cache, Random replacement, Covert channel, Timing attack

1 Introduction

Attacks exploiting the timing channels in the last-level cache (LLC) shared among the cores of a chip-multiprocessor have raised serious security concerns in general-purpose processors [10, 11, 17, 22, 25, 35] as well as embedded system-on-chips found in mobile and notebook devices [12, 14–16, 36]. Communication between two isolated execution entities (e.g., receiver and sender processes) in a clandestine manner using covert channels is one of the practically demonstrated ways of exploiting the cache timing channel. Conflict-based covert channels built on top of Prime+Probe rely on controlled cache contention [17, 20, 33]. These channels have been shown to be very effective in set-associative caches with deterministic replacement policies. The following three steps are used to set up such a channel. (*a*) *Cache region identification:* The receiver and the sender operate on an identified region of the LLC for communication. This region could be an LLC set or a group of LLC sets. This LLC region is filled (or primed) by the receiver with a set R of addresses, referred to as an eviction set [17, 21, 23, 27]. (*b*) *Communication:* The sender encodes a message bit 1 in the form of

Authors' Contact Information: Yashika Verma, yashikav@cse.iitk.ac.in; Debadatta Mishra, deba@cse.iitk.ac.in; Mainak Chaudhuri, mainakc@cse.iitk.ac.in, Indian Institute of Technology, Kanpur, India.

'CASES 2025', Taipei, Taiwan

ACM ISBN 978-1-4503-XXXX-X/2018/06
https://doi.org/XXXXXXX.XXXXXXX

some conflict-induced disturbance in the identified LLC region with the help of a set S of addresses. In techniques based on Prime+Probe, the eviction sets R and S satisfy $R \bigcap S = \phi$. Since the accesses to S necessarily evict some elements of R, the receiver can detect a message bit 1 by probing the elements of R and identifying any missing element with the help of the probe latency. If no element is missing, the message bit is 0. (c) *Cache region maintenance:* Before communication of each bit, the identified LLC region should be in a fixed deterministic pre-negotiated state that contains the set R.

The vulnerabilities of set-associative caches with deterministic replacement policies have been extensively studied [5, 8, 21, 22, 32]. As a countermeasure, recent research has advocated practically implementable fully associative LLC organizations employing the random replacement policy [2, 25]. The fully associative LLC design eliminates leakage of set index bits, while the three steps discussed above becomes extremely challenging with random replacement. The random replacement policy makes it very difficult for the receiver to efficiently prime the identified cache region, to accurately detect the communicated bit, and to quickly undo the disturbance caused by communication.

The central contribution of this paper is to demonstrate that even a fully associative LLC with the random replacement policy can be exploited to carry out covert communication at a reasonably high bandwidth with an ultra-low bit error rate. Formally, we design two processes referred to as the sender and the receiver with no shared memory executing on two cores of a chip-multiprocessor with a fully associative LLC shared between the cores such that the sender communicates a bit string to the receiver by encoding the bits using LLC events. The end-result of our design is *LeakyRand*, a novel error-resilient high-bandwidth covert channel, that dynamically detects communication error syndromes and takes corrective measures to restrict future errors. The design of LeakyRand involves a fresh relook at cache region identification, covert communication, and cache region maintenance. In the following, we first discuss why these three traditional steps do not work well in a fully associative LLC with the random replacement policy (Section 1.1). Next, we briefly introduce our proposal (Section 1.2).

1.1 Covert Channel in Fully Associative LLC

Demarcation of a cache region at the set granularity requires the receiver to prime the entire fully associative LLC (the only LLC set) using an eviction set that has at least as many blocks as the cache has, thus making the process slow. Random replacement policy makes this process non-deterministic because the receiver can occupy only a fraction of the LLC with a certain probability. If the receiver fails to occupy a good portion of the LLC, the sender needs to create a large enough disturbance (for sending a 1 bit) that intersects with the receiver's occupied LLC portion for error-free communication. Thus, we have a new error-bandwidth trade-off. The cache region maintenance step may, in the worst case, need to resort to the heavy-weight Prime step.

In Stochastic Prime+Probe (SPP) [28], a recently proposed covert channel exploiting a fully associative cache with random replacement, the receiver primes the cache with a set R of block addresses such that $|R|$ is equal to the number of blocks in the cache. For example, in a 2 MB cache with 64-byte blocks, the cache has $c = 32\text{K}$ blocks and $R = \{r_1, r_2, \ldots, r_c\}$ has c block addresses. Due to the random replacement policy, the receiver can replace r_i when accessing r_j with $i < j$, leading to a final cache occupancy that is less than 32K blocks. We show in Section 3.3 that the receiver, on average, occupies only 63% of the cache requiring the sender to use a large set S of block addresses when communicating a 1 bit. For example, $|S| = 0.4c = 12.8\text{K}$ blocks for a 2 MB cache offers acceptable communication fidelity, as we will see in Section 5. Communication is error-free if access to S evicts a detectable number of blocks of R. In the Probe step, the receiver accesses R to detect misses, but fails to maintain a deterministic LLC occupancy due to the random replacement policy. For example, in a 2 MB LLC, when sending a 1 bit, the sender replaces a number of blocks belonging to R. When the receiver probes R to infer this bit, it can replace other blocks of R to fill the blocks that the sender replaced, leading to an average LLC occupancy less than 63%.

To achieve acceptable error rates, SPP employs error control techniques such as differential signaling and repetition coding. Differential signaling requires sending a reference bit 0 before each data bit so that the receiver can detect the presence (absence) of disturbance due to a data bit 1 (0) with better fidelity, but the receiver needs to probe R twice to receive the reference bit and the data bit. Repetition coding requires that multiple copies of the same data bit be sent. Thus, in our example of 2 MB cache, even without repetition coding, communication of 0 and 1 bits respectively requires 64K and 76.8K cache accesses together from the receiver and the sender leading to poor average channel bandwidth. Thus, SPP suffers from three shortcomings, namely, (a) due to the receiver's inability to occupy a large portion of the LLC, the sender needs to access a large number of cache blocks for communicating a 1 bit, (b) every data bit needs to be augmented with heavy-weight error control codes, and (c) the receiver fails to maintain a deterministic LLC occupancy. In summary, straightforward adoption of the Prime+Probe technique, as in SPP, to fully associative caches with random replacement leads to covert channels that cannot achieve high fidelity and high bandwidth simultaneously.

1.2 Our Approach and Contributions

Construction of a high-bandwidth error-resilient covert channel for fully associative caches with random replacement policy requires novel redesign of the cache region identification, communication, and cache region maintenance steps. We pose cache region identification as an optimization problem which attempts to minimize the cycles needed by the receiver to achieve a target expected LLC occupancy. We develop a procedure to automatically synthesize reasonably good region identification algorithms using which the receiver quickly occupies 99% or more of the LLC. The primary advantage of having the receiver control a large portion of the LLC is that even a small disturbance in the LLC introduced by the sender is noticed by the receiver during the Probe step with high probability. We propose a novel communication protocol that does not require any shared memory between receiver and sender, has low overhead, and lets the receiver maintain the large expected LLC occupancy with minimum additional work. Communication errors arise when the sender's disturbance does not intersect with the receiver's occupied cache region. Our proposal periodically detects the syndromes of such errors and invokes corrective measures to restrict the number of future errors. Moreover, our proposal is designed to have in-built resilience to external noise.

Putting it all together, we present *LeakyRand*, an efficient error-resilient covert channel (Sections 3 and 4). Since no commercial processor has a fully associative LLC today, we evaluate LeakyRand using a detailed microarchitecture simulation model (Section 5). We use the MIRAGE design to model a secure fully associative LLC with the random replacement policy [25]. Our results show that the LeakyRand channel, when exploiting a 2 MB LLC, offers an average channel bandwidth of 26.4 Kbps which is 3.3× to 5.7× higher than a bandwidth-optimized implementation of SPP running without repetition coding. The LeakyRand channel experiences a bit error rate of 10^{-4} or less.

2 Related Work

A large body of work has explored information leakage through cache timing channels. In the following, we discuss the studies that are most relevant to our covert channel proposal.

Conflict-based covert channels use the Prime+Probe technique to exploit conflicts between sender's and receiver's LLC accesses [17, 20, 27, 28, 33]. A subset of these proposals relies on the deterministic eviction order of the least recently used (LRU) replacement policy [17, 33] or other age-based policies [20]. These proposals lose robustness in the presence of the random replacement policy. While Stochastic Prime+Probe (SPP) augments Prime+Probe with error control codes [28], this proposal fails to guarantee large LLC occupancy for the receiver and suffers from poor bandwidth in fully associative caches with random replacement policy, as already discussed. A recent study has proposed a covert channel fundamentally similar to SPP except that the sender induces two different levels of non-zero disturbance to communicate 0 and 1 bits [7]. In contrast to these proposals, our proposal *guarantees* large expected LLC occupancy for the receiver, helps the receiver maintain this large occupancy by introducing a novel communication protocol different from Prime+Probe, and periodically invokes low-overhead error syndrome detection and correction procedures to achieve ultra-low bit error rates and high bandwidth. The conflict-based attacks on the randomized skewed-associative caches have employed cache flush instructions to enable repeated use of a small partially congruent eviction set [27]. Our proposal optionally makes use of the cache flush instructions for a very different purpose, namely, to help the receiver establish and retain a large LLC occupancy. We show that our proposal also works without cache flush instructions.

The proposals on *shared memory-based covert channels* rely on read-only memory locations shared between the sender and receiver processes for communication [6, 10, 24, 34, 35]. In the Flush+Reload attack, the receiver flushes the shared cache block, waits for the sender to read the block (to transmit a 1 bit), and then measures the time to read the block to decode the transmitted message bit [35]. The first two steps of the Flush+Flush attack are same as the Flush+Reload attack, but in the last step, the receiver, instead of reading the block, flushes the block and measures the latency of the flush operation to infer the message bit [10]. In the Reload+Refresh attack, the receiver relies on the change in the victimization priority of the shared block due to a cache hit from the sender to the block (encoding a 1 bit) to decode the communicated bit [6]. This attack fails to work in the presence of random replacement policy. The time difference in accessing a shared block in E vs. S state has also been exploited to design covert channels [34]. The Streamline channel does not rely on flush operations, but allows the receiver and sender to communicate by streaming through a large shared read-only array [24]. The Streamline attack relies on an age-based LLC replacement policy which helps devise access patterns of the receiver/sender to minimize the probability that a block representing an already sent bit gets replaced before the bit is received. This probability becomes sizable in the presence of the random replacement policy making Streamline significantly less robust. In contrast to these proposals, our proposal does not rely on memory locations shared between the sender and the receiver. We present two different implementations of our proposal, one using the cache line flush instruction and the other without it.

The proposals on *occupancy-based covert channels* estimate the footprint of a process in a shared cache to infer sensitive information [7, 26]. Our proposal uses occupancy measurement in the cache region identification step to help the receiver occupy a large fraction of the LLC. However, the occupancy measurement procedure is crafted differently to work reliably with random replacement.

3 Design of LeakyRand

We discuss our baseline architecture and the threat model in Section 3.1. Section 3.2 presents an overview of the LeakyRand design. Sections 3.3 and 3.4 respectively discuss the cache region identification algorithm and the error-resilient communication protocol of LeakyRand.

3.1 Baseline Architecture and Threat Model

The LeakyRand covert channel is designed to work in a generic chip-multiprocessor with multiple cores. In this section, we assume the support for a user-level cache line flush instruction (e.g., `clflush` of x86 ISA). In Section 4, we show how our proposal works without a cache line flush instruction. Each core is assumed to have a private cache hierarchy and the LLC is shared by all the cores. The LLC has a fully associative organization and exercises the random replacement policy.

The random replacement policy has two possible implementations. In one implementation, invalid blocks are filled before invoking the replacement policy to maximize cache space utilization. In the other implementation, a random way is selected even if invalid ways are present in the cache. The second implementation is simpler because invalid ways need not be tracked, but it may suffer from performance degradation in the scenarios where a buffer is iteratively invalidated from the LLC and then reaccessed multiple times for computation. A skeleton code is shown in Algorithm 1. This iterative pattern is observed in applications with (*i*) DMA copies triggered by iterative I/O

Algorithm 1: Iterative invalidation and reuse of buffer A

```
Int $i$, $j$, num_sharing_episodes, num_compute_iterations
Array A
for $i = 1$ to num_sharing_episodes do
    Invalidate A from LLC
    for $j = 1$ to num_compute_iterations do
        Use A in computation
    end
end
```

operations, (*ii*) read-write sharing between threads scheduled on different CPU sockets, or (*iii*) data sharing between writer threads scheduled on GPU(s) and reader threads scheduled on CPU. Figure 1 shows the LLC miss count of the second implementation of random replacement normalized to the first. The LLC misses are counted for a number of inner loop iterations of the skeleton code shown in Algorithm 1 running on a processor model with a 2 MB LLC.[1] The size of the array A is varied from 256 KB to 2 MB and `num_compute_iterations` is varied from 2 to 8. In terms of LLC miss count, we see a significant advantage of filling the invalid ways first. Therefore, we assume that the invalid ways are filled first in the rest of this section. Section 4 discusses how LeakyRand could be adopted to the other (less optimized) implementation of the random replacement policy.

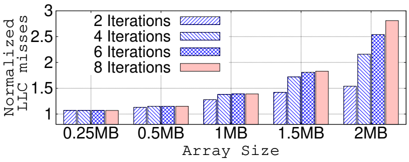


Fig. 1. LLC miss count without fill preference for invalid ways normalized to with fill preference

The threat model considered in this study involves two processes leaking sensitive information through a covert channel constructed by exploiting a shared cache level. The communication is one-way from the sender process (or trojan) to the receiver process (or spy). In this study, the sender and the receiver processes are assumed to be co-scheduled on two cores of a chip-multiprocessor socket exploiting the socket's shared LLC to set up the covert channel.

3.2 Overview of the LeakyRand Design

The LeakyRand design introduces three novel contributions to address the shortcomings of Stochastic Prime+Probe (SPP) [28] discussed in Section 1.1. Figure 2 presents a schematic overview of the LeakyRand design. The first contribution is a cache region identification algorithm designed to *quickly* occupy a large fraction of the LLC with a *deterministically known* set of blocks (CRFill and CRProbe steps shown in Figure 2(a)). The CRFill step uses a short LLC access sequence to achieve the expected target LLC occupancy. The CRProbe step counts the number of occupied LLC blocks, flushes them, and fills this space using a deterministic set of blocks with probability close to one. Section 3.3 discusses our cache region identification algorithm. To the best of our knowledge, this is the first study to propose an algorithm supported by strong analytical guarantees for efficiently occupying a large portion of a fully associative LLC exercising the random replacement policy.

The receiver needs to maintain high LLC occupancy throughout the communication phase. The second contribution of LeakyRand is a novel Evict+Flush+Probe communication protocol that helps the receiver achieve this with no additional work (Figure 2(b)). In this communication protocol, the sender creates invalid LLC ways that the receiver uses to reallocate the blocks that are evicted due to sender's disturbance during communication of a 1 bit. This novel communication protocol plays a key role in achieving a much larger bandwidth at a much lower bit error rate compared to SPP, which relies on traditional Prime+Probe. We discuss the communication protocol in Section 3.4.

[1] Our simulation environment is discussed in Section 5.

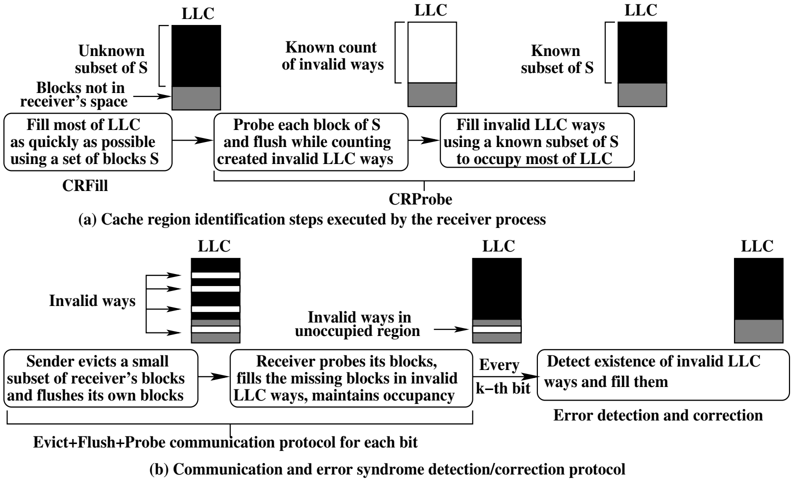

LLC
Unknown subset of S
Blocks not in receiver's space
Known count of invalid ways
Known subset of S
Fill most of LLC as quickly as possible using a set of blocks S
Probe each block of S and flush while counting created invalid LLC ways
Fill invalid LLC ways using a known subset of S to occupy most of LLC
CRFill
CRProbe
(a) Cache region identification steps executed by the receiver process
Invalid ways
Invalid ways in unoccupied region
Sender evicts a small subset of receiver's blocks and flushes its own blocks
Receiver probes its blocks, fills the missing blocks in invalid LLC ways, maintains occupancy
Every k–th bit
Detect existence of invalid LLC ways and fill them
Error detection and correction
Evict+Flush+Probe communication protocol for each bit
(b) Communication and error syndrome detection/correction protocol

Fig. 2. Overview of the LeakyRand design with LLC states shown along the flow

Communication errors arise due to the non-determinism inherent in the random replacement policy. When the sender fails to evict any of the receiver's blocks while transmitting a 1 bit, the bit is received as 0. Thanks to the large LLC occupancy of the receiver, the probability of error increases slowly as more bits are communicated. As a result, unlike SPP which needs error control codes for every data bit, LeakyRand only needs to take corrective measures periodically. The third contribution of the LeakyRand design is a low-overhead algorithm to periodically detect error syndromes and correct the LLC state so that errors do not keep accumulating (Figure 2(b)). Sections 3.4.3 and 3.4.4 discuss the error detection and correction procedures.

3.3 Cache Region Identification

In the cache region identification step, the receiver must occupy a *large enough region* of the fully associative LLC using a *deterministically known* set of blocks. This allows the sender to use a small disturbance during the communication of a 1 bit and the receiver to detect the bit with high probability. The random replacement policy makes it non-trivial to quickly replace the existing LLC blocks for achieving a target LLC occupancy close to 100%. In this section, we consider a set of unique block addresses and synthesize a sequence of accesses using those block addresses to efficiently achieve the target LLC occupancy. We refer to the set of block addresses as the *Occupancy Set* and the generated access sequence as the *Occupancy Sequence*. For example, an Occupancy Set of size three would have three block addresses $\{a, b, c\}$, while an Occupancy Sequence is an access sequence built using these three block addresses e.g., $< a, b, a, c, b, c, a >$.

The cache region (CR) identification procedure has two steps, namely CRFill and CRProbe. In the CRFill step, the receiver executes the synthesized Occupancy Sequence in an attempt to fill the cache with addresses from the Occupancy Set. Therefore, minimizing the length of the Occupancy Sequence optimizes the CRFill step. The second step is CRProbe, in which the receiver accesses the addresses in the Occupancy Set to compute the achieved LLC occupancy. Minimizing the Occupancy Set size optimizes the CRProbe step. Thus, the cache region identification procedure should achieve the target LLC occupancy with the smallest Occupancy Set and Occupancy Sequence.

3.3.1 CRFill Step. We begin the search for a high-performance CRFill algorithm by stating an important relationship between the number of LLC misses and the expected LLC occupancy. Based on this relationship, we explore two alternate strategies for designing the CRFill step, one that minimizes the Occupancy Set size and the other that minimizes the Occupancy Sequence length. These strategies do not optimize the Occupancy Set and the Occupancy Sequence simultaneously. However, they establish that a simple heuristic for designing a high-performance CRFill step would be to iterate over a collection of small Occupancy Sets. Finally, we propose two algorithms to synthesize Occupancy Sequences that are built by iterating over a single Occupancy Set (static algorithm) or a collection of Occupancy Sets with growing sizes (dynamic algorithm).

For an LLC capacity of c blocks and target expected LLC occupancy of fc blocks with $f \in (0, 1]$, the Occupancy Set must have at least fc blocks. Proposition#1 stated below plays a pivotal role in the design of the CRFill step by relating the number of LLC misses to the expected occupancy.[2]

Proposition#1: A fully associative cache of capacity c blocks exercises random replacement with an eviction probability of $\frac{1}{c}$ for any block. *Any* access sequence having n cache misses achieves an expected occupancy of $c - c\left(1 - \frac{1}{c}\right)^n$ blocks with variance $c\left(1 - \frac{1}{c}\right)^n + c(c-1)(1 - \frac{2}{c})^n - c^2\left(1 - \frac{1}{c}\right)^{2n}$.

Since the expected occupancy depends only on the number of misses, the minimum-length Occupancy Sequence should have only misses and no hits. Using Proposition#1 we find that the required number of misses (n) to achieve an expected occupancy of fc is bigger than fc as shown in Eq.(1). This also follows intuitively, as some of the misses can evict Occupancy Set blocks.

$$n = \frac{\log(1-f)}{\log(1-\frac{1}{c})} = \frac{-f - \frac{f^2}{2} - \frac{f^3}{3} - \cdots}{-\frac{1}{c} - \frac{1}{2c^2} - \frac{1}{3c^3} - \cdots} = \frac{fc(1 + \frac{f}{2} + \frac{f^2}{3} + \cdots)}{1 + \frac{1}{2c} + \frac{1}{3c^2} + \cdots} > fc \text{ for all } f > \frac{1}{c}. \tag{1}$$

Since one iteration of accesses through the minimum-sized Occupancy Set having fc blocks can induce only fc LLC misses, one iteration is not sufficient. If M elements of this Occupancy Set are present in the LLC after one iteration, accessing the Occupancy Set again induces at least $(fc - M)$ additional LLC misses, thus increasing the LLC occupancy. So, the receiver can approach the target occupancy by iterating over a minimum-sized Occupancy Set, but the Occupancy Sequence length may not be optimal.

Alternately, we could choose an Occupancy Set of size $n = \log(1-f)/\log(1-\frac{1}{c})$ and access the LLC using the Occupancy Set once, resulting in the number of misses required to achieve an expected occupancy of fc according to Proposition#1. While this Occupancy Sequence is of optimal length, the Occupancy Set size n is greater than the optimal (fc). Figure 3 shows the results obtained from simulation of this strategy on a 2 MB LLC ($c = 32768$). The Occupancy Set size is varied as an integral multiple (x) of c. The figure shows the percent LLC occupied averaged over a number of experiments for each x where each experiment accesses the entire Occupancy Set once.

From Figure 3, we observe that an Occupancy Set of size 1× achieves an expected occupancy of only 63%. This is what the receiver of SPP [28] achieves when communication begins. Achieving 99% LLC occupancy requires an Occupancy Set of size 5× (4.6× by Proposition#1). The average occupancy is also close to the minimum and maximum occupancy (the standard deviation is at most a few tens of blocks for a 2 MB LLC by Proposition#1). Importantly, in Figure 3, we see that the CRProbe step is more expensive than the CRFill step (PC much larger than FC) due to the timed accesses in the CRProbe step needed to measure occupancy. Therefore, it is much more important to optimize the Occupancy Set size than the Occupancy Sequence length. Hence, in the following, we formalize a strategy of iterating over small Occupancy Sets to attain the target occupancy.

Let $T_{Fill}(x)$ be the CRFill step's latency for an Occupancy Sequence of length x and $T_{Probe}(x)$ be the CRProbe step's latency for an Occupancy Set of size x. If an Occupancy Set OS is accessed R times, we need to search for $(|OS|, R)$ that minimizes $T_{Fill}(|OS|{\times}R) + T_{Probe}(|OS|)$ and achieves the target occupancy. The search space is bounded because $fc \leq |OS| \leq \log(1-f)/\log(1-\frac{1}{c})$. The search for the optimal $(|OS|, R)$ proceeds by increasing $|OS|$ in steps of Δ (a precision parameter) and finding the minimum R that achieves the target occupancy for each $|OS|$. This optimal solution leads to a *static cache region identification procedure* for which $|OS|_{opt}$ is fixed for all R_{opt} iterations.

[2] Proofs of the propositions are omitted for brevity. Interested readers can access them at https://www.cse.iitk.ac.in/users/mainakc/pub/leakyrand-proofs.pdf.

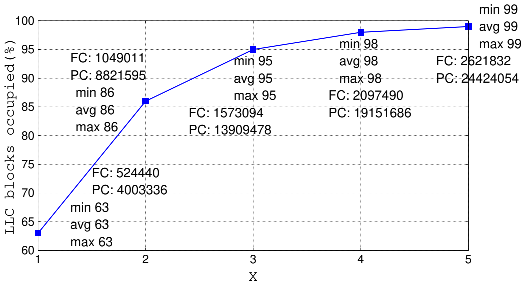


Fig. 3. Percent blocks occupied in a 2 MB LLC after an Occupancy Set having 32768x block addresses is accessed once. FC: CRFill cycles, PC: CRProbe cycles.

A *dynamic cache region identification procedure* that optionally grows $|OS|$ over iterations can achieve the target occupancy faster. Thus, a general formulation of the cache region identification problem is to find the optimal sequence of pairs $\{(N_1, R_1), (N_2, R_2), \ldots, (N_m, R_m)\}$ to achieve the target occupancy. The i^{th} Occupancy Set S_i of size N_i is accessed R_i times in the Occupancy Sequence, which is of length $\sum_i N_i R_i$. To prune the search space of such sequences, we assume that $S_i \subset S_{i+1}$ and use the same Δ as the static procedure to grow the Occupancy Set sizes i.e., $N_{i+1} - N_i$ is a multiple of Δ. The CRProbe step is executed using S_m, the largest Occupancy Set used by CRFill. The search space explored for finding the optimal sequence can be viewed as a recursion tree with each vertex v representing a sequence $\{(N_1, R_1), \ldots, (N_i, R_i)\}$. The children of this sequence are $\{(N_1, R_1), \ldots, (N_i, R_i + 1)\}$ and all sequences of the form $\{(N_1, R_1), \ldots, (N_i, R_i), (N_j, 1)\}$ for all $N_j > N_i$. The root vertex is $\{(N_1, 1)\}$. The sequence at each vertex v is evaluated for the estimated total time T_v and the achieved LLC occupancy Occ_v. The children of the vertex v are explored only if $Occ_v < fc$ and $T_v < T^{opt}_{static}$ where T^{opt}_{static} is the total time of the best static procedure. Also, a child of the form $\{(N_1, R_1), \ldots, (N_i, R_i), (N_j, 1)\}$ is explored only if it satisfies $T_{Probe}(N_j) + T_v < T^{opt}_{static}$.

Before starting the recursive exploration of the search tree, the optimal static cache region identification procedure is synthesized and, during this synthesis process, a record $\mathcal{R}$ is built containing the attained LLC occupancy, the CRFill time, and the CRProbe time for each explored pair (N, R). The search tree exploration uses the same Occupancy Set sizes N as the static synthesis procedure. Hence, record $\mathcal{R}$ can be employed to carry out linear interpolation for efficient estimation of T_v and Occ_v. Let Occ_X be the LLC occupancy attained by sequence $X = \{(N_1, R_1), \ldots, (N_i, R_i)\}$. To estimate Occ_Y for its child $Y = \{(N_1, R_1), \ldots, (N_i, R_i), (N_j, 1)\}$, the record $\mathcal{R}$ is consulted to find the minimum number of iterations k needed by an Occupancy Set of size N_j to attain LLC occupancy $O_{j,k} \geq Occ_X$. The fraction w of the Occupancy Set size N_j needed to attain the occupancy $O_{j,k}$ starting from Occ_X can be computed by linear interpolation. Next, starting from $O_{j,k}$, the occupancy attained using the remaining fraction of the Occupancy Set is estimated as Occ_Y by one more linear interpolation. Similarly, if the CRFill step's time for sequence X is TF_X, the total time for sequence Y is estimated by linear interpolation. This calculation is shown in Eq.(2). The values of $O_{j,k-1}$, $O_{j,k+1}$, T_{Fill}, and T_{Probe} are obtained from Record $\mathcal{R}$. Algorithm 2 summarizes the steps.

$$\begin{aligned} w &= \frac{O_{j,k} - Occ_X}{O_{j,k} - O_{j,k-1}};\ Occ_Y = O_{j,k} + (1 - w)(O_{j,k+1} - O_{j,k}); \\ \text{Total time of Y} &= TF_X + w(T_{Fill}(N_j \times k) - T_{Fill}(N_j \times (k-1))) \\ &\quad + (1 - w)(T_{Fill}(N_j \times (k+1)) - T_{Fill}(N_j \times k)) + T_{Probe}(N_j). \end{aligned} \tag{2}$$

3.3.2 CRProbe Step. By counting hits to the Occupancy Set elements, the CRProbe step computes the LLC occupancy achieved by CRFill. The CRProbe algorithm (Algorithm 3) always maintains an

Algorithm 2: Synthesis of dynamic cache region identification procedure

```
Input:
$CacheSz$: LLC capacity in blocks, $BlockSz$: LLC block size
$TargetOcc$: Target LLC occupancy in blocks
$\Delta$: Increment step in Occupancy Set size (e.g., 10% of $CacheSz$)
Output:
$SolutionList$: All sequences $S$ that perform better than the best static procedure

Record $\mathcal{R}$
Function SynthesizeDynamicProcedure($CacheSz, BlockSz, TargetOcc, \Delta$):
    $SynthesizeStaticProcWithRecord(CacheSz, BlockSz, TargetOcc, \Delta, \mathcal{R})$
    For each enumerated sequence $\mathcal{S}$, estimate the total time $T_{\mathcal{S}}$ and achieved LLC occupancy $O_{\mathcal{S}}$ using $\mathcal{R}$ and linear interpolation
    if $O_{\mathcal{S}} \geq TargetOcc$ AND $T_{\mathcal{S}} <$ the best static solution time then
        $SolutionList \leftarrow SolutionList \cup \mathcal{S}$
    end
    else if $T_{\mathcal{S}} <$ the best static solution time then
        Recursively enumerate and evaluate children sequences of $\mathcal{S}$
    end
    return $SolutionList$
End Function
```

Algorithm 3: Algorithm for CRProbe step

```
Input: $OccupancySet$: Collection of LLC block addresses
Output: $occ$: Achieved LLC occupancy

Int $size$ = len($OccupancySet$), $fetched = 0$, $occ = 0$, $i = size - 1$
goto Occupy // Fetches the code block at label Occupy
NOPs till the end of this code block

ProbeAndFlush: // New code block begins here
while $i >= 0$ do
    $t1$ = rdtsc // Read processor's timestamp counter
    load($OccupancySet[i]$)
    $delay$ = rdtsc - $t1$
    clflush($OccupancySet[i]$)
    if $delay < hit\_miss\_threshold$ OR $i == size - 1$ then
        $occ$++
    end
    $i = i - 1$
end
NOPs till the end of this code block

Occupy: // New code block begins here
if $fetched == 0$ then
    $fetched = 1$
    goto ProbeAndFlush
end
NOPs // To reduce chance of LLC fill on wrong path
i=0
while $i < occ - senderFootprint - receiverFootprint$ do
    load($OccupancySet[i]$)
    $i$++
end
```

invalid LLC way so that a miss to an Occupancy Set block does not evict another Occupancy Set block. To minimize the probability of an LLC miss in the first iteration when no invalid LLC way is available, the Occupancy Set is accessed in the reverse order. Each iteration of the `ProbeAndFlush` `while` loop loads one Occupancy Set element, flushes it, and increments `occ` if the load is an LLC hit. The variable `occ` counts the number of invalid LLC ways. The `while` loop at the label `Occupy` fills up a subset of the invalid ways by loading the Occupancy Set blocks from the beginning leaving space to accommodate the additional footprints of the sender and receiver during communication.

The `occ` variable needs to be equal to the number of invalid LLC ways till the second `while` loop starts. However, this may not hold due to accesses to non-Occupancy Set blocks: (*a*) a non-Occupancy Set block fills up an invalid LLC way, (*b*) a hit to `OccupancySet[i]` at line number 8 is inferred as a miss because a non-Occupancy Set block replaces the code block for those instructions, (*c*) the block containing `OccupancySet[i]` gets replaced by a non-Occupancy Set block after experiencing a hit at line number 8 but before being flushed at line number 10 so that no invalid LLC way is created. The number of non-Occupancy Set blocks is reduced by register-allocating all temporaries and accommodating the two `while` loops in two code blocks. The second loop's code block is also fetched into the LLC by jumping to `Occupy` before starting the first loop. Nonetheless, the following proposition establishes that `occ` is accurate with high probability.

Proposition#2: Let the LLC capacity be c blocks and the Occupancy Set size be $|OS|$. If k is the number of blocks other than the Occupancy Set blocks accessed in the `while` loop starting at the label `ProbeAndFlush`, the probability that the value of the variable `occ` is equal to the number of invalid LLC ways created during this loop is at least $\left(1-\frac{k}{c}\right)\left(1-\frac{5k}{c}\right)\left(1-\frac{k}{c-k}\left(1-\left(\frac{k}{c}\right)^{|OS|-2}\right)\left(\frac{3k}{c}+2\right)\right)$.

For a 2 MB LLC ($c = 32768$) and $|OS| = 1.2c$ (as per our best CRFill algorithm to attain 99% LLC occupancy), this probability is one when $k = 0$ and 0.99 even with $k = 40$. Hence, with probability nearly one, the receiver knows the blocks of the Occupancy Set residing in the LLC at the end of the Probe step. These blocks will be referred to as the *Occupancy Blocks*.

The second `while` loop of the CRProbe algorithm requires the LLC footprints (excluding the Occupancy Blocks) of the receiver and sender. To make the LLC footprint of the receiver and sender binaries constant, we use the `-no-pie` flag of gcc to disable PIE. Using the `-entry` flag of gcc, we override the default entry point `_start` of the receiver and sender executables by a function, which sets the stack pointer to the start of a global array of fixed size and then jumps to the `main` function. Additionally, we use the gcc flag `-fno-stack-protector` and avoid the use of the heap segment.

3.4 Communication Protocol

Before the actual communication begins, the receiver and the sender processes access their code and data in a trial bit communication to fill up the invalid LLC ways left vacant by the CRProbe algorithm. In the following, we discuss our Evict+Flush+Probe protocol for covert communication (Section 3.4.1), bandwidth optimization strategies (Section 3.4.2), types of errors (Section 3.4.3), error syndrome detection and correction (Section 3.4.4), and noise handling (Section 3.4.5).

3.4.1 Evict+Flush+Probe Protocol. For communicating a 1 bit, the sender induces the eviction of at least one Occupancy Block of the receiver by accessing a set S of cache blocks in the sender's address space. We will refer to S as the *Disturbance Set*. Thanks to the large LLC occupancy achieved by the receiver, now it is possible to make $|S|$ small. However, the receiver's LLC occupancy may get reduced if a missing Occupancy Block replaces another Occupancy Block in the Probe step. To *guarantee* that the receiver maintains its LLC occupancy, we observe that it is enough for the sender to access the Disturbance Set blocks and then flush them (using e.g., the `clflush` instruction). The receiver can use these invalid LLC ways created by the sender to fill the missing Occupancy Blocks while probing. The existing covert channels use the flush operation to induce a miss to a shared block [6, 10, 34, 35] or to enable reuse of a partially congruent eviction set [27]. We use the flush operation for a very different purpose. Our Evict+Flush+Probe protocol is summarized below.

Sender's step (Evict+Flush): To send a bit 1, access the Disturbance Set blocks to induce eviction of some Occupancy Block(s) and flush the Disturbance Set blocks. To send a bit 0, do nothing.

Receiver's step (Probe): Probe all Occupancy Blocks and measure the probe latency for each Occupancy Block. If at least one miss is detected based on the measured latency, infer the communicated bit as 1; otherwise the communicated bit is inferred as 0.

Synchronizing the sender and the receiver: The sender's and receiver's steps must strictly alternate without any overlap, as is usual in this kind of channels. This synchronization can be achieved by empirically measuring the worst-case latency of the sender's (receiver's) step and making the receiver (sender) wait for that duration. Given that the receiver's step takes a significant amount of time, it may experience some variation in latency. If the receiver overshoots the predetermined worst-case time, its step is terminated, and the received bit is inferred using the probes done so far.

This synchronization protocol may lead to three different error scenarios as discussed in the following. (*i*) If the receiver overshoots its time budget and is forced to terminate the Probe step prematurely when receiving a 1 bit, the bit may get received as a 0 bit. This error arises if all the Occupancy Blocks evicted by the sender belong to the population of the Occupancy Blocks that the receiver could not probe. (*ii*) Let us consider the scenario in which the receiver overshoots its time budget and is forced to terminate the Probe step prematurely when receiving a 1 bit. If the unprobed population of the Occupancy Blocks has at least one block B missing from the LLC and the next communicated bit is a 0 bit, the next bit will be received as a 1 bit because the receiver will suffer a miss to B in its Probe step when receiving the next bit. This error can be avoided if the receiver remembers which segment of the Occupancy Blocks remains unprobed and does not take into account the LLC misses arising from the access to the unprobed Occupancy Blocks when inferring the next bit. However, this mitigation technique gives rise to the third error scenario. (*iii*) Let us consider the scenario in which the receiver's Probe step is terminated prematurely while receiving a bit and the next communicated bit is a 1 bit. While communicating the next bit, let us suppose that the sender's accesses evict Occupancy Blocks only from the unprobed population. If the receiver ignores all LLC misses arising from the accesses to the previously unprobed population of Occupancy Blocks, the next bit will be erroneously inferred as a 0 bit.

Thanks to the small number of LLC misses experienced by the receiver (due to small Disturbance Sets in LeakyRand), the observed variation in the latency of the receiver's step is small. As a result, the probability that the receiver overshoots the predetermined worst-case interval is very low, and even in those cases, only a small fraction of the Occupancy Blocks remains unprobed leading to a small probability that the sender's accesses evict Occupancy Blocks from the unprobed population only. Thus, the probability of error scenarios (*i*) and (*iii*) is small. Hence, after the receiver overshoots its time budget, it can ignore the LLC misses arising from accesses to the unprobed population of the Occupancy Blocks while receiving the next bit. We further study the implications of this synchronization protocol on the bit error rate in Section 5.

3.4.2 *Bandwidth Optimization.*

The receiver's step is slow due to the timed access to each Occupancy Block. One iteration of the receiver's probe loop has the structure shown in Algorithm 4. `Threshold` is a pre-determined constant decided based on the maximum LLC hit latency ($MaxHL$) and minimum LLC miss latency ($MinML$) observed empirically in a large number of training samples (see Section 5). In general, `Threshold` can be set to $MaxHL + \tau$ for a suitably chosen τ that

Algorithm 4: Receiver's probe loop iteration

```
Int $t1, t2, i$, Threshold;
fence; $t1$ = rdtsc; fence;
load($OccupancyBlock[i]$);
fence; $t2$ = rdtsc; fence;
$i = i + 1$;
if $t2 - t1 >$ Threshold then
    infer miss;
end
else
    infer hit;
end
```

minimizes the number of LLC hit/miss inference errors in the training samples. In the commonly observed case of $MaxHL < MinML$, the value of τ can be set to $(MinML - MaxHL)/2$.

To reduce the overhead of latency measurement and introduce memory-level parallelism, we club the measurement of a group of loads. We refer to the number of loads in a group as the unrolling factor. Since the number of Occupancy Blocks N_{OB} may not be divisible by the unrolling factor F, the receiver does not probe the last (N_{OB} mod F) Occupancy Blocks sacrificing LLC occupancy by at most $F - 1$ blocks. One iteration of the probe loop for $F = 2$ is shown in Algorithm 5.

Algorithm 5: Receiver's probe loop iteration with unrolling factor $F = 2$

```
Int $t1, t2, i$, Threshold;
fence; $t1$ = rdtsc; fence;
load($OccupancyBlock[i]$);
load($OccupancyBlock[i + 1]$);
fence; $t2$ = rdtsc; fence;
$i = i + 2$;
if $t2 - t1 >$ Threshold then
    infer miss;
end
else
    infer hit;
end
```

For an unrolling factor F, let $MaxHL_F$ denote the maximum observed latency of a group of F LLC hits and $MinML_F$ denote the minimum observed latency of a group of F LLC accesses with at least one miss. We find τ_F such that setting Threshold to $MaxHL_F + \tau_F$ minimizes the number of inference errors in the training samples. Due to the accumulated variation in the hit/miss latency of a group of F LLC accesses, $(MinML_F - MaxHL_F)$ decreases with increasing F and eventually becomes negative. Therefore, beyond a certain F, the number of inference errors may increase. However, for a given F, increasing the Disturbance Set size may lead to at least one group of F loads with more than one LLC miss, thus improving the inference margin while sacrificing bandwidth.

3.4.3 Communication Errors. A communication error is said to occur when the sender's step has the effect of sending the inversion of the actual bit. These communication errors arise due to unexpected LLC eviction events. To understand these errors, we categorize the LLC blocks into four groups. Group I consists of the Occupancy Blocks of the receiver. Group II consists of all code and data blocks except the Occupancy Blocks used by the receiver's and sender's steps in the communication of every bit. Group III consists of the blocks that are used by the receiver's and sender's steps only in certain scenarios which may not arise in the communication of every bit. Group IV consists of all the remaining LLC blocks. These blocks are not used during communication.

The sender's Disturbance Set must evict at least one Group I block from the LLC. A 1 bit is received as a 0 bit if all blocks of the Disturbance Set evict only Group II, Group III, and Group IV blocks. Let us suppose that the sender's accesses to the Disturbance Set evict a Group II block and $|DS|-1$ Group I blocks from the LLC where $|DS|$ is the Disturbance Set size. Since the sender flushes all the Disturbance Set blocks from the LLC, the LLC ways from which the Group II block and the Group I blocks are evicted become invalid after the flush operations. During the receiver's Probe step, $|DS|-1$ invalid LLC ways get filled with the evicted Occupancy Blocks. Since the Group II blocks are accessed whenever the receiver and sender execute their communication steps, the remaining invalid LLC way also gets filled with the evicted Group II block. Thus, the invalid LLC ways created due to sender's eviction of Group I and Group II blocks get filled automatically without much delay. On the other hand, if the sender evicts a Group III or Group IV block, the invalid LLC way thus created may not get filled immediately in the absence of third party noise because Group III and Group IV blocks are not accessed in the communication of every bit. If the number of such invalid LLC ways keeps increasing and becomes equal to the Disturbance Set size, the sender will not be able to evict any Group I block because all the Disturbance Set blocks will get filled into the invalid LLC ways. From then on, all 1 bits will be erroneously communicated as 0 bits. Thus, accumulation of invalid LLC ways increases the likelihood of 1s getting communicated as 0s.

3.4.4 Error Handling. Inference errors can be avoided through appropriate setting of hit/miss inference threshold and Disturbance Set size for a given unrolling factor F (see Section 3.4.2). Here we focus on handling communication errors only. Two possible reasons for such errors that flip a 1 bit to a 0 bit are (a) accumulation of invalid LLC ways due to eviction of idle Group III and Group IV blocks, and (b) inability of the sender's Disturbance Set in evicting a Group I block. Handling the first type of error requires detecting the relevant error syndrome, i.e. the existence of invalid LLC ways and then correcting the LLC state. The probability of the second type of error can be reduced by appropriately sizing the Disturbance Set. We discuss both in the following.
Error Syndrome Detection and Correction. The error syndrome detection algorithm for a 1 bit to 0 bit error is invoked periodically to avoid accumulation of too many invalid LLC ways. Every k^{th} bit communicated by the sender is a marker bit of value 1. To receive the marker bit, the receiver probes the Occupancy Blocks. Additionally, it counts the number of iterations of the unrolled probe loop that detect LLC misses. If this number is M, the number of invalid LLC ways is at most $(|DS|-M)$ where $|DS|$ is the Disturbance Set size. The receiver uses an empirically chosen small unrolling factor in the probe loop of the error syndrome detection algorithm so that at most one LLC miss can occur in one iteration of the unrolled loop with high probability. If $M < |DS|$, the error syndrome is detected and the receiver invokes the correction algorithm.

The correction algorithm runs at most $(|DS|-M)$ iterations. In each iteration, the receiver fills one invalid LLC way by accessing a new block B from a set of blocks referred to as the *Correction Set*. Next, the receiver flushes B from the LLC recreating the invalid LLC way. Then the receiver probes the Occupancy Blocks. If a miss is detected while probing, the receiver concludes that the access to B must have evicted an Occupancy Block and therefore, there is no more invalid LLC way. Hence, the correction algorithm is terminated. On the other hand, if no miss is detected while probing the Occupancy Blocks, the receiver accesses B again to fill the invalid LLC way created by flushing B and moves on to the next iteration if $(|DS|-M)$ iterations are not yet completed.

The error correction mechanism introduces the possibility of a new error where a 0 bit gets communicated as a 1 bit. To see this, let us suppose that while communicating a 1 bit, the sender replaces a Group III block B creating an invalid LLC way. During the next error correction step, all the invalid LLC ways are filled up. After that the replaced Group III block B is accessed and that replaces an Occupancy Block B'. Next, the sender communicates a 0 bit and the receiver observes a miss to B' while probing the Occupancy Blocks, thus wrongly receiving a 1 bit. In the absence of error correction, when the Group III block B is reaccessed, it would have occupied the invalid LLC way without disturbing the Occupancy Blocks. Fortunately, this error can be avoided by forcing the reaccess of B to occur before the error correction step. Hence, the sender touches all its blocks after sending the marker and the receiver touches all its blocks before error correction. Thus, the error correction iterations fill up the invalid LLC ways created due to eviction of Group IV blocks only.

A 0 bit can still get received as a 1 bit if the number of error correction iterations is over-estimated leading to eviction of Group II or Group III blocks by Correction Set blocks. This recreates the scenario discussed above leading to eviction of some Occupancy Blocks when the evicted Group II/III blocks are reaccessed. An over-estimation in the number of error correction iterations happens if an invalid LLC way that existed when transmitting the marker bit gets filled up when the sender touches its blocks after sending the marker. To detect the 0 bit to 1 bit error syndrome, the sender sends a marker bit of value 0 after 1 bit to 0 bit error correction is completed. If this 0 bit is received as a 1 bit, the error syndrome is detected. The error correction algorithm proceeds iteratively. In the i^{th} iteration, the receiver creates an invalid LLC way by flushing the i^{th} last accessed Correction Set block and probes the Occupancy Blocks to detect any LLC miss. In each iteration, the invalid LLC way can accommodate one evicted Occupancy Block. In the worst-case, $|DS|$ correction iterations are needed to fill all evicted Occupancy Blocks. In summary, each stretch of $(k + 1)$ bits contains $(k - 1)$ data bits and two marker bits where k is defined as the error correction interval. The Occupancy Blocks are probed at most $2(|DS|+1)$ times in each invocation of error detection and correction.

Disturbance Set Size. The sender's accesses to the Disturbance Set should evict an Occupancy Block with high probability when sending a 1 bit. The following proposition shows that even a small Disturbance Set has a low expected bit error rate (BER) for a large Occupancy Block population.

Proposition#3. Consider communicating n bits with each bit equally likely to be 0 or 1. If the Occupancy Blocks fill up a fraction f of the LLC, the expected 1 bit to 0 bit error rates for Disturbance Set sizes one and two are respectively $\frac{1}{2} - \frac{f}{n(1-f)}\left[1 - \left(\frac{1+f}{2}\right)^n\right]$ and $\frac{1}{2} - \frac{f(f+2)}{n(1-f^2)} + \frac{2f}{n(1-f)}\left(\frac{1+f}{2}\right)^n - \frac{f^2}{n(1-f^2)}\left(\frac{1+f^2}{2}\right)^n$.

With an error correction interval of k, we can use this proposition to compute BER by setting $n = k - 1$ as the number of data bits between two error correction steps. For $f = 0.99$ and $k = 32$, the expected BERs for $|DS|= 1$ and 2 are respectively 0.0404 and 0.0043 irrespective of the total number of communicated bits. A larger $|DS|$ would be needed to achieve a BER lower than this.

Analytical derivation of the relationship between a general $|DS|$ and the expected BER is tedious. We employ a fast bit error simulation algorithm (Algorithm 6) to estimate the BER. We use this algorithm to decide $|DS|$ for a target BER. This $|DS|$ is then used in a more detailed processor simulation setup. In each trial, Algorithm 6 simulates the sender's step by generating random data bits (line 5) and modeling random LLC evictions to communicate the bits (line 9). The Occupancy Blocks are assumed to be the first `OBCount` LLC blocks. When the number of invalid LLC ways (`invalWays`), created outside the Occupancy Blocks due to sender's evict and flush operations (lines 10-12), becomes equal to the Disturbance Set size (`DSSize`), bit errors occur (lines 14-16).

Algorithm 6: Algorithm to estimate bit error rate

```
Input:
  CacheSize: Number of LLC blocks, OBCount: Number of Occupancy Blocks
  DSSize: Number of Disturbance Set blocks, NumTrials: Number of experiments
  BitStringLen: Number of data bits to communicate
Output: BER: Estimated bit error rate
1  Int $BER = 0$
2  for $k = 0; k < NumTrials; k{+}{+}$ do
3      Int $invalWays = 0$
4      for $i = 0; i < BitStringLen; i{+}{+}$ do
5          Int $bit$ = random() % 2
6          if $bit$ then
7              Int $numEvictions = DSSize - invalWays$
8              for $j = 0; j < numEvictions; j{+}{+}$ do
9                  Int $evictId$ = random() % $CacheSize$
10                 if $evictId >= OBCount$ then
11                     $invalWays{+}{+}$
12                 end
13             end
14             if $invalWays == DSSize$ then
15                 $BER{+}{+}$
16             end
17         end
18     end
19 end
20 $BER = BER/(BitStringLen \times NumTrials)$
```

3.4.5 Handling Noise. The noise process has the effect of accessing the LLC with third party blocks that do not belong to the receiver or sender. LeakyRand, by design, has in-built noise resilience.

The size of the sender's Disturbance Set can be adjusted to absorb the noise blocks in the invalid ways created by the sender in the LLC region that is unoccupied by the receiver.

3.5 Summary of LeakyRand

The LeakyRand covert channel setup involves two high-level steps—cache region identification and covert communication. In the cache region identification step, the receiver quickly fills up a large fraction of the LLC. A novel Evict+Flush+Probe protocol is used to carry out high-bandwidth covert communication while helping the receiver maintain its occupancy. Efficient error syndrome detection and correction algorithms help increase channel fidelity.

4 Discussion

In this section, we discuss the extended scope and applications of LeakyRand, the designs of LeakyRand without any reliance on the cache line flush instructions as well as with a simpler implementation of the random replacement policy, and directions for mitigating LeakyRand.

Scope and Applications of LeakyRand. The scope of LeakyRand extends beyond fully associative caches with random replacement policy. In the following, we argue that the working principles of LeakyRand are not tied to the uniform eviction probability distribution of the fully associative cache ways. The static cache region identification procedure does not depend on the replacement policy, but appeals to Proposition#1 only to bound the search space. However, Proposition#1 can be restated in terms of a general probability distribution in which the cache way i has an eviction probability p_i with $\sum_{i=0}^{c-1} p_i = 1$ for a cache of capacity c blocks. Specifically, the expected cache occupancy due to n misses takes the form $c - \sum_{i=0}^{c-1}(1 - p_i)^n$. Given a target expectation, a new bound for the search space can be computed numerically. The dynamic cache region identification procedure traverses a search tree and employs linear interpolation on the data collected by the static procedure to estimate the achieved occupancy and the latency of an Occupancy Sequence. The linear interpolation step relies on the fact that a bigger Occupancy Set or Sequence occupies a proportionately bigger cache region and imposes a proportionately bigger latency. This principle continues to hold as long as the cache replacement policy honors the access ages of the blocks and does not have an unduly large eviction bias for any particular group of ways. This is true for all age-based and pseudo-random cache replacement policies. The CRProbe step relies on Proposition#2 for a correct outcome with high probability. The probability bound stated in Proposition#2 improves further for age-based replacement policies, since these policies reduce the chance of the recently accessed Occupancy Blocks getting replaced due to accesses to the other blocks.

The communication protocol relies on the sender's ability to evict at least one Occupancy Block of the receiver when communicating a 1 bit. For an age-based replacement policy, the Occupancy Blocks can be placed on the higher eviction priority side of the eviction stack if the replacement policy is known; otherwise, the Disturbance Set size can be adjusted to affect eviction of Occupancy Blocks by the sender. Error syndrome detection and correction protocols do not rely on any specific property of the cache replacement policy. However, the average BER is a function of the cache replacement policy as is evident from Proposition#3 and Algorithm 6. In summary, LeakyRand can be adopted to work in the presence of any age-based or pseudo-random replacement policy. Futher, since a set of a set-associative cache can be viewed as a standalone fully associative cache, LeakyRand can be used to construct a covert channel on a set of a shared set-associative LLC. We evaluate such adoptions of LeakyRand to real-world settings and different replacement policies in Section 5. Additionally, since the *CRFill* and *CRProbe* steps help an attacker occupy most of the LLC with a known set of blocks, even small footprints of victim workloads can be identified accurately enabling high-precision fine-grain fingerprinting attacks in a fully associative LLC. In contrast, SPP would be able to identify only large footprints. We discuss the application of LeakyRand to fingerprinting attacks in Section 5.

LeakyRand without Cache Line Flush Instructions. In a multi-socket processor, a block residing in one socket's cache hierarchy can be invalidated with the help of a coherent store to the block executed on another socket, thereby mimicing the effect of a cache line flush instruction. The cache line flush operations of the sender in LeakyRand's communication rounds can be implemented by making the sender process dual-threaded and scheduling the two threads on two different sockets using the POSIX API `pthread_setaffinity_np`. Let us suppose that the receiver process and the main thread of the sender process are scheduled on two cores of socket S1. A helper thread of the sender process that assists the main thread in invalidating the Disturbance Set blocks from socket S1 is scheduled on a core in socket S2. The skeleton procedure to invalidate the Disturbance Set blocks $B_1, B_2, \ldots, B_n$ resident in the cache hierarchy of socket S1 is shown in Algorithms 7 and 8. The variables `flag1` and `flag2`, shared among the threads of the sender process and both initialized to zero, are used to synchronize the two sender threads. We assume total store order memory consistency model for correct execution of the code. Since the receiver is, by far, the biggest bottleneck in the performance of LeakyRand, we do not expect to see a large performance loss if the cache line flushes in the sender are replaced by cross-socket coherent invalidations. We further evaluate this "flushless" design of LeakyRand using the Evict+Invalidate+Probe communication protocol in Section 5.

Algorithm 7: Main sender thread (scheduled on socket S1)

```
Access $B_1, B_2, \ldots, B_n$; // Sender's Evict step
flag1 = 1;
while (flag2 == 0);
flag2 = 0; // Reinitialize for next bit
```

Algorithm 8: Helper sender thread (scheduled on socket S2)

```
while (flag1 == 0);
flag1 = 0; // Reinitialize for next bit
Write to $B_1, B_2, \ldots, B_n$; // Invalidate blocks in S1
flag2 = 1;
```

LeakyRand without Fill Preference for Invalid Ways. Random replacement has a sub-optimal, yet simpler, implementation which does not fill invalid ways first. In this case, LeakyRand does not use any cache line flush operation since invalid ways are not treated specially. The receiver uses a simpler (but less efficient) CRFill step to fill a large fraction of the LLC by iterating over an array A of size equal to the LLC capacity. The CRProbe step probes all the blocks of the array A and records the total latency. The sender's communication step only accesses a Disturbance Set to evict some of the receiver's blocks. Thanks to the large LLC occupancy of the receiver, the Disturbance Set is much smaller than what SPP needs for high fidelity. To infer a received bit, the receiver probes the array A, records the total latency as L_{curr}, compares it with the probe latency L_{prev} observed while receiving the previous bit (or during CRProbe). If $L_{curr} - L_{prev} > \tau$, the received bit is inferred as 1 and 0 otherwise. The inference threshold τ is determined using a set of training runs (see Section 5).

To further improve fidelity, the receiver employs an n-bit history of received bits to divide the inference space of the next bit into 2^n subspaces and uses a different inference threshold τ_h for each history subspace h. Let us refer to $L_{curr} - L_{prev}$ after a history h as δ_{h0} and δ_{h1} when the current bit is 0 and 1 respectively. If the distributions of δ_{h0} and δ_{h1} are non-overlapping in the training set, we fix τ_h as $(\min \delta_{h1} + \max \delta_{h0})/2$. For the history subspaces with overlapping distributions, we replace a subset of the sender's Disturbance Set with a new subset of block addresses so that the distributions have a high probability of becoming non-overlapping. To bootstrap the process, the CRProbe step runs n probes over the entire array A so that the LLC reaches the state with history equal to 0^n and L_{prev} is set to the latency of the last among these n probes. This implementation of the LeakyRand channel is most general as it works with any implementation of the random replacement policy and without the cache line flush instruction. We refer to this channel as *LeakyRandGen*.

Detection and Mitigation. Studies on online detection of cache timing attacks have exploited the cyclic access pattern of the receiver and sender to the blocks within a set of a set-associative LLC [13] or run-time features collected using hardware event counters [19]. The number of active cache blocks involved in the construction of the LeakyRand channel is much larger than a timing channel set up within a set of a set-associative cache. Therefore, the complexity of the detection algorithm is likely to increase. Effective mitigation of LeakyRand would need to isolate the receiver and sender processes in the LLC so that the sender cannot evict receiver's blocks. Partitioning the LLC among security domains can offer such isolation. For a fully associative LLC, this is equivalent to way partitioning. A complete study exploring the accuracy of the existing detectors when used to detect LeakyRand and efficient partitioning of fully associative caches is left to future work.

5 Evaluation

In this section, we discuss our simulation methodology and present a quantitative evaluation of the LeakyRand channel including a comparison with Stochastic Prime+Probe (SPP) [28].

5.1 Evaluation Methodology

5.1.1 Simulation Environment. We use ChampSim, a trace-based simulation infrastructure [9], to evaluate our proposal. ChampSim accepts a trace of instructions of the executable binary and simulates the trace on the processor model. It has detailed cycle-accurate timing models of the out-of-order issue cores, the cache hierarchy, and the main memory. Table 1 shows the details of the simulated processor. We augment ChampSim with `fence` and `clflush` instructions. We use CACTI [1] to determine the cache lookup latency for 7 nm FinFET process. The fully associative inclusive LLC with random replacement policy is modeled following the MIRAGE design, which is provably secure against conflict-based side-channel attacks [25]. The MIRAGE design implements decoupled tag and data stores. The tag store is skewed-associative with two set-associative load-balanced skews. Each set-associative skew has an adequately over-provisioned tag array to guarantee freedom from set-associative conflicts with high probability. Additionally, the address to tag set mapping is randomized using the low-latency PRINCE encryption function. The data store supports fully associative placement of data blocks with indirection pointers maintaining the association between a tag and its data block. Independent of the MIRAGE design, we have also validated our results through simulation of a traditional (impractical) fully associative LLC with lookup latency same as MIRAGE. The instruction traces of the sender and receiver programs are collected using a Pin tool [18]. These traces are replayed on two cores of the simulated processor.

Table 1. Simulated processor configuration

Processor	Dual-core, 4 GHz, out-of-order issue superscalar cores
Microarchitecture of each core pipeline	Fetch, execute, retire width: 4, (6 non-memory + 2 loads + 1 store), 4; Scheduler/Load/Store queue sizes: 100/72/56; ROB size: 256
L1D and L1I caches	32 KB/8-way/LRU/2 cycles (private per core)
L2 cache	256 KB/8-way/LRU/3 cycles (private per core)
Shared LLC [25]	2 MB/FA/Random/15 cycles (11 cycles array lookup + 3 cycles PRINCE cipher [4] + 1 cycle tag→data pointer indirection)
Cache line size	64 bytes for all caches
DRAM	800 MHz, tCAS=tRP=tRCD=11 cycles

To study the impact of operating system noise on LeakyRand, we use the Gem5 simulator [3] in the full-system mode. We boot a simulated quad-core processor with Linux kernel 5.2.3 and use two of the cores to schedule the receiver and the sender. The configuration of the simulated cache hierarchy is as shown in Table 1.

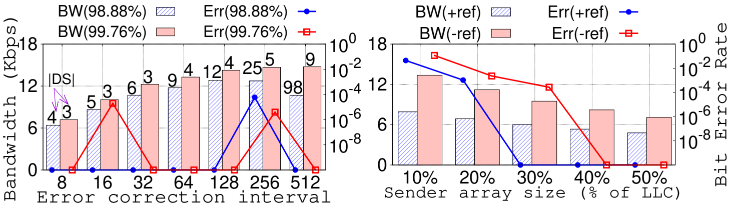
5.1.2 Benchmark Suite. We generate 750 random 512-bit strings that the sender communicates to the receiver. A subset of 250 strings, referred to as the train suite, is used to decide various inference thresholds. We report our results on the remaining 500 strings which constitute the test suite.

5.1.3 Comparison to Related Work. As discussed in Section 2, the covert channel proposals either exploit LLC conflicts or rely on shared memory between the receiver and the sender. Our proposal belongs to the first category. Therefore, to keep the comparison fair, we confine our evaluation to the conflict-based covert channels which do not use shared memory. Furthermore, a vast majority of the proposals are designed for LLCs that exercise age-based replacement policies (e.g., traditional Prime+Probe). These are rendered ineffective in the presence of random replacement. We compare LeakyRand with SPP in the presence of random replacement policy, while an adoption of LeakyRand to the LRU policy is compared with the traditional Prime+Probe covert channel. We also show application of LeakyRand to mounting fine-grain fingerprinting attacks.

5.1.4 SPP Configuration. To improve communication bandwidth, we study SPP without repetition coding in a noiseless setup. In SPP, the receiver uses an array of size equal to the LLC capacity for priming and probing. We study the impact of varying the sender's Disturbance Set size. Differential signaling employed by SPP to reduce inference errors sends a reference 0 bit followed by the actual data bit. If the receiver's probe latencies to receive the reference bit and the data bit are respectively L_0 and L_d, the data bit is inferred as a 1 if $L_d > L_0 + \tau$ and 0 otherwise. We also evaluate SPP without differential signaling, in which the data bit is inferred as a 1 if $L_d > \tau'$ and 0 otherwise. For each Disturbance Set size, we choose τ and τ' that minimize bit errors in the train suite.

5.2 Efficiency and Robustness of LeakyRand

Figure 4 quantifies the average channel bandwidth and bit error rate (BER) of LeakyRand and SPP using the 500 test suite strings. The receiver's communication step in LeakyRand uses an unrolling factor of 16 in its probe loop. The LeakyRand channel is studied for two different Occupancy Block populations (OBPs) namely, 98.88% and 99.76% of LLC capacity (Figure 4a). For each OBP, we evaluate LeakyRand for seven error correction intervals (x-axis of Figure 4a). For each error correction interval (ECI) k, we use the $|DS|$ value (in number of LLC blocks) that minimizes the BER in Algorithm 6 when the number of trials is $\lceil \frac{512}{k-1} \rceil \times 500$ (the total number of data communication stretches in the test suite). These $|DS|$ values are shown on top of the bars in Figures 4a. As expected, $|DS|$ is lower for higher OBP and increases with larger ECI (i.e., less frequent error correction). The SPP channel is evaluated with (+ref) and without (−ref) reference bits for five different Disturbance Set sizes namely, 10%, 20%, 30%, 40%, and 50% of the LLC capacity (x-axis of Figure 4b).



(a) LeakyRand

(b) SPP

Fig. 4. Efficiency and robustness of LeakyRand and SPP.

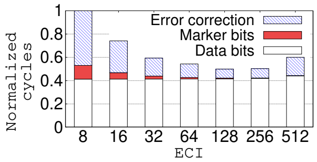


Fig. 5. End-to-end cycles of LeakyRand with OBP=98.88% normalized to ECI=8.

For OBP of 98.88%, LeakyRand channel bandwidth increases with ECI and attains a peak bandwidth of 12.8 Kbps for ECI=128. On the other hand, a peak bandwidth of 14.8 Kbps is attained for the ECI of 512 when OBP is 99.76%. The reason for this trend is that a larger ECI leads to an overall lower error correction overhead. However, the bandwidth decreases for ECIs greater than 128 for OBP of 98.88% because ECI > 128 requires a significantly bigger $|DS|$ to keep BER low. Figure 5 confirms this trend by showing a breakdown of the total cycles expended by LeakyRand.

The SPP channel bandwidth decreases from 7.9 Kbps to 4.6 Kbps with increasing $|DS|$ in the presence of reference bits ("+ref" in Figure 4b). At the best bandwidth point of 7.9 Kbps, the BER is

0.043. For error-free communication, the best bandwidth of SPP is only 6 Kbps attained with a $|DS|$ of 30% LLC capacity. In the absence of reference bits, the best bandwidth of SPP is 13.4 Kbps with a BER of 0.114, while for error-free communication, the best bandwidth is 8.2 Kbps (for $|DS|$= 40%).

LeakyRand, on the other hand, offers error-free communication in all except three cases namely, for ECI=16 when OBP is 99.76% and for ECI=256, experiencing a maximum BER of 0.000058. High OBP and low-overhead error correction are the reasons for this remarkably low BER and high channel bandwidth. We find that when error correction is disabled, LeakyRand is able to offer error-free communication of the test suite strings with $|DS|$ of 100 and 7 blocks respectively for 98.88% and 99.76% OBPs achieving 14.5 Kbps and 15.3 Kbps bandwidth. However, with increasing message length, $|DS|$ will increase to keep BER low and the bandwidth will drop. So, for long messages, error correction should be enabled with the best ECI, which is independent of the message length.

Figure 6 studies the effect of increasing $|DS|$ on the bandwidth and the BER of the LeakyRand channel for fixed ECI of 128 and OBP of 98.88%. The errors observed for $|DS|$< 12 are all 1 bit to 0 bit errors. As $|DS|$ increases, the channel observes exponentially decreasing BERs while offering linearly decreasing bandwidth in the range of 15 Kbps to 12.8 Kbps. Thus, $|DS|$ can be effectively used to operate the LeakyRand channel at high bandwidth with an ultra-low BER. Without error correction (not shown in Figure 6), the BERs for $|DS|$= 1, 3, 6, 9, 12 are 0.34, 0.19, 0.12, 0.082, 0.048 respectively while the bandwidth ranges from 15.6 Kbps to 15.4 Kbps. Thus, our error detection and correction algorithms help achieve remarkably low BER while imposing a small overhead.

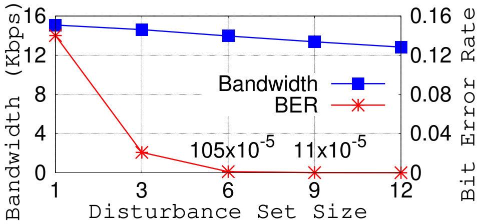


Fig. 6. LeakyRand's sensitivity to Disturbance Set size (ECI=128, OBP=98.88%).

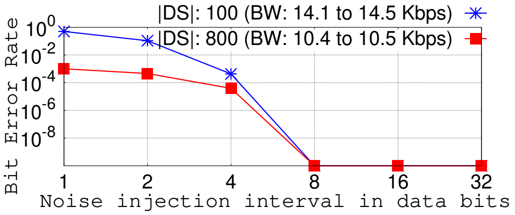


Fig. 7. Impact of noise on LeakyRand's BER with varying $|DS|$ (OBP=98.88%, no error correction).

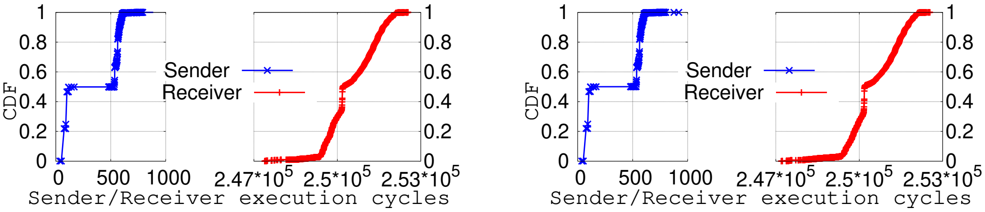
5.3 Inherent Noise Resilience of LeakyRand

To evaluate LeakyRand in a noisy setup, we model a noise process by injecting third party block accesses to the LLC. Figure 7 shows how the BER varies as the noise injection rate is decreased from one noise block access per data bit to one noise block access per 32 data bits. We disable error detection and correction in this study to show how $|DS|$ can be adjusted to mitigate the noise effects. For $|DS|$ values of 100 and 800 (with 98.88% OBP), the observed BER is zero when noise injection rates are 8, 16, 32. Importantly, as $|DS|$ is increased from 100 to 800, the BER drops significantly for high noise injection rates while maintaining a channel bandwidth close to 10.5 Kbps which is still higher than what SPP achieves for the same level of BER in a noiseless setup (Figure 4b).

To study the impact of operating system (OS) noise, we run LeakyRand with $|DS|$ = 1500, 3000, 4500 blocks on the Gem5 simulator in full-system mode. We observe that the corresponding average bit error rates are 0.025, 0.004, and 0.001 respectively. Furthermore, 80% of the strings are communicated without error for $|DS|$ = 4500. Thus, $|DS|$ can be adjusted to mitigate OS noise. Importantly, for $|DS|$= 4500 (13.7% of LLC), SPP has a much higher BER in a noiseless setup (Figure 4b).

5.4 Synchronization Protocol of LeakyRand

In Section 3.4.1, we have discussed how the sender and receiver processes synchronize themselves based on the worst-case execution cycles of the receiver and sender, respectively. Figure 8 shows that the cumulative distributions of the sender's and receiver's execution cycles for sending and receiving a bit in the test suite match closely with those in the train suite. The sender's distribution is bimodal in nature due to very different amounts of work needed to transmit a 0 bit and a 1 bit. Overall, we conclude that the worst-case execution cycles observed in the train suite can be used with an appropriate additional noise margin to synchronize the receiver and the sender during communication of the test suite strings. Since the receiver's execution time varies over a wide range of about 6000 cycles, there is a possibility that the receiver may overshoot its time budget. However, this variation is less than 3% of the receiver's minimum execution cycles observed in the entire test suite, indicating that any overshoot is most likely to be small.



(a) CDF of cycles in train suite

(b) CDF of cycles in test suite

Fig. 8. CDF of the sender's and receiver's execution cycles for sending and receiving a bit.

In Section 3.4.1, we have discussed how we handle scenarios when the receiver overshoots its predetermined time budget. In such cases, the receiver's Probe step is terminated prematurely and the received bit is inferred based on the probes done so far. Also, the receiver ignores the LLC misses arising from the accesses to the unprobed population of the Occupancy Blocks when receiving the next bit. For a direct evaluation of this synchronization protocol, we deliberately terminate the receiver after it has probed X% of the Occupancy Blocks when receiving randomly chosen Y% of the bits. Our evaluation on the entire test suite shows that for $Y \in \{1, 2, 3, 4, 5, 6\}$ and $X \geq 87$, LeakyRand does not experience any additional errors showing the effectiveness of our synchronization protocol. We observe one, five, and one errors respectively for $(X, Y) = (86, 4), (86, 5), (86, 6)$. However, given that the variation in the receiver's execution time is small, it is most unlikely that the unprobed region would be as large as 14% of the Occupancy Blocks.

5.5 Performance of “Flushless” LeakyRand

We study the performance of the “flushless” version of LeakyRand as discussed in Section 4. The pseudocode employing coherent cross-socket invalidations to emulate cache line flush operations was presented in Algorithms 7 and 8. We find that invalidating one cache block in a socket using cross-socket invalidation takes about 1400 ns including the flag-based synchronization overhead in a dual-socket platform with Intel Xeon Gold 6246 processors. When the corresponding 5600-cycle (at 4 GHz) latency to emulate one cache line flush operation is incorporated in our simulation model, we observe that LeakyRand achieves peak bandwidths of 12.6 Kbps and 14.5 Kbps respectively for 98.88% and 99.76% OBPs. The drop in the peak bandwidth due to non-availability of cache line flush instructions is small because the sender is not the bottleneck.

5.6 Performance of LeakyRandGen

As discussed in Section 4, LeakyRandGen is the most general implementation of LeakyRand that works with a simpler implementation of the random replacement policy. LeakyRandGen also does not rely on the cache line flush instructions. We evaluate LeakyRandGen on the test suite using history length of 3 bits and $|DS|= 6250$ (19% of LLC capacity) with a new set of 1000 block addresses replacing a subset of DS when the history is 001, 011, 101, or 111. For the entire test suite, LeakyRandGen does not experience any error while delivering a bandwidth of 11.2 Kbps. In contrast, the best bandwidth achieved by SPP for error-free communication is only 8.2 Kbps (Figure 4b).

5.7 Bandwidth Optimization in LeakyRand

As discussed in Section 3.4.2, LeakyRand's bandwidth can be improved by increasing the unrolling factor. For each unrolling factor, Table 2 shows the maximum latency of an iteration having all LLC hits (*MaxHL*) and the minimum latency of an iteration having at least one LLC miss (*MinML*) observed while communicating the train suite strings with ECI=128 and OBP=98.88%. When *MinML* becomes less than *MaxHL*, the latency distributions of the iterations with only LLC hits and the iterations with at least one LLC miss overlap. We set the LLC hit/miss inference threshold (last column of Table 2) to $(MaxHL + MinML)/2$ for non-overlapping distributions; otherwise we set the threshold to $MaxHL + 1$ so that all 0 bits may be received correctly. During error detection and correction, the probe loop uses unrolling factor of 16 to ensure high reliability.

Table 2. Impact of probe loop's unrolling factor on train suite

Unrolling factor	*MaxHL* (cycles)	*MinML* (cycles)	Overlapped iterations (%)	Threshold (cycles)
16	121	127	0	124
32	133	135	0	134
64	232	162	0.02	233
128	315	216	0.04	316
256	630	366	4.09	631

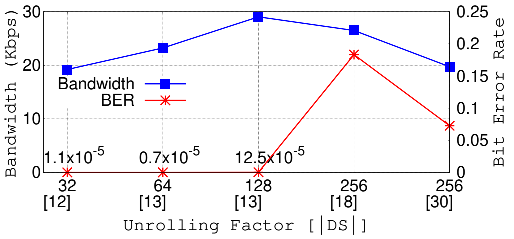


Fig. 9. Impact of probe loop's unrolling factor on test suite

Figure 9 shows the variation in LeakyRand's bandwidth and BER with unrolling factor when communicating the test suite strings for ECI=128 and OBP in the range 98.44% to 98.88% for a 2 MB LLC. The $|DS|$ values are chosen using Algorithm 6 to minimize 1 bit to 0 bit errors and shown below each unrolling factor in the figure. The OBP and $|DS|$ vary slightly with unrolling factor because the number of Occupancy Blocks probed by the receiver must be an integer multiple of the unrolling factor. The channel bandwidth peaks at about 29 Kbps with a BER of about 0.0001 for unrolling factor of 128. For unrolling factor of 256, the bandwidth drops due to an increase in $|DS|$ and BER increases to 0.17 due to overlapping latency distributions; when $|DS|$ is increased to 30 LLC blocks (last x-axis point), the BER drops to 0.07 due to reduced overlap in latency distributions.

5.8 Sensitivity to LLC Capacity

Figure 10 shows how the channel bandwidth of LeakyRand and SPP (with reference bits) varies with LLC size. LeakyRand observes a BER of 10^{-4} or less while delivering a bandwidth that is more than 4× of SPP across the board.

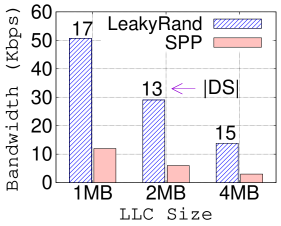


Fig. 10. Sensitivity to LLC size. LeakyRand: ECI=128, OBP=98.88%, unrolling factor=128. SPP: $|DS|$=30% LLC.

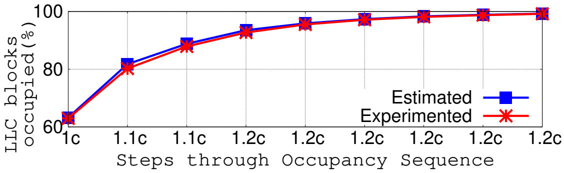


Fig. 11. Estimated and simulated LLC occupancy.

5.9 Efficiency of Cache Region Identification

Table 3 lists ten Occupancy Sequences generated by our dynamic synthesis procedure (Algorithm 2) to occupy 99% of a 2 MB LLC with Δ =10% of LLC block count. Each sequence is represented as

$N_1^{R_1} N_2^{R_2} \dots N_m^{R_m}$ where an Occupancy Set of size N_i is accessed R_i times and N_i is represented as a multiple of the number of LLC blocks c. The second and fourth columns of the table show that the percent deviation in cycles estimated by the algorithm using linear interpolation from the cycles observed in detailed simulation is very small. The best performing dynamic sequence is $(1c)^1(1.1c)^2(1.2c)^6$, while the best static sequence is $(1.2c)^9$. Figure 11 shows that the LLC occupancy estimated by Algorithm 2 using linear interpolation at each step of the best Occupancy Sequence (x-axis) matches closely with the LLC occupancy observed in detailed simulation.

Figure 12 shows the contribution of each LeakyRand component to the end-to-end time averaged over 500 strings of the test suite as the LLC capacity is varied. The data are normalized to the 4 MB LLC capacity point. The $|DS|$ values are the same as shown in Figure 10. Data communication and error correction take up most of the time, while the channel setup time arising from the CRFill and CRProbe steps is less than 10% across the board. Including the CRFill and CRProbe overhead, the LeakyRand channel achieves end-to-end bandwidth of 46.6 Kbps, 26.4 Kbps, and 12.6 Kbps respectively for 1 MB, 2 MB, and 4 MB LLC sizes. For a 2 MB LLC, LeakyRand's bandwidth is 3.3× to 5.7× of SPP's bandwidth in the presence of reference bits ("+ref" in Figure 4b).

Table 3. Occupancy Sequences generated by the recursive procedure (Algorithm 2) for occupying 99% of LLC.

Occupancy Sequence	Cyc. dev. (%)	Occupancy Sequence	Cyc. dev. (%)
$(1c)^3(1.1c)^1(1.2c)^6$	0.071	$(1c)^2(1.1c)^3(1.2c)^5$	0.039
$(1.1c)^2(1.2c)^7$	0.900	$(1c)^1(1.1c)^5(1.2c)^4$	0.215
$\mathbf{(1c)^1(1.1c)^2(1.2c)^6}$	**0.044**	$(1.1c)^2(1.2c)^7$	0.092
$(1c)^1(1.1c)^1(1.2c)^7$	0.076	$(1c)^1(1.2c)^8$	0.516
$(1.1c)^6(1.2c)^4$	0.239	$(1.1c)^3(1.2c)^6$	0.109

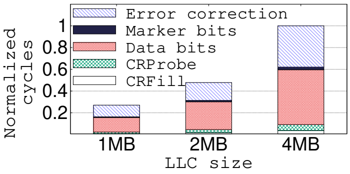


Fig. 12. End-to-end cycles of LeakyRand with OBP=98.88%, ECI=128, unrolling factor=128.

5.10 Extended Scope and Applications of LeakyRand

In this section, we discuss several applications of LeakyRand to setting up covert channels in different types of caches and to mounting fingerprinting attacks. These applications extend LeakyRand's scope beyond covert channel attacks on fully associative LLCs with random replacement.

5.10.1 Adoption of LeakyRand to Real-world Settings. A set of a set-associative cache can be viewed as a standalone fully associative cache with the number of entries equal to the associativity of the set-associative cache. In the following, we discuss how we use LeakyRand to construct a covert channel exploiting a set of the set-associative LLC in an Intel Xeon processor without any knowledge of the LLC replacement policy, thereby showcasing the generality of LeakyRand, its application to a real-world setting, and its robustness in the presence of operating system noise. Before mounting the LeakyRand attack on the Xeon processor, we conduct a cycle-level functional validation of LeakyRand by synthesizing it end-to-end on a field programmable gate array (FPGA).

FPGA Synthesis of LeakyRand: We synthesize a pipelined controller of a fully associative cache supporting read, write, flush, and random replacement operations. A pseudo-random number generator using a linear feedback shift register (LFSR) is employed to implement the random replacement policy. For cycle-level functional testing of the LeakyRand attack, we also synthesize the receiver and sender processes as two finite-state machines that interact with the cache controller. We synthesize our Verilog RTL on a Xilinx Spartan-3E FPGA. The synthesized fully associative cache has 16 entries. The CRFill step uses the dynamic cache region identification procedure. The cache occupancy achieved by the receiver after the CRProbe step averaged over all experiments is seen to be 98.8%. We experiment with $|DS| = 2, 4$ and ECI $= 8, 32, 128$. For ECI=8 and 32, LeakyRand achieves error-free communication for both values of $|DS|$. For ECI=128 and $|DS|$=2, the average bit error rate is 0.0009 and when $|DS|$ increases to 4, the communication becomes error-free.

Implementation on an Intel Xeon Processor: To mount the LeakyRand attack, we exploit a set of the 16 MB 16-way set-associative LLC shared among eight cores of an Intel E-2278G Xeon processor running commodity Linux operating system. First, the receiver and sender processes, pinned to two cores of the processor, construct the Occupancy Set and the Disturbance Set, respectively, both of which must map to a common set of the LLC. This construction follows the traditional protocol for the design of eviction sets [31]. Briefly, the receiver constructs its Occupancy Set by streaming through a large array of block addresses in its virtual address space and identifying the ones that conflict in an LLC set based on the observed latency of access. First, a pair of such conflicting addresses A_1 and A_2 is identified. In each subsequent step, the set of conflicting addresses is extended by adding a new address A_i that conflicts with the existing addresses $A_1, A_2, \ldots, A_{i-1}$ of the set. Once the receiver has constructed its Occupancy Set, the sender accesses a different array B (larger than the LLC size) of block addresses in its virtual address space. The receiver then accesses its Occupancy Set. Next, the sender accesses the array B and identifies the missing block addresses by observing the access latency. These missing addresses form a superset of the Disturbance Set, which is iteratively refined by the sender by checking each individual address for conflict with the Occupancy Set. We observe that different sets of the LLC experience different amounts of noise due to operating system and other activities. Therefore, different choices of the Occupancy Set and the Disturbance Set lead to differing fidelity of communication.

Once the Occupancy Set and the Disturbance Set have been identified, the LeakyRand attack executes the CRFill and CRProbe steps of the receiver to occupy a large portion of the set followed by the steps to communicate a bit string. Since the purpose of this exercise is to develop a proof-of-concept implementation of the LeakyRand attack on a real machine, we exclude error detection and correction from the implementation to keep it simple. We observe that the achieved occupancy is either 15 or 16 out of the 16 blocks in the set. Without any error correction, the attack achieves a bit error rate of 0.003 averaged across the entire test suite executed for 15 different configurations with $\{|DS|=1, 2, 4\} \times \{\text{unrolling factor}=1, 2, 4, 8, 16\}$ while offering bandwidth up to 570 Kbps. The significant increase in bandwidth compared to what we have seen in our simulation results is due to the small size (16 entries) of the fully associative cache (an LLC set) exploited in these experiments.

5.10.2 *Adoption of LeakyRand to LRU Caches.*

Having presented an application of LeakyRand to setting up a covert channel exploiting the shared LLC in a commercial processor with unknown replacement policy, we now turn to a case in which the replacement policy is known to the attacker and it is not the random replacement policy. We show that the Evict+Flush+Probe protocol of LeakyRand can be simplified and easily adopted to create a covert channel with significantly higher bandwidth in a fully associative LLC exercising the LRU replacement policy. The receiver fills the only LLC set with its Occupancy Blocks leaving aside space for the code and other data blocks of the receiver and sender. Importantly, exploiting the deterministic nature of the LRU replacement policy, the receiver positions the Occupancy Blocks toward the LRU side of the set. For communicating a 1 bit, the sender uses a Disturbance Set containing just one block B. Accessing B evicts the Occupancy Block in the LRU position and then the sender flushes B. The receiver probes the Occupancy Block in the LRU position to infer the received bit. Thus, the receiver does not need to probe the entire set of Occupancy Blocks as is done in the traditional Prime+Probe protocol. Evaluation of this attack on the simulated processor shows that our Evict+Flush+Probe protocol delivers an average communication bandwidth of 7.1 Mbps compared to 59 Kbps achieved by the traditional Prime+Probe on the test suite strings in a 2 MB LLC exercising LRU replacement policy.

5.10.3 *High-precision Fingerprinting Attack.*

LeakyRand can be used to mount high-precision fingerprinting attacks exploiting a fully associative LLC with random replacement policy. Since the CRFill and CRProbe steps allow the attacker to occupy most of the fully associative LLC, the attacker can easily detect even small differences in the LLC disturbance caused by different victim workloads making high-precision fingerprinting attacks possible. We show that using this attack it is possible to accurately distinguish between two inputs used by the md5 hash utility. One input uses a file $F1$ of size 8 KB and another uses a file $F2$ of size 20 KB. The attacker first uses the CRFill and CRProbe steps to occupy close to 99% of the LLC. Next, the victim uses the md5 hash utility to process $F1$ or $F2$. In the final step, the attacker uses the Probe step (of the receiver in LeakyRand) counting the number of unrolled iterations of the Probe step that experience LLC misses. Based on these LLC miss counts, the attacker infers whether the victim used $F1$ or $F2$ as input.

We evaluate this attack on our simulated processor. The attacker uses an unrolling factor of 16 in the Probe step. We conduct 250 samples of this attack using $F1$ and another 250 samples using $F2$ for different LLC replacement randomization seeds to train the best inference threshold C such that if the observed LLC miss count is less than C, the inference is $F1$; the inference is $F2$ otherwise. Note that we already have a pre-trained threshold τ applied to the measured latency of an unrolled iteration of the Probe step to infer whether the iteration suffers from an LLC miss or not. We similarly determine the best inference threshold when this attack is mounted using the Prime+Probe steps of SPP. We test our setup on 1000 execution samples of the attack employing different LLC replacement randomization seeds. The victim uses $F1$ 500 times and $F2$ in the remaining 500 cases. The fingerprinting attack using the CRFill, CRProbe, and Probe steps of LeakyRand leads to 100% accurate identification of $F1$ and $F2$ in all 1000 cases. LeakyRandGen can also be used to mount this attack if the cache line flush instruction is not available. We find that the fingerprinting attack using the CRFill and Probe steps of LeakyRandGen leads to 100% accurate identification of $F1$ and $F2$. However, the use of the Prime+Probe steps of SPP leads to 161 errors in identifying $F1$ and 143 errors in identifying $F2$ resulting in an overall 30.4% errors out of 1000 attack attempts. SPP's failure to occupy a large portion of the LLC is the primary reason for this large error rate.

6 Summary

We present LeakyRand that exploits a shared fully associative LLC exercising random replacement policy for setting up a covert communication channel between two co-scheduled processes (receiver and sender). An automatically synthesized cache region identification procedure enables the receiver to occupy a large portion of the LLC with probability nearly one. A novel Evict+Flush+Probe communication protocol allows the receiver to maintain its high LLC occupancy. Periodic invocation of low-overhead error syndrome detection and correction algorithms helps maintain an ultra-low bit error rate. On a 2 MB LLC, LeakyRand achieves a bandwidth of 26.4 Kbps while maintaining a bit error rate of 10^{-4} or less. This high-bandwidth high-fidelity channel exploiting the fully associative LLC with random replacement necessitates continued search for a more secure caching substrate.

Acknowledgments

We thank Pragati Agrawal and Dhruv Gupta for carrying out the FPGA synthesis of LeakyRand and demonstrating the attack on an Intel Xeon server. We also acknowledge the help of Abhishek Khandelwal and Pallav Goyal in setting up the Gem5 full-system infrastructure that we use for operating system noise studies. We thank the anonymous reviewers for their constructive feedback.

A Details of the Artifact

A.1 Abstract

The artifact includes the implementation of LeakyRand on the trace-based microarchitecture simulator ChampSim, full-system architecture simulator Gem5, Spartan-3E FPGA, and Intel Xeon server. We include scripts to run the experiments and generate outputs. The README of the repository contains the instructions for each setup [30]. The ChampSim and Gem5 codes have been tested on Linux systems with ubuntu distribution. To ease the setup process and portability, we provide a Docker image for the ChampSim infrastructure and ready-to-execute scripts that automate the execution of applications and plotting of graphs. The artifact is archived on Zenodo [29].

A.2 Artifact Checklist (Meta-information)

- **Algorithm:** Static and dynamic cache region identification procedures for the CRFill and CRProbe steps, and the LeakyRand covert communication protocol with error detection and correction procedures.
- **Inputs:** 750 random binary strings of length 512 each are used to generate ChampSim, Gem5, and Intel Xeon processor results. For functional testing of LeakyRand synthesized end-to-end on the FPGA, three types of input strings (all of length 1024 bits) are used, (*a*) all 1s, (*b*) 1011 repeated 256 times, and (*c*) 0010 repeated 256 times.
- **Compilation:** All required compilation dependencies for ChampSim are pre-installed in the Docker image. Gem5 requires GCC version 10 or above, python3 and scons.
- **Binary:**
  - The required build scripts or binaries are provided in the git repository [30] and Zenodo DOI archive page [29].
  - ChampSim: Scripts to build program binaries, generate instruction traces from program binaries using Pin tool, and build ChampSim are included.
  - Gem5: Pre-built linux kernel and ubuntu disk image with program binaries are included.
  - Intel Xeon: Commands for compiling and mounting the LeakyRand attack are included.
- **Run-time environment:** Linux distribution, Docker. Attack fidelity of LeakyRand on the Intel Xeon server improves if single-user login is used with sibling cores switched off. For synthesis of LeakyRand on Spartan-3E FPGA, the Xilinx ISE 14.7 toolchain is used.
- **Hardware:** Intel processors are used for simulation. The LeakyRand attack on the Xeon server should be executed on a multi-core processor with inclusive LLC. For FPGA synthesis of LeakyRand, the Xilinx Spartan-3E starter board is used.
- **Execution:** The artifact includes scripts to launch experiments and plot graphs.
- **Metrics:** For different experiments, we report channel bandwidth, bit error rates, LLC occupancy (in cache blocks or percentage of LLC capacity), normalized LLC miss counts, or normalized execution cycles.
- **Output:** Per-figure scripts (*script.sh*) are included in the git repository to generate either a plot or a result file.
- **Experiments:** *script.sh* per figure/result is used to launch experiments.
- **How much disk space required (approximately)?:** The ChampSim results may take around 1.5 to 2 TB of space if all experiments are run on a single machine.
- **How much time is needed to prepare workflow (approximately)?:** Around an hour including Gem5 compilation and docker setup.
- **How much time is needed to complete experiments (approximately)?:** More than a month if all the experiments are run sequentially. *num_prl* used in *script.sh* controls the number of parallel simulations, wherever applicable.
- **Publicly available?:** Yes
- **Archived (provide DOI)?:** Zenodo page URL: https://doi.org/10.5281/zenodo.16237834

A.3 Description

A.3.1 How to Access: The source code of LeakyRand is available in the git repository [30] and Zenodo archive page [29].

A.3.2 Hardware and Software Dependencies: For mounting the LeakyRand attack on Intel Xeon server, the 16 MB 16-way set associative inclusive LLC of an Intel E-2278G Xeon processor, clocked nominally at 3.4 GHz running ubuntu 18.04, is used. For ChampSim and Gem5 simulations, we use Intel Xeon x86-64 processors with 64 cores, 100GB RAM and 2.5TB disk space running Linux Ubuntu 20.04.5 LTS having support to run Docker containers. For FPGA synthesis, we use the Xilinx Spartan-3E starter board, the Xilinx ISE 14.7 toolchain, connecting cables, and cable driver.

A.4 Installation

- Install Docker.
- Download Docker image from Zenodo archive [29]. Use this image to start a Docker container.
- Inside Docker container, clone the git repository [30] and start simulations.
- All the steps with required commands are mentioned in the README of git repository.

A.5 Experiment Workflow

In the git repository, we provide the source code of our implementation and the bash scripts to generate outputs corresponding to the figures and results presented in the paper. For each figure, there is a separate directory named `fig*` that contains all the corresponding scripts. To start the simulations for generating a figure, one needs to switch to the corresponding directory and launch *script.sh* using the *./script.sh* command. The variable *num_prl* defined in *script.sh*, wherever applicable, launches those many parallel simulations, thus reducing the simulation time. To reduce the overall execution time of experiments, one can create multiple Docker containers (using the Docker image shared on Zenodo) and start parallel simulations for generating different results simultaneously. One Docker container, at a time, can run the simulations for generating only one result figure.

A.6 Evaluation and Expected Results

script.sh corresponding to a particular figure generates the plot for the result with the name *fig*.pdf* or stores the result generated in a text file in the same directory. The results shared in the paper are stored in a text file *our_data.txt* for reference.

A.7 Methodology

Submission, reviewing and badging methodology:

- https://www.acm.org/publications/policies/artifact-review-badging
- http://cTuning.org/ae/submission-20201122.html
- http://cTuning.org/ae/reviewing-20201122.html

We are applying for all the three independent badges: Artifacts Available, Artifacts Evaluated, and Results Validated.

References

[1] 2008. Pcacti Tool. https://sportlab.usc.edu/downloads/download.

[2] Anubhav Bhatla, Navneet, and Biswabandan Panda. 2024. The Maya Cache: A Storage-efficient and Secure Fully-associative Last-level Cache. In *51st International Symposium on Computer Architecture*. Buenos Aires, Argentina.

[3] Nathan Binkert et al. 2011. The Gem5 Simulator. *ACM SIGARCH Computer Architecture News* 39, 2 (2011), 1–7.

[4] Julia Borghoff et al. 2012. PRINCE–A Low-latency Block Cipher for Pervasive Computing Applications. In *18th International Conference on the Theory and Application of Cryptology and Information Security*. 208–225.

[5] Thomas Bourgeat et al. 2020. CaSA: End-to-end Quantitative Security Analysis of Randomly Mapped Caches. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture*. 1110–1123.

[6] Samira Briongos et al. 2020. RELOAD+REFRESH: Abusing Cache Replacement Policies to Perform Stealthy Cache Attacks. In *29th USENIX Security Symposium*. 1967–1984.

[7] Anirban Chakraborty et al. 2025. Systematic Evaluation of Randomized Cache Designs against Cache Occupancy. In *34th USENIX Security Symposium*. Seattle, WA, USA.

[8] Craig Disselkoen et al. 2017. Prime+Abort: A Timer-Free High-Precision L3 Cache Attack using Intel TSX. In *26th USENIX Security Symposium*. Vancouver, BC, 51–67.

[9] Nathan Gober et al. 2022. The Championship Simulator: Architectural Simulation for Education and Competition. (2022). doi:10.48550/arXiv.2210.14324

[10] Daniel Gruss et al. 2016. Flush+Flush: A Fast and Stealthy Cache Attack. In *Detection of Intrusions and Malware, and Vulnerability Assessment: 13th International Conference, DIMVA*. 279–299.

[11] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. 2015. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In *24th USENIX Security Symposium*. Washington, D.C., 897–912.

[12] Gregor Haas, Seetal Potluri, and Aydin Aysu. 2021. iTimed: Cache Attacks on the Apple A10 Fusion SoC. In *IEEE International Symposium on Hardware Oriented Security and Trust*. Tysons Corner, VA, USA, 80–90.

[13] Austin Harris et al. 2019. Cyclone: Detecting Contention-Based Cache Information Leaks Through Cyclic Interference. In *52nd International Symposium on Microarchitecture*. Columbus, OH, USA.

[14] Jason Kim, Jalen Chuang, Daniel Genkin, and Yuval Yarom. 2025. FLOP: Breaking the Apple M3 CPU via False Load Output Predictions. In *34th USENIX Security Symposium*. Seattle, WA, USA.

[15] Jason Kim, Daniel Genkin, and Yuval Yarom. 2025. SLAP: Data Speculation Attacks via Load Address Prediction on Apple Silicon. In *46th IEEE Symposium on Security and Privacy*. San Francisco, CA, USA.

[16] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clementine Maurice, and Stefan Mangard. 2016. ARMageddon: Cache Attacks on Mobile Devices. In *25th USENIX Security Symposium*. Washington, DC, USA, 549–564.

[17] Fangfei Liu et al. 2015. Last-level Cache Side-channel Attacks are Practical. In *2015 IEEE Security & Privacy*. 605–622.

[18] Chi-Keung Luk et al. 2005. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. 40, 6 (2005). doi:10.1145/1064978.1065034

[19] Samira Mirbagher-Ajorpaz et al. 2020. PerSpectron: Detecting Invariant Footprints of Microarchitectural Attacks with Perceptron. In *53rd International Symposium on Microarchitecture*. Athens, Greece.

[20] Antoon Purnal et al. 2021. Prime+Scope: Overcoming the Observer Effect for High-precision Cache Contention Attacks. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 2906–2920.

[21] Antoon Purnal, Lukas Giner, Daniel Gruss, and Ingrid Verbauwhede. 2021. Systematic Analysis of Randomization-based Protected Cache Architectures. In *2021 IEEE Symposium on Security and Privacy*. IEEE, 987–1002.

[22] Moinuddin K Qureshi. 2018. CEASER: Mitigating Conflict-based Cache Attacks via Encrypted-Address and Remapping. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 775–787.

[23] Moinuddin K Qureshi. 2019. New Attacks and Defense for Encrypted-Address Cache. In *Proceedings of the 46th International Symposium on Computer Architecture*. 360–371.

[24] Gururaj Saileshwar, Christopher W Fletcher, and Moinuddin Qureshi. 2021. Streamline: A Fast, Flushless Cache Covert-channel Attack by Enabling Asynchronous Collusion. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 1077–1090.

[25] Gururaj Saileshwar and Moinuddin Qureshi. 2021. MIRAGE: Mitigating Conflict-Based Cache Attacks with a Practical Fully-Associative Design. In *30th USENIX Security Symposium*. 1379–1396.

[26] Anatoly Shusterman et al. 2019. Robust Website Fingerprinting Through the Cache Occupancy Channel. In *28th USENIX Security Symposium*. 639–656.

[27] Wei Song et al. 2021. Randomized Last-level Caches Are Still Vulnerable to Cache Side-channel Attacks! But We Can Fix It. In *2021 IEEE Symposium on Security and Privacy*. IEEE, 955–969.

[28] Tarunesh Verma et al. 2022. These Aren't The Caches You're Looking For: Stochastic Channels on Randomized Caches. In *2022 IEEE International Symposium on Secure and Private Execution Environment Design*. IEEE, 37–48.

[29] Yashika Verma, Debadatta Mishra, and Mainak Chaudhuri. 2025. Archived LeakyRand Artifact. https://doi.org/10.5281/zenodo.16237834.

[30] Yashika Verma, Debadatta Mishra, and Mainak Chaudhuri. 2025. README of LeakyRand Artifact. https://github.com/YashikaVerma156/LeakyRand_CASES_2025.git.

[31] Pepe Vila, Boris Kopf, and Jose F. Morales. 2019. Theory and Practice of Finding Eviction Sets. In *IEEE Symposium on Security and Privacy*. San Francisco, CA, USA.

[32] Mario Werner et al. 2019. ScatterCache: Thwarting Cache Attacks via Cache Set Randomization. In *28th USENIX Security Symposium*. 675–692.

[33] Wenjie Xiong and Jakub Szefer. 2020. Leaking Information Through Cache LRU States. In *2020 IEEE International Symposium on High Performance Computer Architecture*. IEEE, 139–152.

[34] Fan Yao, Milos Doroslovacki, and Guru Venkataramani. 2018. Are Coherence Protocol States Vulnerable to Information Leakage?. In *2018 IEEE International Symposium on High Performance Computer Architecture*. 168–179.

[35] Yuval Yarom and Katrina Falkner. 2014. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *23rd USENIX security symposium*. 719–732.

[36] Jiyong Yu, Aishani Dutta, Trent Jaeger, David Kohlbrenner, and Christopher W. Fletcher. 2023. Synchronization Storage Channels (S2C): Timer-less Cache Side-channel Attacks on the Apple M1 via Hardware Synchronization Instructions. In *32nd USENIX Security Symposium*. Anaheim, CA, USA, 1973–1990.