Received 19 January 2023, accepted 11 February 2023, date of publication 16 February 2023, date of current version 23 February 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3246119

RESEARCH ARTICLE

CoDi$: Randomized Caches Through Confusion and Diffusion

LUÍS FIOLHAIS [1,2], **MANUEL GOULÃO** [3], **AND LEONEL SOUSA** [1,2], **(Senior Member, IEEE)**

[1] Instituto de Engenharia de Sistemas e Computadores-Investigação e Desenvolvimento (INESC-ID), 1000-029 Lisbon, Portugal
[2] Instituto Superior Técnico, Universidade de Lisboa, 1049-001 Lisbon, Portugal
[3] Instituto de Telecomunicações, Department of Mathematics, Instituto Superior Técnico, Universidade de Lisboa, 1049-001 Lisbon, Portugal

Corresponding author: Luís Fiolhais (luis.azenhas.fiolhais@tecnico.ulisboa.pt)

This work was supported in part by the National Funds through the Fundação para a Ciência e a Tecnologia (FCT), Instituto de Engenharia de Sistemas e Computadores: Investigação e Desenvolvimento (INESC-ID), Instituto de Telecomunicações (IT), under Grant UIDB/50021/2020 and Grant UIDB/50008/2020; and in part by the European Union, Scaling extreme analYtics with Cross-architecture acceLeration based on OPen Standards (SYCLOPS), under Agreement 101092877. The work of Luís Fiolhais was supported by FCT, Portugal, under Grant SFRH/BD/145477/2019.

ABSTRACT Cloud providers are increasingly exposed to malicious actors through transient attacks, such as Spectre and Meltdown. The cache hierarchy is the main target to build the required side-channels to leak data. Randomized caches can be employed to provide security but often rely on cryptographic primitives to deter side-channel attacks. These increase the access latency and deteriorates the system performance. This paper shows that randomized caches do not have to increase the cache access latency, and that their security does not have to rely on a cryptographic hash function or block-cipher. Herein, CoDi$ is proposed, a randomized last level cache that achieves security by tying the local and global states. Security is achieved through a higher miss energy consumption and occupied area, instead of penalizing performance. CoDi$ is able to evict any cached address by allowing addresses to be displaced in two levels of freedom, through hopscotch (local state) and cuckoo hashing (global state), without increasing the hit latency. Through these displacements, paths can be built to the eviction address. The security of CoDi$ relies on the hardness to control all possible displacement paths when a miss occurs, which requires control over the local and global states of the cache simultaneously. Confusion is generated when evicting an address, as there are many possible cache states that could result from this eviction. Also, for each executed eviction, multiple addresses in the cache are non-deterministically displaced, providing diffusion. Moreover, experimental analysis for a 48-bit secure CoDi$, using SPEC, NPB, and Polybench benchmarks, shows improvements in the number of instructions per cycle and up to 5% in misses, when compared to two state-of-the-art randomized caches.

INDEX TERMS Covert channels, last level cache, randomized cache, side-channel attacks.

I. INTRODUCTION

As the proliferation of off-loading compute intensive tasks continues to grow, either through Internet-of-Things devices or Software-as-a-Service, cloud platforms become a larger target for malicious agents. It has been shown that these platforms, despite following the best practices for task isolation, are susceptible to leaking information through transparent micro-architectural states [1], [2], [3], [4], [5], [6], [7]. An attacker with knowledge of which micro-architectural component is leaking information can target it. Specifically, an attacker seeks to build a covert communication channel with the victim through an insecure micro-architectural component that has high bandwidth and low noise [8], [9].

There are multiple micro-architectural components that have been shown to leak data [1], [6], [10]. However, none is as ubiquitous as caches in the memory hierarchy. Due to the high latency between the micro-architecture and the external memory, a hierarchy of caches is built to reduce the latency of most memory accesses, increasing the performance of the system. Since the usage of a cache hierarchy is a necessity in modern systems to provide good performance, it is the main component exploited to build covert communication channels. Employing an attack on private caches (L1, L2) is harder, as it requires the attacker and victim to be sharing the same physical core. The Last Level Cache (LLC), however,

The associate editor coordinating the review of this manuscript and approving it for publication was Songwen Pei.

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License.
For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

is physically shared between all cores in the system. Due to the LLC being a global shared resource, it is the main targeted micro-architectural component. Thus, this work focuses on the development of a secure LLC, and assumes that the private caches are already secured by other means [11], [12].

Creating communication channels in the cache hierarchy has been an extensively researched topic [8], [9], [13], [14], [15], [16], [17], [18], [19], [20], [21], [22]. Generally, the receiver (attacker) sets up a specific state in the cache. The sender (victim) transmits data, if some secret dependent data is accessed, by modifying the cache state. The attacker can infer what data was transmitted, *i.e.*, how the cache state was modified, by timing specific memory accesses.

This paper proposes a secure LLC, CoDi$, based on the observation that, for a communication channel to be successfully deployed, the attacker must know *how any victim modification creates the resulting state*. Failing to understand how certain states are reached implies that there are transmissions the attacker can not infer, introducing *confusion*. Therefore, a secure cache will modify the state of multiple addresses in the cache on every access, introducing *diffusion*, such that the attacker state is randomly modified. However, changing the state on every access may also change the cache hit latency. For the cache to be performant and practical, the hit latency needs to be constant and low. Therefore, the modifications to the state must not alter the hit latency.

CoDi$ chooses to perform the state modifications only on misses instead of on every access. This method allows CoDi$ to keep the performance of the system, while still offering a secure cache. Contrary to state-of-the-art caches, CoDi$ does not degrade the performance to achieve a secure cache. Instead, it uses more energy and occupies a larger area to attain this goal. Modifications to the state are performed on misses, in order to hide the latency of the modifications behind the latency of the external memory. To achieve this goal, CoDi$ allows addresses to be displaced across the cache, while still maintaining the same hit latency. A consequence of the displacement mechanism is that any address can be chosen for eviction, because CoDi$ can build any sequence of displacements to reach the eviction address. Thus, the security of CoDi$ lies on the sequence of displacements, or number of modifications, performed to the cache state.

A. CONTRIBUTIONS

The main contributions are: *i)* an LLC architecture that is robust against side-channel attacks; *ii)* the LLC is designed as not to increase the hit latency or degrade the performance of a system; *iii)* the security model does not rely on cryptographic primitives (*e.g.*, cryptographic hash functions, block ciphers), does not require key refreshing periodically, and does not partition the cache between processes.

B. ORGANIZATION

The paper is organized in eight sections. The second section describes the background concepts required. Section three introduces the proposed cache, CoDi$, and the fourth section goes into detail on how CoDi$ should be designed. Section five provides a security discussion and a theoretical security model. Section six gives estimates for an implementation of CoDi$, and analyses its performance, with two comparisons to state-of-the-art randomized caches. Section seven contextualizes CoDi$ with other state-of-the-art proposals, and the last section draws the main conclusions.

II. BACKGROUND

A. CACHES AND MEMORY HIERARCHY

Modern cores require all of its pipeline stages to be operating over data to achieve a high instruction throughput. Due to the high latency of the external memory and the high operating frequency of the core, most of its time is spent waiting for data from the external memory. In order to reduce the data access latency, caches are introduced between the core and the external memory, with larger capacities and latencies as the distance to the core increases. The cache hierarchy is a physically shared resource by multiple threads. Typically, such a cache hierarchy is composed of three levels, being the first two private to a core and the third shared between cores.

Caches are self-managed memories that store recently accessed data, and data that may be accessed in the future. A cache memory is split into sets and ways. A cache access selects a set (index) through a hash function. A set contains multiple ways where data can be stored. The number of ways in a set indicates the number of candidates (associativity) that can be considered for eviction. The replacement policy selects which candidate is evicted. A cache with more than one set and more than one way is referred to as a *set-associative cache*. A cache access follows three steps: *i)* hash the memory address to obtain a pointer to a set; *ii)* access the set and compare the stored addresses with the input memory address; *iii)* if the memory address matches any stored address (*hit*) return the data to the core, else (*miss*) fetch the missing data from external memory, evict one address from the set, store the fetched data in the set, and return the data to the core.

B. COVERT COMMUNICATION CHANNELS

A covert communication channel requires two processes, a receiver (*attacker*) and a sender (*victim*), whose memory access patterns implicitly reveal secret data. When the victim accesses the secret-dependent memory addresses, it is transmitting data by modifying the state of the cache. The attacker will manipulate the state of the cache, such that the transmissions of the victim can be identified.

Data transmission has three stages: *setup*, *wait* and *analysis* [23], [24]. In the setup stage, the attacker sets the cache to a known state; in the wait stage, the attacker waits for the victim to modify the cache state; and, in the analysis stage, the attacker verifies whether the modification caused by the victim affected its known state. Current literature [23], [24] defines two types of setups: *set conflict* and *shared memory*.

1) SET CONFLICT

Set Conflict setups exploit cache set sharing between different processes. The attacker needs to have knowledge of the hash function to target a specific set. If a cache set is fully occupied, then one line is evicted according to the replacement policy. As such, an attacker can exploit the number of replacement candidates in the cache set (associativity) to force the victim to evict the data of the attacker, *i.e.*, the attacker creates an eviction set. Then, if the victim transmitted data, one of the accesses of the attacker to the same set will have high latency [4], [15]. Furthermore, since an attacker can fill a cache set with its data, the replacement policy can be controlled. Controlling the replacement policy allows the attacker to select which address will be evicted when the victim accesses the cache [9], [24], [25].

2) SHARED MEMORY

Shared memory setups rely on deduplicated shared memory in the cache, between the attacker and victim [3], [18]. Flushing shared data removes it from all caches in the hierarchy. When the sender accesses the address again, the data will be placed in the hierarchy. Therefore, the receiver can measure the latency of a load [18] or flush [8] instructions to the same memory location to detect a transmission.

Secure caches aim to block both setups. However, one setup can be reduced into the other. The shared memory channel can be reduced into a set conflict channel by allowing data to be duplicated in memory when it belongs to a different process. This can be achieved by XORing the output of the hash function with a private process identifier [26], [27]. This private nonce is generated and managed by a Trusted Execution Environment (TEE) in hardware. This way, each process has a private and unique copy of a shared page, and flush instructions will only remove the cache data from the process that issued the instruction. The downside of duplicating shared memory in the cache hierarchy is that writable shared memory cannot be cacheable as that would require one store instruction to update multiple cache lines.

C. SECURE CACHES

The state-of-the-art literature defines two types of caches that prevent the creation of covert communication channels: *partitioned caches* and *randomized caches*. *Partitioned caches* split the cache between processes. A process that is not the owner of a partition cannot modify the cache state of another process [26], [28], [29], [30]. *Randomized caches* decorrelate the output (evicted address) of a transmission channel from the input (address inserted). All processes can modify the cache state of other processes, but the interaction that makes one process action affect the cache state of other processes is hard to determine. Until now [27], [31], [32], [33], randomized caches relied on cryptographic primitives for this behavior, but CoDi$ shows that this strong security constraint may be relaxed.

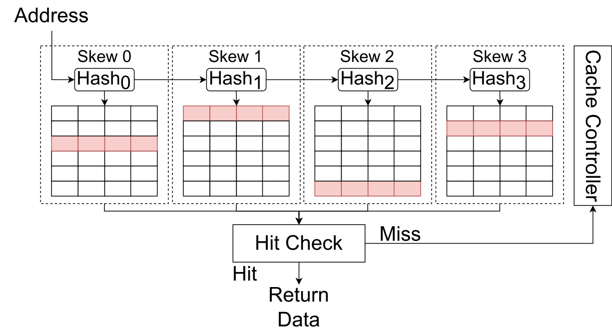

FIGURE 1. **High-level view of CoDi$.**

D. THREAT MODEL

In the design of the CoDi$ LLC, the following assumptions are made: *i)* all LLC parameters are public, including the family of hash functions; *ii)* when querying the LLC, an attacker only learns if a hit or miss have occurred (by timing the memory access); *iii)* the system has multiple threads running and competing for time to execute on the core and cache lines; *iv)* an attacker is a possibly privileged multi-threaded process; *v)* an attacker and victim do not run in the same physical core; *vi)* an attacker and victim may have a shared memory region; *vii)* the cores in the system may be susceptible to transient attacks [1], [2], [5], [6].

III. PROPOSED CACHE

CoDi$ is a full-associative randomized cache. It performs random address displacements on a miss, to change the state of the cache. This forces a state setup by an attacker to be randomly modified. The modifications to the cache state are concealed by the latency of the external memory response, as not to affect the performance of the system.

Similarly to other randomized caches, CoDi$ is split into multiple set-associative caches, where each cache uses a different hash function (Fig. 1). Here, each set-associative cache is referred to as a *skew* [34]. The hash functions are selected uniformly at random from a universal family, as to minimize collisions. Universal hashing [35] is used, as this selection is done by sampling a secret uniform random seed (*e.g.*, 64 bits sampled during boot), and the outputs of the hash functions are never revealed.

Herein, the state that is modified on misses is interpreted as two separate entities. On one hand, there is the *global state*, which comprises the state of the entire cache. On the other hand, the *local state* only covers the state of a subset of the cache, each independent of every other. Traditional caches use the global state to derive a local state, *e.g.*, a hash function mapping to some subset. However, it is a one-way interaction, as the inverse is not allowed, *i.e.*, a local state cannot modify the global state. Therefore, attacking these caches targets both states separately, *i.e.*, the best-case complexity of an attack is the maximum probability of attacking either the global or the local state. Due to this separation of states, noise has less impact in the security model of these unidirectional caches.

CoDi$ is designed with the security concepts of **Con**fusion and **Di**ffusion [36] in mind (hence the name CoDi), where the overall (local and global) state of the cache is the secret, and is changed on all cache misses. First, to add *confusion*, ambiguity in the resulting overall state is created by allowing the eviction address to be reached through many different paths, chosen non-deterministically. Then, *diffusion* acts by changing the overall state of the cache by displacing addresses through every skew and within each skew. Confusion and diffusion are achieved by connecting the global and local states bidirectionally, *i.e.*, modifying the local state modifies the global state and vice-versa. Here, the best-case complexity for an attack is the joint probability of attacking the global *and* local states. Also, since the global and local states are now connected, noise plays an enhanced role in defining the security guarantees of the cache, which is discussed in Section V-B.

Since creating a covert communication channel reduces to an eviction set creation problem, the attacker relies on finding enough addresses to occupy the same set as the victim, to control the eviction candidates. Traditional randomized caches are designed to make these addresses hard to find [31]. Instead, CoDi$ opts to change the overall state of the cache in a random manner, by allowing addresses to be displaced on a miss. The displacements break the relation between the set where the address is going to be allocated in the cache and the eviction address. In addition, CoDi$ allows any address of the cache to be evicted. The hard problem for an attacker is guessing the resulting overall state of the cache after the addresses displacements. To allow the displacements CoDi$ adopts two hashing schemes: cuckoo [37] and hopscotch [38].

A. GLOBAL STATE: CUCKOO HASHING

Cuckoo hashing [37] splits the cache into (s) skews. On a cache miss, addresses can be displaced between skews. Different addresses will define different sets (with high probability). Still, even though the sets are different, each address can collide with parts of the set of other addresses. These collisions are what allow displacements between skews without changing the hit latency [39]. When displacing an address, its set remains the same, but the state of the cache changes. Displacing an address to a different skew introduces a new collision with another address. Therefore, the displacement procedure can be recursively applied using different addresses each time. The procedure creates a graph of address collisions. So, evicting an address from the graph requires following the sequential collisions, moving the addresses between skews, until the eviction address is reached.

Fig. 2 a) shows a cache miss with address `x`. The cache has 2 skews and 1 way per skew. Each address shows the indices, $\{x, y\}$, to where it can be moved to, where x is the set number in skew 0 and y is the set number in skew 1. Address `x` collides with `ae` (set 1 of skew 0) and `aj` (set 2 of skew 1), and can be stored in either location. Addresses `ae` and `aj` can be further displaced themselves. Address `ae` can move

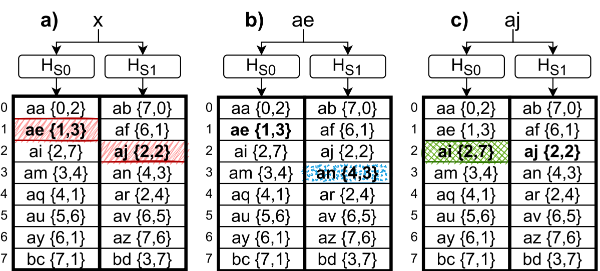

FIGURE 2. Cuckoo hashing example for a cache with 2 skews and 1 way per skew. Each skew is within a black frame.

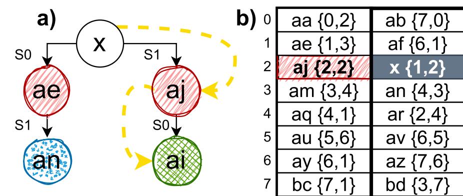

FIGURE 3. a) Eviction candidates from the example in Fig. 2. The replacement path is in a dashed arrow. b) cache state after applying the replacement path.

to set 3 of skew 1 (Fig. 2 b)), and `aj` can move to set 2 of skew 0 (Fig. 2 c)). If the graph of address collisions stopped here, with a depth of 2, Fig. 3 a) shows all eviction candidates. Here, the replacement policy selected `ai` for eviction and displaces the addresses in its path. The path to eviction is `x` $\rightarrow$ `aj` $\rightarrow$ `ai` (dashed arrows in Fig. 3 a)). Address `x` occupies the position previously held by `aj` in skew 1. `aj` is displaced from skew 1 to skew 0 and occupies the position previously held by `ai`. `ai` is the eviction address, its contents are written back to memory or dropped. Fig. 3 b) shows the cache state after performing the displacements.

B. LOCAL STATE: HOPSCOTCH HASHING

CoDi$ allows any address in the cache to be evicted. However, cuckoo hashing requires too many displacements to reach the eviction address before the memory controller responds with the missing data. For CoDi$ to be fully-associative, more freedom of movement is required. This is achieved by allowing displacements within the same skew, and hopscotch hashing [38] adds that freedom.

The main component in hopscotch hashing is the neighborhood (of size h). Herein, a *neighborhood* is a group of two sequential sets of the same skew. The base index of the neighborhood is called the *Neighborhood ID (nID)*. The neighborhood is given by the concatenation of the ways of the sequential sets (sets are i and $i + 1$). As a result, neighborhoods overlap between each other. This overlap allows addresses of one neighborhood to displace addresses of other neighborhoods, *i.e.*, addresses can change sets in the same skew, while still in the same neighborhood. The properties of neighborhoods relax the placement of addresses within a skew, which doubles the number of ways available for an

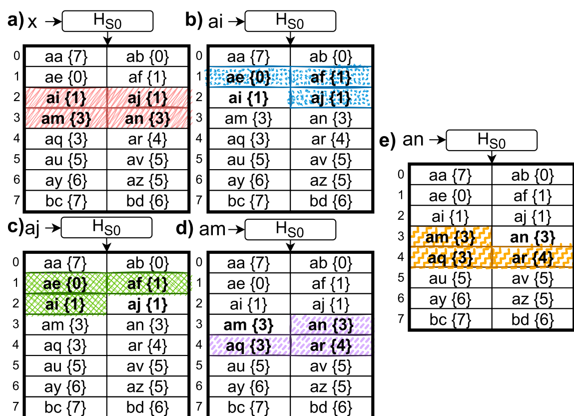

FIGURE 4. Hopscotch hashing example for a cache with $s = 1$, $h = 4$ (2 ways per skew), and the nID is in braces for each address. Each skew is within a black frame.

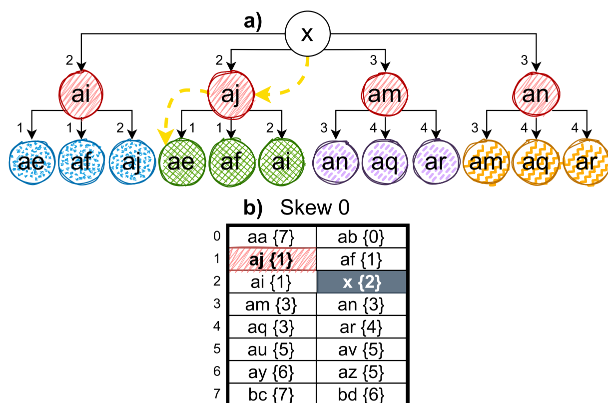

FIGURE 5. a) Eviction candidates from the example in Fig. 4. The replacement path chosen is in a dashed arrow. b) cache state after applying replacement path.

address to be placed in. Thus, a graph of address collisions can also be built from multiple successive displacements.

Fig. 4 shows an example of a cache with hopscotch hashing, a single skew (s) and a neighborhood size (h) of 4 (2 ways per skew). Each address shows the nID, $\{x\}$, where they can be displaced to. For example, an address with nID 1 can reside in sets 1 and 2. An address with this property is `ai` in set 2. Fig. 4 a) shows address `x` accessing nID 2, missing, and colliding with addresses `ai`, `aj`, `am`, and `an`. Since some addresses in the collision have different nIDs, they can be displaced to different sets. Addresses `ai` and `aj` can move to set 1, and addresses `am` and `an` can move to set 4. Figs. 4 b), c), d), and e) show where the addresses in the collision of `x` can be displaced to. From these collisions, similarly to cuckoo hashing, a displacement graph can be built (Fig. 5 a)). The replacement policy selects address `ae` to be evicted and displaces the addresses in its path to the eviction address. The path to eviction is `x` → `aj` → `ae` (dashed path in Fig. 5a)). The replacement procedure is the same as in the cuckoo method. The cache state after applying the replacement path is shown in Fig. 5b).

C. COMPOSING A RANDOMIZED LLC

CoDi$ is implemented by merging together cuckoo and hopscotch hashing. The addresses are placed in the cache whose structure is seen as two-dimensional, as they can be placed in their respective randomly assigned neighborhood in any skew. Once a miss occurs, the goal is to find a path (successive displacements of addresses until the eviction address) to a pre-selected eviction address. To improve this path finding procedure, an interval Δ around the pre-selected eviction address is defined. Defining Δ, named *vicinity*, guarantees that the eviction address is within reach by applying a given number of *vertical moves* (hopscotch hashing). Therefore, prior to performing any movement, the controller can verify if an address can be *horizontally moved* (cuckoo hashing) into this vicinity, by checking whether its nID for that skew is in Δ. Performing this optimization mitigates the reliance on the controller executing an exact movement into a neighborhood with the eviction address. Hence, considering the concept of vicinity increases the likeliness that a path to the eviction address is found.

Specifically, a valid Δ collision occurs when the nID of an eviction address is in $\Delta = [e - \delta, e + \delta] \mod N$, where N is the number of neighborhoods per skew and δ is the maximum number of vertical moves within the vicinity, around the eviction address e. Note that Δ is independent of the neighborhood size (h), although both define regions of interest in a skew. The neighborhood indicates the number of places where an address can reside, where Δ indicates to the cache controller the vertical movement proximity to reach the eviction address.

An access to CoDi$ is described in Algorithm 1. A cache access first hashes the memory address for each skew (Algorithm 1, 4) to obtain the addresses stored in each neighborhood (Algorithm 1, 5-6). Then, for each way in the neighborhood, the cache checks if any of the addresses is equal to the requested address (Algorithm 1, 7-13). If there is a hit, the data is returned (Algorithm 1, 9, 11). Otherwise, the controller stores the inspected addresses (Algorithm 1, 14) and handles the miss for the inspected addresses. This miss prompts a change in the overall state of the cache.

When a miss is detected, Algorithm 2 is executed. Initially, the CoDi$ controller selects the eviction address (Algorithm 2, 4). The eviction address selection is a two step process. First, the controller will select one neighborhood (h addresses) uniformly at random from every skew. Second, from this random selection of neighborhoods, the replacement policy selects which address is going to be evicted. After the selection of the eviction address, the cache controller will attempt to find a path to the eviction address by randomly displacing addresses (Algorithm 2, 15-21). The search for a path is hidden behind the latency of the external memory response, so a path must be found within that time frame (Algorithm 2, 5). However, the displacements required to reach the eviction address may not be available due to the current overall state of the cache. In such a case, the cache controller defaults to evicting one of the previously inspected

Algorithm 1 Basic Operation of CoDi$

1: **function** cache_access(addr)
2: insp_nodes ← []
3: **for** s < skews **do**
4: I_1 ← hash(s, addr)
5: $I_2 \leftarrow (I_1 + 1)$ mod sets ▷ nIDs for skew s.
6: set_addrs_1 ← cache_addrs[s][I_1] ▷ Neigh. 1st set
7: set_addrs_2 ← cache_addrs[s][I_2] ▷ Neigh. 2nd set
8: **for** w < ways **do**
9: **if** set_addrs_1[w] == addr **then** ▷ Hit!
10: **return** cache_data[s][I_1][w]
11: **else if** set_addrs_2[w] == addr **then**
12: **return** cache_data[s][I_2][w]
13: **end if**
14: **end for**
15: insp_nodes.push(set_addrs_1, set_addrs_2)
16: **end for** ▷ Cannot find address. Miss!
17: **return** HANDLE_MISS(addr, insp_nodes)
18: **end function**

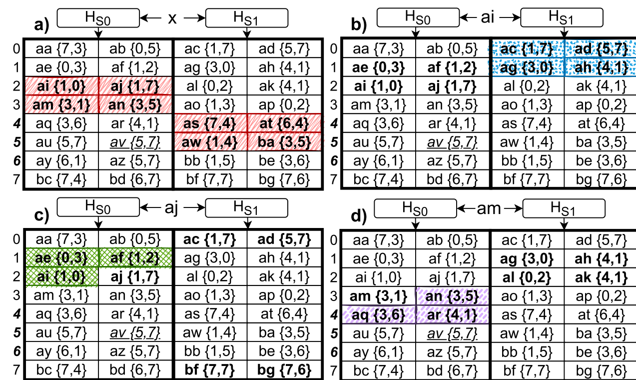

FIGURE 6. CoDi$ example with $s = 2$, $h = 4$ (2 ways per skew), $\delta = 1$. The nIDs $\{x, y\}$ mean x for skew 0, y for skew 1.

Algorithm 2 CoDi$ Miss Handling

1: **function** handle_miss(addr, insp_nodes)
2: index ← 0
3: found_path ← `false`
4: (ev_addr, ev_addr_s) ← replacement_policy()
5: **while** !ext_mem_resp && !found_path **do**
6: current_addr ← insp_nodes[index]
7: I_{ev1} ← *hash*(ev_addr_s, current_addr)
8: $I_{ev2} \leftarrow (I_{ev1} + 1)$ mod sets
9: **if** $I_{ev1} \in \Delta$ || $I_{ev2} \in \Delta$ **then**
10: found_path ← perform_vertical_moves()
11: **if** found_path **then**
12: **break**
13: **end if**
14: **end if** ▷ Not in Δ or path not found.
15: s ← random() mod skews ▷ Random move
16: I_1 ← *hash*(s, current_addr)
17: $I_2 \leftarrow (I_1 + 1)$ mod sets
18: **for** w < ways **do**
19: insp_nodes.push(cache_addrs[s][I_1][w])
20: insp_nodes.push(cache_addrs[s][I_2][w])
21: **end for**
22: index++
23: **end while**
24: ev_addr ←!found_path ? random(insp_nodes) : ev_addr
25: displace(ev_addr, addr)
26: **return** data
27: **end function**

addresses uniformly at random (Algorithm 2, 24). The probability of this event occurring is analyzed in Section IV.

The graph of collisions for the miss is created following a depth-first approach. After the eviction address is selected and after failing the vicinity check (Algorithm 2, 7-14), the cache controller will uniformly pick a move for each address in the collision. The random choice is for a movement to a skew (Algorithm 2, 15). If the randomly chosen skew is equal to the current skew, the address will perform a vertical move. If the randomly chosen skew is different than the current skew, the address will perform a horizontal move.

Fig. 6 shows a miss handling example where CoDi$ has 2 skews, a neighborhood size of 4 (2 ways per skew), and a $\delta = 1$. Address `x` accesses the cache and misses (Fig. 6a)). To handle the miss, first, the cache controller randomly selects one neighborhood from each skew to form the eviction candidates. In this instance, from the eviction candidates, the replacement policy selects the address `av`, in nID 5 of skew 0, to be evicted (Algorithm 2, 4). δ is 1, so the controller will look for nIDs collisions in the vicinity $\Delta = [4, 6]$ of skew 0. Now that the eviction address is fixed, the cache controller will attempt to find a path starting from the collisions of address `x` (`ai`, `aj`, `am` and `an` from skew 0, and `as`, `at`, `aw` and `ba` from skew 1). The first inspected address is `ai` (Fig. 6 b)) (Algorithm 2, 6), which does not intersect with Δ (Algorithm 2, 7-9). The controller randomly chooses a horizontal move to skew 1 ($nID = 0$) (Algorithm 2, 15). The controller stores the collision addresses for later inspection (Algorithm 2, 16-20) and moves on to the next address in the initial collision, `aj` (Algorithm 2, 21, 6). Its nID is also not in Δ (Algorithm 2, 7-9). Here, `aj` (Fig. 6 c)) randomly performs a vertical move (skew 0, $nID = 1$) (Algorithm 2, 15). The controller follows the same procedure as address `ai` and moves on to address `am`. This time `am` is in Δ (Fig. 6 d)). Therefore, the cache controller can perform vertical moves to reach the eviction address `av` (Algorithm 2, 10). Specifically, `am` can perform a vertical move to `ar` and `ar` can perform another vertical move to `av`. This is possible because `ar` has an nID of 4 in skew 0, which means it can be moved to set 5 where `av` is. Thus, the controller has found the path $x \rightarrow am \rightarrow ar \rightarrow av$ (Fig. 7 c)). Figs. 7 a) and b) show the

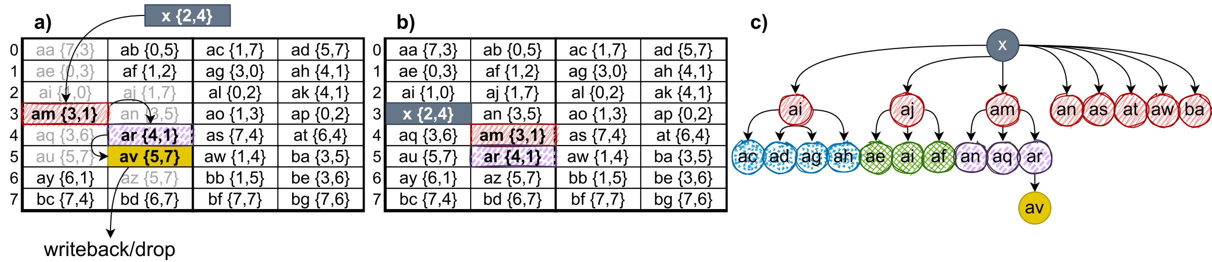

FIGURE 7. CoDi$ state after finding a path. $x \rightarrow am \rightarrow ar \rightarrow av$ (before in a), after in b)). c) shows all inspected nodes.

state of the cache before and after performing the addresses displacement.

IV. DESIGN SPACE EXPLORATION

The example in Fig. 7 shows a small cache where a path to the eviction address was found after inspecting 3 neighborhoods. For larger caches it rarely is that fast. There are many paths to the eviction address, with variable length. Each path that is built is unique to the current cache state, because the collisions generated by the controller depend on which addresses are stored in a given neighborhood. Therefore, instantiating CoDi$ requires balancing the number of skews (s), the size of the neighborhood (h), and the size of the vicinity ($|\Delta| = 2\delta + 1$). These should be set not only with security in mind, but also to increase the likelihood of finding a path.

A. PROBABILITY OF FINDING A PATH

The CoDi$ miss handling algorithm (Algorithm 2) is a probabilistic algorithm that depends on the state of the cache. Let M be a random variable with support $\{v, h\}$, representing the event of making a vertical (v) or horizontal (h) move.

First, horizontal moves can always be performed to the skew of the eviction address. An address will perform a horizontal move to the skew of the eviction address if its nID is in Δ ($|\Delta| = 2\delta + 1$). Since the outputs of the hash functions are uniformly distributed and each horizontal move is independent, the probability of colliding with the Δ interval is $\Pr(M = h) = (2\delta + 1)/N$, where N is the number of neighborhoods in one skew.

Once within Δ, the controller must perform sequential vertical moves to reach the eviction address. Depending on the location of the eviction address, the controller will want to move up or down the interval towards this address. A successful move up implies that the current nID can be modified to nID $-\,1$, and a successful move down implies that the current nID can be modified to nID$\,+\,1$. Consider the 4 neighborhood states in Fig. 8 for nID t. In Fig. 8 a), addresses `ac` $\{\mathtt{t} - 1\}$ and `ad` $\{\mathtt{t} - 1\}$, in the first half of the neighborhood, can be moved up, and address `ah` $\{\mathtt{t} + 1\}$ can be moved down. The neighborhood state depicted in Fig. 8 b) shows that there is a move up. The first half of the neighborhood contains an address with a nID of the previous neighborhood, `ae` $\{\mathtt{t} - 1\}$. But, there is no move downwards, as `ai` $\{\mathtt{t}\}$ and `aj` $\{\mathtt{t}\}$

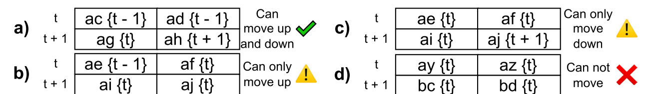

FIGURE 8. Four possible neighborhood states and vertical movement options for nID t. nIDs are in braces.

cannot move further down to an nID higher than $\{\mathtt{t} + 1\}$. A similar instance occurs in the neighborhood state depicted in Fig. 8 c). The first half of the neighborhood blocks any upwards movement, but the second half of the neighborhood allows a move down. Lastly, Fig. 8 d), shows a neighborhood state where no vertical movement is possible, as all addresses belong to the same neighborhood $\{\mathtt{t}\}$.

Recall that a neighborhood is composed of two sets from one skew and, in each, there can only be two nIDs: t and $t - 1$ in the first half, or t and $t + 1$ in the second half. So, the total number of possible states in half a neighborhood is $2^{h/2}$. Moreover, a vertical movement is possible except in the single case where there is a presence of a set wherein all nIDs are the same for that skew. Thus, a vertical move depends on half the neighborhood. Since (from the universality of the hash functions) vertical moves are statistically independent and each nID has the same probability of occurring, then the probability of performing a vertical move is given by $\Pr(M = v) = 1 - 1/2^{h/2}$ ($h \geq 2$ and even). Note that, since the moves are always performed in the direction of the eviction address, there is only one possible arrangement that blocks the movement in that direction. Indeed, reaching the eviction address in the Δ interval relies on multiple vertical moves happening sequentially. The worst case scenario would be performing δ individual vertical moves from the limit of Δ.

There are three possible scenarios for which the eviction address can be reached: a horizontal move is performed to the eviction address nID ($\Pr = 1/N$), a horizontal move is performed to the eviction address immediately preceding nID ($\Pr = 1/N$), or a horizontal move is performed to Δ and one or more vertical moves are performed to the eviction address nID. The probability of performing a horizontal move to the neighborhood of the eviction address is $2/N$. The probability of performing a horizontal move and a vertical move is $1/N \times \Pr(M = v)$, and for m vertical moves is $1/N \times (\Pr(M = v))^m$. Let F be a random variable with support

$\{t, f\}$, which represents the event of Algorithm 2 returning with found_path = true (t) or found_path = false (f). Then, (1) gives the probability of finding a path to the eviction address,

$$\Pr(F = t) = \frac{2}{N} + \frac{1}{N}\sum_{i=1}^{\delta}(\Pr(M = v))^{i}. \quad (1)$$

Finally, finding a path to the eviction candidate is modeled by a binomial distribution with probability of success $\Pr(F = t)$. So, the probability of finding at least one path given μ, the *maximum number of inspected neighborhoods*, is $1 - \Pr[X = 0]$, where $X \sim B(n = h\mu, p = \Pr(F = t))$.

Therefore, increasing the number of inspected neighborhoods μ results in a greater probability of success. Also, the probability of finding a path increases by decreasing the number of neighborhoods in a skew (N) or by increasing the probability of a vertical move ($\Pr(F = t)$). In turn, this means increasing the size of the neighborhood (h). Increasing δ also improves $\Pr(F = t)$, however, each higher order term will have less influence on the result as δ increases.

B. EXPERIMENTAL VERIFICATION

Section IV-A detailed the baseline statistical model to perform a sequence of movements to reach any eviction address in the cache. The assumption is that the overall state of CoDi$ will tend towards uniformly distributed nIDs in all neighborhoods as more cache misses occur. The theoretical model suggests that to increase the probability of finding a path requires increasing the neighborhood size (h), increasing δ, or decreasing the neighborhoods per skew (N). However, there are two constraints that are not modeled: the limited number of inspected neighborhoods prior to the arrival of the external memory response, and that each application programs the global state of the cache differently. Even though inspecting a large number of addresses yields a path to the eviction address with high probability, that may not be possible. Since each application programs the overall state of the cache differently, the vertical movements that are required to reach the eviction address may not be available. Due to these new constraints, the theoretical model needs to be compared with experimental results to understand how s, h, and δ relate to each other in practice. Moreover, since in practice the amount of time to inspect neighborhoods is limited, the number of inspected neighborhoods needs to be related to all variables.

To verify these relations, 19 SPEC [40] benchmarks are executed with two configurations of an 8MiB CoDi$ LLC. One configuration has a smaller number of skews and a larger neighborhood ($s = 4$, $h = 16$), while the other has a larger number of skews but a smaller neighborhood ($s = 8$, $h = 8$). Furthermore, each configuration inspects 512 addresses ($\mu = 32$ for $s = 4$, and $\mu = 64$ for $s = 8$) and 1024 addresses ($\mu = 64$ for $s = 4$, and $\mu = 128$ for $s = 8$) during a miss. Both configurations consider $\delta \in \{5, 10, 20\}$. Fig. 9 shows the experimental and theoretical results for the different configurations.

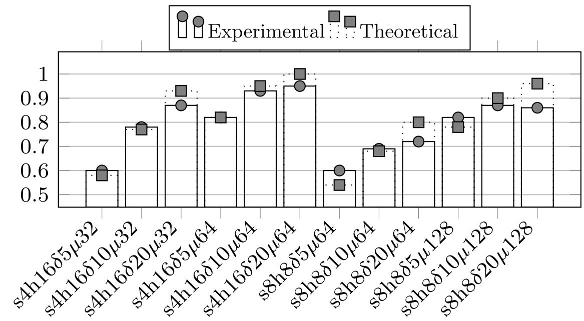

FIGURE 9. Probability of finding a path to the eviction candidate for an 8MiB CoDi$ with different configurations.

TABLE 1. CoDi$ parameters.

Var.	Description
s	Number of skews
N	Number of neighborhoods per skew
h	Neighborhood size
Δ	Vicinity: set interval around the eviction address ($\lvert\Delta\rvert = 2\delta + 1$)
δ	Maximum vertical moves in a vicinity
μ	Maximum inspected neighborhoods during HANDLE_MISS

Experimental results show that, even for the same set of parameters, having a large neighborhood (h) yields a better probability of finding a path. Moreover, the experimental results show how dependent of a large neighborhood the Δ is. Namely, doubling δ from $s4h16\delta10\mu32$ to $s4h16\delta20\mu32$ increases the probability by 12%. The configuration with $h = 8$ has smaller neighborhoods, and so, increasing δ has diminishing returns. Therefore, increasing the neighborhood size (h) has a higher impact in finding a path than increasing the number of skews (s). Indeed, comparing $s8h8\delta20\mu64$ to $s4h16\delta20\mu64$ increases the probability by 32%. This occurs for two reasons: increasing the neighborhood size increases the probability of a vertical move allowing δ to be larger, and increasing the neighborhood size also *decreases* the number of neighborhoods per skew (N). Increasing the number of skews only has the latter result. To increase the probability of finding a path, instead of inspecting a large number of neighborhoods, one can achieve a high probability by using a large neighborhood size and a large δ. If the latency of the external memory allows to inspect more neighborhoods, then the neighborhood size and δ may be reduced. In fact, increasing μ from 32 to 64, in $s4h16\delta20$, provides a 9% percentage improvement.

V. SECURITY DISCUSSION

The design of CoDi$ focuses on mitigating side-channel attacks, by making each cache miss rearrange the overall cache state. This is done by allowing many possible displacement paths from the insertion address to the eviction address, and the actual path is chosen randomly. These paths can move by any address in the cache, thus transforming its overall state. So, CoDi$ requires an attacker to control the addresses

of the entire displacement tree, which spans through the entire cache, to get the required relations through the side-channel.

Since addresses may be displaced between skews and in the same skew, and any address can be evicted from the cache, CoDi$ requires unique attacks. The attacker cannot control which addresses will be considered eviction candidates nor which will be chosen to be evicted. Further, the attacker can not control the path to the eviction address or how many addresses will be displaced in the process. As such, current attacks that reduce the attack complexity of randomized caches [41], [42], [43] are not applicable.

A. ATTACKING CoDi$

The behavior of the CoDi$ miss handling (Algorithm 2) depends on whether a (random) path to a pre-selected eviction address is found. An attacker aims at extracting information from the action of a victim, assessing if it accessed a certain address. To extract this information from CoDi$, the attacker must consider that a miss from the victim triggers a chain of displacements throughout the cache. This random and interlinked displacement of local states one after the other effectively changes the global state of the cache. The effects of this process will be analyzed next, and how they affect the capabilities of an attacker to exploit the cache mechanisms. Specifically, this section analyzes how the attacker must manipulate the local and global state to successfully attack CoDi$. Table 1 summarizes every parameter used to design CoDi$.

1) LOCAL STATE MANIPULATION

An attacker may consider occupying the whole s neighborhoods to where the victim may be (randomly) mapped to with addresses having the same *nID arrangement* (set of the nIDs for each skew of an address). This would force the vertical and horizontal movements to be limited to these attacker controlled neighborhoods, trapping the path search algorithm. Also, this would make these addresses the only candidates for eviction, as they should be the only inspected neighborhoods. However, such an attack requires a physical address space large enough where hs addresses map to the same s nIDs. So, since the cache has N^s nID arrangements, the physical address space must be at least $\log_2(hsN^s)$ bits to map to all these arrangements and occupy the entire neighborhoods. Indeed, for an address space of b bits, and a uniform local state map $H : 2^b \rightarrow N^s$ (representing all hash function of all skews), the probability that there exists a nID arrangement $H(i)$, of *one* address i, which collides with the victim nID arrangement $H(v)$, is

$$\Pr(H(i) = H(v)) = \sum_{i=1}^{2^b} \frac{1}{N^s} = \frac{2^b}{N^s}. \tag{2}$$

Thus, the probability that hs collisions with $H(v)$ *exist* is $\Pr(H(i) = H(v))^{hs}$. For instance, for an 8MiB CoDi$ with $s = 4, h = 16$, the number of local states is 2^{48}. However, current x86_64 implementations only support physical addresses up to 40 bits [44]. Even a 1MiB CoDi$, with the same parameters, requires a physical address space of 42 bits. Hence, for the 8MiB parameters, the probability that enough addresses fill up the neighborhoods of the victim exist is $(\Pr(H(i) = H(v))^{hs} = (2^{40}/2^{48})^{64} = 2^{-512}$. Consequently, relying on different addresses colliding with the victim is infeasible. Note that a birthday-style attack [45] does not help, as a second preimage of the nID arrangement of the victim is required (in opposition to any arbitrary collision).

2) GLOBAL STATE MANIPULATION

Hinging on the rationale of the local state manipulation method, it is intuitive to generalize this technique. Consider capturing the path search mechanism by covering more neighborhoods that do not directly collide with the neighborhood arrangement of the victim. These neighborhoods must be fully occupied by an attacker and should only map to one another, effectively closing a *loop*. This prevents the path searching from reaching the pre-selected eviction address, forcing the eviction of an attacker address instead. Then, observe that this kind of strategy induces a tree structure, with neighborhoods as nodes and (horizontal) movements as branches. In the first level, the attacker must occupy the s possible neighborhoods for the victim with addresses mapping to any nID arrangement. For each subsequent level i, the attacker redirects to $s(s-1)^i$ neighborhoods, but $1/s$ match the nID corresponding to the skew being targeted, thus only $(s-1)^i$ new nIDs are introduced. At a certain point, in level d, the attacker hopes to close the loop. Hence, for each skew, the number of nID possibilities (those of the previous levels) for an attacker to redirect to and close the loop is $C(d) = s+\sum_{i=2}^{d-1}(s-1)^i$. The goal of the attacker is to occupy the neighborhoods until level d (*i.e.*, $s(s-1)^d$ nIDs), such that they all map, through the local map H, to any of its controlled $C(d)$ neighborhoods. Fig. 10 a) shows an example of a CoDi state with nID loops. Every node in the collision graph connects to an attacker controlled nID. Therefore, following any path will inevitably lead to an attacker nID, Fig. 10b). The attacker must sample nID arrangements by querying the cache on random addresses, targeting an nID combination that only includes attacker controlled nIDs for all skews. Successfully sampling an address means getting the right nID for the current skew (1 possibility, as to block vertical movement), and one of the $C(d)$ controlled nIDs for the other skews. The total number of nID arrangements that map to attacker controlled neighborhoods is then $C(d)^{s-1}$. Clearly, the success probability increases with d, as more occupied nIDs are available to the attacker. Here, d is upper-bounded by the total number of neighborhoods, $(s-1)^d < N/2$ (in this model $d \geq 2$, as the case for $d = 1$ was analyzed in the local state manipulation method). Then, the attacker must find $k = h\sum_{i=0}^{d} s(s-1)^i$ addresses, *i.e.*, occupy all neighborhoods until the last level d with h addresses each. Finally, the probability that the attacker can find at least k addresses with the required nIDs is given by the cumulative

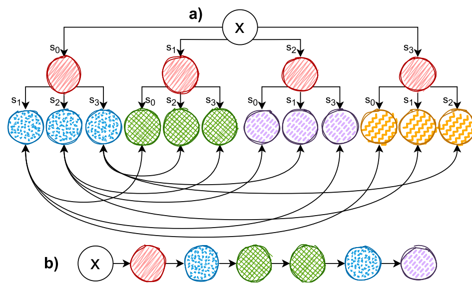

FIGURE 10. Manipulating CoDi$ global state. Colors denote nIDs. a) loops in the collision graph. b) path in tree by unrolling one loop from a).

distribution function of a random variable X that follows the hypergeometric distribution $X \sim H(k, N^s, C(d)^{s-1}, n)$. The hypergeometric distribution is used because at each trial the sampled nID arrangement is not repeated, so the number of available arrangements is reduced. The parameters are: number of successes k, sample space N^s, favorable cases $C(d)^{s-1}$, and number of trials n. Here, n is the number of addresses sampled by the attacker, and must be so that $\Pr(X \geq k)$ is feasible. Then, the security of CoDi$ implies that its parameters must be chosen such that either n or $\Pr(X \geq k)$ are infeasible for $O(2^{-\kappa})$, where κ is the security parameter. The probability that an attacker obtains at least k addresses mapping to its controlled nIDs is

$$\Pr(X \geq k) = 1 - \sum_{i=0}^{k-1} \frac{\binom{C(d)^{s-1}}{i}\binom{N^s - C(d)^{s-1}}{n-i}}{\binom{N^s}{n}}. \tag{3}$$

For the example of an 8MiB CoDi$ with parameters $s = 4$, $h = 16$, for $d = 6$ (the maximum), the number of required iterations is of the order of $n \approx 2^{39}$ for a probability $\Pr(X \geq k) = 2^{-64}$ (for the attacker to always win, the number of required iterations is of the same order, 2^{39}). For $d = 2$ (the minimum), the number of required iterations is of the order of $n \approx 2^{46}$ for a probability $\Pr(X \geq k) = 2^{-64}$.

B. NOISE MODULATION

The previous analysis was unrealistically biased towards the attacker, and thus does not accurately model a real system. In particular, it foregone the entire environment around the system of a real-world application. Indeed, this analysis did not account for other processes also querying the cache, disturbing its overall state and generating *noise*. Since any cache miss can evict an attacker address a, the attacker will not be able to discern which cache miss caused the eviction. *I.e.*, the attacker will receive invalid transmissions through the covert channel but they are indistinguishable from valid transmissions. As a lower-bound on the noise, the attacker is given optimal capabilities, meaning that it can choose any loop at will and program the overall cache state correctly (see Section V-A), prior to letting the victim run. (This is an extremely optimistic scenario for the attacker, as in practice it must always compete for the cache with the other processes in the system.) So, the neighborhood occupation of an attacker in the cache is considered $o = \sum_{i=0}^{d} s(s-1)^d$, for a loop up to level d. As such, the noise should be expressed as a function of d. Letting the attacker choose d means allowing it to occupy as little as it can ($d = 2$), which is the hardest construction, and to occupy the maximum number of neighborhoods allowed (bounded by N), which is the easiest. The noise will be modeled as affecting the cache uniformly, which is again optimistic for an attacker, as its structures (loops) might bias the noise towards removing attacker addresses.

Hereafter, the noise R is modeled as the eviction of any address from the cache, caused by other threads. When there is a cache miss, Algorithm 2 can either find ($F = t$) or fail to find ($F = f$) a path. And, for these two cases, two different distributions model the selection of the eviction address, with the noise being modeled as the expected value over them.

First, if Algorithm 2 finds a path, the eviction address follows the distribution induced by the replacement policy. Here, this distribution will be modeled as a uniform choice over any address in the cache (the attacker occupies o neighborhoods), *i.e.*, $\Pr(R = a|F = t) = o/(sN)$. The assumption of a uniform distribution originates from the fact that there is no way to model every replacement policy, and the eviction candidates are selected from uniformly chosen neighborhoods.

Second, if Algorithm 2 does not find a path, the eviction address is selected uniformly at random from the previously inspected addresses. Then, the distribution depends on how many attacker addresses were inspected in the external memory time frame. For $h\mu$ inspected addresses, the sampling is repeated μ times. So, the probability of evicting an attacker address is the expected value over the number of inspected attacker neighborhoods I,

$$\begin{aligned}\Pr(R = a|F = f) &= \sum_{i=1}^{o} \Pr(R = a|F = f, I = i)\Pr(I = i) \\ &= \sum_{i=1}^{o} \frac{i}{\mu}\binom{\mu}{i}\left(\frac{o}{sN}\right)^i\left(\frac{sN - o}{sN}\right)^{\mu - i}, \end{aligned} \tag{4}$$

where I follows the binomial distribution with parameters $I \sim B(n = \mu, p = o/(sN))$.

Thus, the probability to evict an attacker address a is

$$\begin{aligned}\Pr(\text{noise}) &:= \Pr(R = a) \\ &= \Pr(F = t)\Pr(R = a|F = t) \\ &\quad + \Pr(F = f)\Pr(R = a|F = f) \\ &= \Pr(F = t)\frac{o}{sN} \\ &\quad + (1 - \Pr(F = t))\sum_{i=1}^{o} \frac{i}{\mu}\binom{\mu}{i}\left(\frac{o}{sN}\right)^i\left(\frac{sN - o}{sN}\right)^{\mu - i}. \end{aligned} \tag{5}$$

For the same example of an 8MiB CoDi$ with parameters $s = 4, h = 16$, the two bounds $d = 2$ and $d = 6$ of the global state manipulation method will be instantiated (for $d = 1$ the attack in infeasible as shown in Section V-A). It is conservatively assumed that between the setup of the attacker and the access of the victim, $\nu = 300$ misses occur [46].

Thus, the probability that an attacker loop structure is not affected by the noise is $(1 - P(noise))^{\nu}$. First, for $d = 2$ (the minimum), the attacker requires 2^{46} trials to build its loop, by occupying $o = 52$ neighborhoods, and the probability that this loop is not affected by noise is 0.39. Then, for $d = 6$ (the maximum), the attacker requires 2^{39} trials to build its loop, by occupying $o = 4372$ neighborhoods, and the probability that the attacker loop is not affected by noise is 2^{-134}. Therefore, assuming independent attacker executions, in expectation, it needs 2^{48} trials (its best chance) to build a $d = 2$ loop without being affected by noise, and 2^{174} trials to build a $d = 6$ loop without being affected by noise. Unsurprisingly, exponentially increasing the occupation o by increasing d strongly affects the noise susceptibility of the loop structure.

C. COMPARISON WITH THE STATE-OF-THE-ART

The first randomized cache proposal that bidirectionally connected the global and the local state was MIRAGE [27]. There, a set with invalid ways is able to evict a way from other sets. Therefore, a local state can modify another local state (modifying the global state). However, it can only remove entries from the set, it can not add entries. Moreover, the ability to globally modify the cache hinges on the availability of invalid ways on the set, *i.e.*, the connection between global and local states may disappear. Contrastingly, CoDi$ does not limit modifications to the global state (a modification to a local state can add or remove ways from any other local state), and modifications to the global state are not limited by a specific local state.

Even though all randomized caches share the same base problem (eviction set-creation and setup) for security, the proposals use different security models. Furthermore, some proposals account for noise, while others do not. Hence, Table 2 shows the smallest number of operations required per cache, for an adversary to perform a successful attack. Note that a cryptographic Index Derivation Function (IDF) is a stronger requirement than a Pseudo-Random Number Generator (PRNG) (*e.g.*, implemented with an Linear-Feedback Shift Register (LFSR)), as a cryptographic IDF is implemented from cryptographic hash functions or block ciphers [33] and these primitives cannot be derived directly from PRNGs. Additionally, while CoDi$ security increases for larger caches, this may not be true for ScatterCache [33]. In ScatterCache, the number of required accesses depends only on the number of ways. Thus, having a larger cache requires the same eviction set, but this set is less affected by noise, making the attack easier. Even though MIRAGE [27] achieves a better level of security (for their provided parameters), CoDi$ relaxes the requirement on cryptographic functions of the previous proposals.

VI. EXPERIMENTAL AND COST ANALYSIS

As a test system, the ZSim [47], a PIN-based [48] execution-driven x86 simulator, is used. ZSim models a quad-core system using a cache hierarchy similar to Skylake [49].

Five LLCs are considered in the experiments: *i)* two traditional caches (Full-Associative and Set-Associative); *ii)* two secure state-of-the-art caches (ScatterCache [33] and MIRAGE [27]); *iii)* CoDi$. The baseline is the full-associative design with an access latency of 27 clock cycles. This baseline is optimistic, however, it is used to compare the theoretically best cache against designs with high associativity, *e.g.*, CoDi$ and MIRAGE. The configuration used for CoDi$ follows the guidelines outlined in Section IV-B. MIRAGE uses 2 skews, each with 14 ways [27]. ScatterCache uses 16 skews, each with 1 way [33]. Both ScatterCache and MIRAGE use QARMA [50] as the hash function with a 3 cycle latency, the same as the experiments in [27] and [33]. Therefore, MIRAGE and ScatterCache have a 30 clock cycle access latency. The cache hierarchy is inclusive and the replacement policy used is Least Recently Used (LRU). Table 3 shows the configuration of the simulated system for each LLC used.

For the experiments, 23 benchmarks from SPEC CPU2006 [40] are used with the reference inputs, 10 benchmarks from NPB [51] serial are used with the B problem class, and 30 benchmarks from Polybench [52] are used with the `EXTRALARGE` data sets. Single-threaded experiments execute one benchmark on one core of the system. For multi-threaded experiments, 12 mixes of four randomly selected SPEC, NPB, or Polybench benchmarks were used. Every experiment simulates 10 billion instructions.

A. MISS-RATE ANALYSIS

Table 4 shows the mean Misses per Kilo Instructions (MPKI) for the 63 single-threaded experiments, for the 36 multi-threaded experiments, and for all (specs + NPB + Polybench + mixes) experiments for each considered LLC. For the SPEC benchmarks, the set-associative designs, Set-associative and ScatterCache, are close to one another. ScatterCache increases the miss-rate by 1.2% when compared to the baseline, due to the different skews [33]. The standard set-associative design increases the miss-rate by 0.3%. The high associativity designs, CoDi$ and MIRAGE, have a larger gap between them. CoDi$ has an increase of 0.2% in the miss-rate where MIRAGE has a 5.1% increase. The large penalty in MIRAGE stems from the random selection of the eviction address [27]. In the NPB benchmarks, ScatterCache and CoDi$ increase MPKI by 1%, while MIRAGE increases by 5%. The Polybench benchmarks show approximately the same MPKI across all caches. Analyzing all benchmarks together, ScatterCache and MIRAGE both show an increase in MPKI. ScatterCache increases the MPKI by 1%, and MIRAGE increases by 3%. CoDi$ is the only design that approximates the baseline miss-rates while providing strong security guarantees.

B. PERFORMANCE ANALYSIS

The Instructions per Cycles (IPCs) obtained for SPEC CPU2006, NPB, and Polybench are presented in Fig. 11 for

TABLE 2. Quantitative comparison of CoDi$ *vs.* other cache proposals. Security in number of operations. Noise is assumed in the system according to the models provided, and applied to the eviction set; IDF — Index Derivation Function; PRNG — Pseudo-Random Number Generator.

	Size	Security	Global State	Local State
Full-Associative (no noise)	8MiB	2^{17}	Replacement Policy	—
	16MiB	2^{18}		
**CoDi$** ($s4h16\delta20\mu32$)	8MiB	2^{48}	Universal Hashing + PRNG	Neighborhoods
	16MiB	2^{51}		
ScatterCache [33] (16-ways)	8MiB	2^{47}	Cryptographic IDF	Replacement Policy
	16MiB	2^{40}		
MIRAGE [27] (16+6 ways, no noise)	16MiB	2^{113}	Cryptographic IDF	Invalid ways in set

TABLE 3. ZSim configuration.

Core	x86-64 ISA, 2.3 GHz, Skylake-like OoO [47], [49] 32B-wide ifetch; 2-level bpred with 2k×18-bit BHSRs + 4k×2-bit PHT, 4-wide issue, 36-entry IQ, 224-entry ROB, 72/56-entry LQ/S
L1	32 KiB, 4-way set-associative, split D/I caches, LRU
L2	256 KiB private per-core, 8-way set-associative, LRU, inclusive
LLC	8MiB, LRU, inclusive, 27 cycle latency baseline Configurations: • **Full-Associative**; • **Set-Associative** 16-way; • **ScatterCache** [33] 16-skews (1 way/skew), QARMA64 [50]; • **MIRAGE** [27] 2 skews (14 ways/skew) QARMA64 [50]; • **CoDi$** 4-skews, δ = 20, h = 16 (8 ways/skew), μ = 32, H3 hash [53];
Mem	3 DDR3-1333 channels

each secure LLC state-of-the-art proposal (ScatterCache [33] and MIRAGE [27]), the proposed CoDi$, and a standard set-associative cache. The baseline is a full-associative cache.

Overall CoDi$ comes closest to the performance of the baseline cache due to the low miss-rate and no increase in the hit latency. CoDi$ outperforms the baseline in the benchmarks sphinx3 and soplex. Since the baseline and CoDi$ are high associative designs, this discrepancy occurs because of the replacement policy. A full-associative design selects the LRU address where CoDi$ only selects the LRU address from a pool of randomly selected candidates. In both benchmarks CoDi$ is selecting eviction addresses that may not be the LRUs. Since both benchmarks are not a perfect fit for LRU, CoDi$ is making more suitable evictions. MIRAGE and ScatterCache have worse performance due to the increase in hit latency, being both slower than CoDi$ in 15 SPEC benchmarks. The state-of-the-art caches perform better in the NPB benchmarks. MIRAGE outperforms the baseline in two benchmarks, and ScatterCache matches the baseline. However, CoDi$ still matches or outperforms the state-of-the-art caches. The Polybench benchmarks show the same conclusions as the SPEC benchmarks. CoDi$ outperforms both state-of-the-art caches in 5 benchmarks, and matches them in 17 benchmarks. Inspecting the geometrical mean results further emphasizes this point, CoDi$ outperforms the state-of-the-art caches up to 1%.

C. AREA AND ENERGY ESTIMATES

Since CoDi$ requires addresses to be cheaply displaced, the addresses to access the tag memory must be different from those to access the data memory. Using the same address for tag and data memory would force the controller to displace the data and the tag, which is costly in the forms of energy and resources. As such, CoDi$ can be implemented with a serial lookup architecture. Each tag needs to store the address to the data memory, the current skew nID, and the metadata.

Herein, an example 8MiB CoDi$ with $h = 16$, $s = 4$, $\mu = 32$ and a maximum displacement of 30 addresses is used to estimate the required area and energy. The cache is modeled using one set-associative cache per skew with one read-write port, and three or four read ports. CoDi$ requires multiple read ports for each skew to allow multiple searches to be performed in parallel and to read the whole neighborhood on an access. Each skew is a 2MiB 8-way set-associative cache where each tag entry occupies 108 bits. A Non-Uniform Cache Access model in a 22nm technology node is used. The process of handling a miss requires multiple reads of the tag memory of the different skews. When a path is found to the eviction address, the displacement process will perform one write to the data memory (the newly allocated address). The remaining addresses in the path will only perform writes to the tag memory. The data memory remains unchanged. Therefore, only the tag memory needs the extra read ports, not the data memory. Moreover, regarding the maximum amount of energy per miss CoDi$ would consume, it is assumed that the energy to write to the tag is double of the read energy. So, this max energy per miss considers: the energy required to find the path (reading the tags of all neighborhoods in the path), plus the energy of writing the new address and respective data, plus the energy of moving all nodes in the eviction path until the eviction address.

Table 5 shows the estimates using CACTI 7.0 [54] for multiple 8MiB LLCs: *i)* a single skew and the example CoDi$ with multiple read ports; *ii)* a set-associative (8 ways) cache with a single read-write port; *iii)* a full-associative cache with a single read-write port and one search port; *iv)* a ScatterCache for 1 and 16 skews (1 way/skew); *v)* a MIRAGE cache for 2 skews (14 ways/skew). CoDi$ is similar to a full-associative cache as both allow any address to be evicted. In this instance, CoDi$ reduces the area at least by 45% and the read energy is reduced at least by 79%.

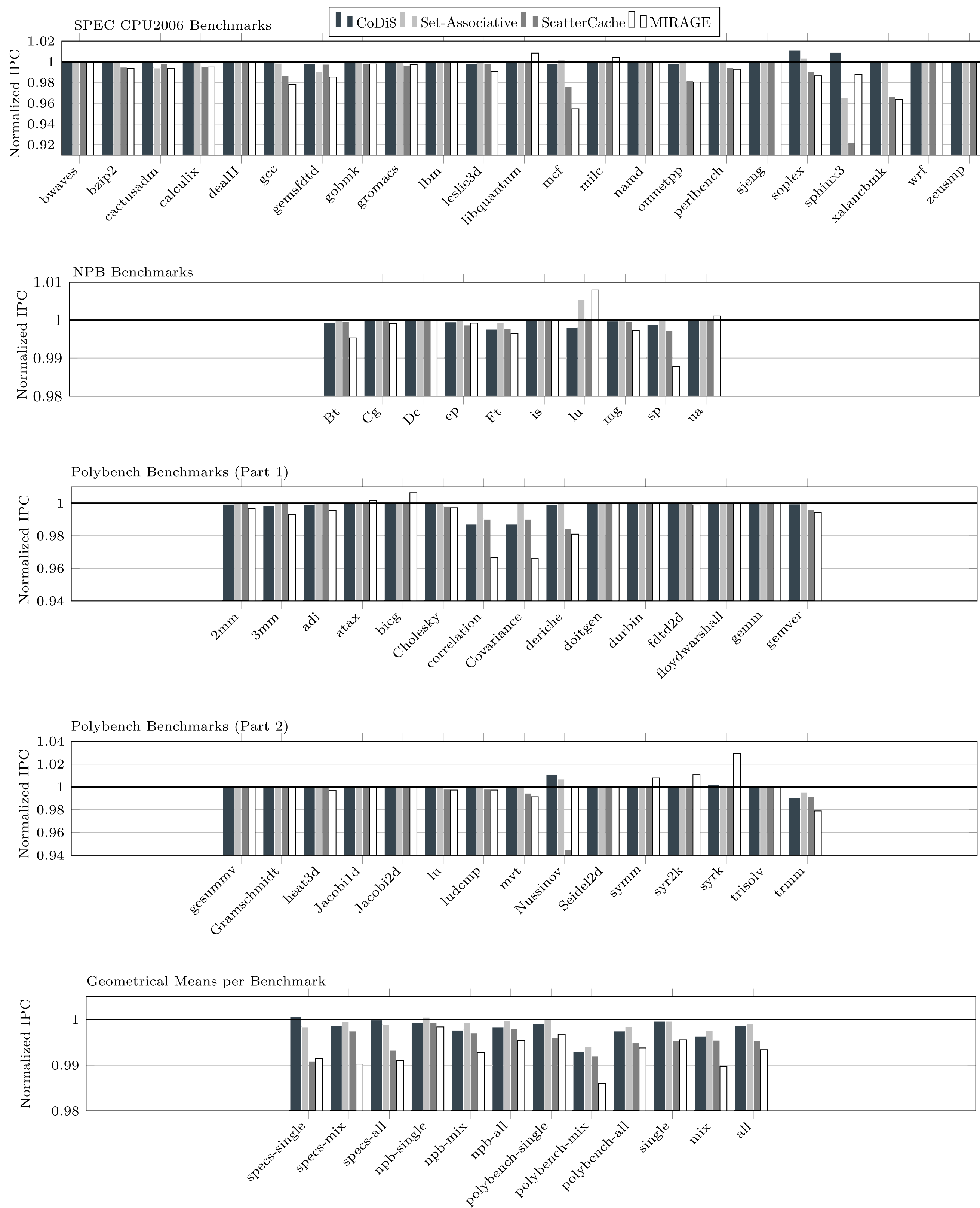

FIGURE 11. Experimental benchmarks for the described system using CoDi$, Set-Associative, ScatterCache, and MIRAGE using a Full-Associative cache as a baseline. All LLCs have a capacity of 8MiB.

TABLE 4. Average MPKI of all SPEC, NPB, and Polybench experiments for each LLC. All means SPEC, NPB, and Polybench benchmarks and mixes.

Experiments	Baseline	Set-Assoc	CoDi$	MIRAGE	Scatter
Specs-Single-23	6.518	6.563	*6.554*	6.900	6.685
Specs-Mixes-12	28.543	28.618	*28.569*	29.907	28.713
Specs-All	14.069	14.124	*14.102*	14.788	14.238
NPB-Single-10	4.342	4.292	*4.338*	4.346	4.302
NPB-Mixes-12	14.713	14.703	*14.810*	15.679	14.894
NPB-All	9.999	9.971	*10.050*	10.528	10.079
Polybench-Single-30	12.484	12.471	*12.483*	12.322	12.499
Polybench-Mixes-12	57.042	57.071	*57.314*	57.527	57.116
Polybench-All	25.215	25.214	*25.292*	25.237	25.247
All-Single	7.781	7.775	*7.792*	7.856	7.828
All-Mixes	33.433	33.464	*33.564*	33.371	33.574
All	16.428	16.436	*16.481*	16.851	16.521

The largest discrepancy between both caches is the max energy per miss. CoDi$ has to perform a path search and then displace the addresses, where a full-associative cache does not. CoDi$ increases the max energy per miss 8.5 times when compared to the set-associative cache, and 3 times when compared to the full-associative cache. However, this is the worst-case scenario for CoDi$. In the majority of cases, the controller will find a path to the eviction address by inspecting fewer neighborhoods and displacing fewer addresses. Experimental results have shown that, on average, the controller inspects 128 nodes and performs 10 displacements. Therefore, on average, CoDi$ increases the energy per miss by a factor of 4 when compared to the set-associative cache, and equals the energy per miss when compared to the full-associative cache. Note that the factors with the largest impact are the writes performed to the tag memory when displacing the addresses due to the extra read ports. Assuming 64 bit addresses, each tag in CoDi$ uses 94 bits: 64 bits for the address to perform a hit check, 15 bits for the data memory pointer (recall that each skew is 2MiB), and 15 bits for the nID. The tag storage overhead for CoDi$ is 214% for the set-associative cache (tag stores 44 bits), and 147% for the full-associative cache (tag stores 64 bits). Despite CoDi$ having a larger tag memory than the full-associative cache, it occupies a smaller area. This happens because the full-associative cache needs to be implemented using Content Addressable Memories (CAMs), which occupy more area due to requiring one comparator per tag. CoDi$ uses a construction of standard set-associative caches with small modifications.

Even though the state-of-the-art secure caches offer different security guarantees and functional properties than CoDi$, they are still included for completeness. ScatterCache and MIRAGE were constructed using the same methodology as CoDi$, one set-associative cache per skew. However, for MIRAGE, CACTI can not build a 14-way cache. Therefore, the 14-way construction was achieved using a group of set-associative caches of 8, 4, and 2 ways. The final MIRAGE cache interprets a skew as the group of all set-associative caches. As expected, for all metrics, both state-of-the-art secure caches ScatterCache and MIRAGE use less energy and are smaller than the proposed CoDi$. Nonetheless, CoDi$ offers better performance with stronger security guarantees without relying on cryptographic IDFs, periodic key refreshes, or cache partitions.

D. IMPLEMENTATION DETAILS

The cache controller has to asynchronously issue requests to the cache to fetch more collisions, store the collisions, and, when a path is found, displace the addresses. Asynchronous operations are already possible in modern caches as they operate under multiple misses through Miss Status Hold Registers (MSHRs) [55]. CoDi$ can reuse MSHRs to handle the extra state. Fig. 12 shows the architecture to handle cache misses. A cache miss allocates a MSHR and schedules all addresses to access the skews selected by the randomly chosen move. Each MSHR stores all previously inspected addresses in a private buffer. When a path is found or a random selection is performed, a Finite State Machine (FSM) issues write commands to each skew to apply the path to the cache.

The previously inspected addresses buffer, for the current CoDi$ configuration, needs to store 512 addresses ($h\mu$) where each is 24 bits. In total each MSHR needs a buffer with 1.5KB. Moreover, the buffer needs to withstand multiple writes simultaneously every time a skew is inspected. The inspected addresses buffer needs a number of write ports equal to the number of addresses returned per skew (the neighborhood size) plus one read port for the displacement FSM. For the current CoDi$ example, the buffer needs 16 write ports and one read port. Therefore, each MSHR needs an extra 0.07mm^2 of area, which is inconsequential in relation to the total. The PRNGs are not accounted for because it is assumed they are implemented using LFSR. Since the LFSR only needs to choose a skew, they will at most use $\log_2(s)$ bits which is even smaller compared to the extra memory needed.

CoDi$ needs to handle cache accesses while either displacing addresses or searching nodes to reach the eviction address. New requests are given priority when entering the cache as hits are handled faster than misses. A path search will only be stalled if the new incoming request and the current neighborhood access the same neighborhood in the same

TABLE 5. CACTI 7.0 [54] area and energy estimates for set-associative, full-associative, ScatterCache (1 way/skew), MIRAGE (14 ways/skew), and CoDi caches with $s = \{1, 4\}$ using $h = 16$. RW = read-write, R = read, S = search.

	Ports	Area (mm^2)			Energy per Bank (nJ)				Max Energy per Miss (nJ)	Avg. Energy per Miss (nJ)
		Tag/Bank	Data/Bank	Total	Tag	Data	Read	Write		
Set-Assoc (8MiB) 32 banks	1RW	0.026	0.572	11.22	0.003	0.070	0.073	0.099	0.172	-
Full-Assoc (8MiB) 32 banks	1RW 1S	-	1.200	38.59	-	-	0.377	0.113	0.490	-
ScatterCache (s=1, 512KiB) 4 banks	1RW	0.016	0.168	0.84	0.003	0.029	0.032	0.043	0.046	-
ScatterCache (s=16, 8MiB) 64 banks	1RW	0.016	0.168	13.42	0.003	0.029	0.032	0.043	0.088	-
MIRAGE (s=1, 8 ways, 2MiB) 4 banks	1RW	0.077	0.572	3.57	0.003	0.070	0.074	0.104	0.107	-
MIRAGE (s=1, 4 ways, 1MiB) 4 banks	1RW	0.033	0.381	2.18	0.003	0.047	0.050	0.065	0.069	-
MIRAGE (s=1, 2 ways, 1MiB) 8 banks	1RW	0.015	0.168	1.68	0.003	0.029	0.032	0.046	0.049	-
MIRAGE (s=2, 8MiB) 32 banks	1RW	0.126	1.121	14.86	0.009	0.070	0.155	0.104	0.123	-
CoDi$ (s=1, 2MiB) 8 banks	1RW 3R	0.435	0.571	4.71	0.008	0.070	0.078	0.115	1.091	0.387
	1RW 4R	0.582	0.571	5.4	0.011	0.070	0.081	0.115	1.457	0.489
CoDi$ (s=4, 8MiB) 32 banks	1RW 12R	0.435	0.571	18.85	0.008	0.070	0.078	0.115	1.091	0.387
	1RW 16R	0.582	0.571	21.6	0.011	0.070	0.081	0.115	1.457	0.489

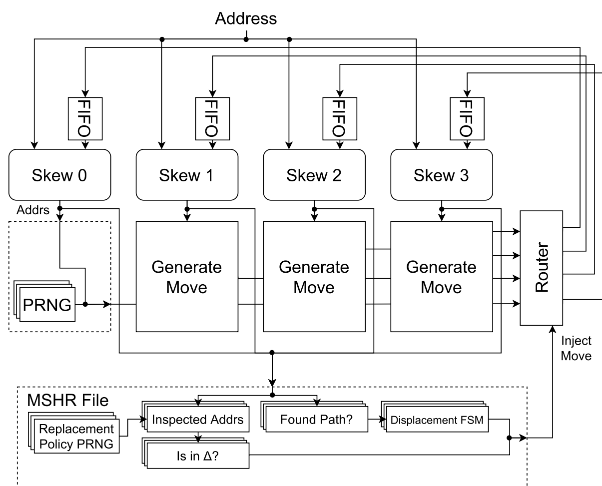

FIGURE 12. CoDi$ miss handling architecture.

skew, or if two neighborhood accesses during a search are to the same nID. Otherwise, all accesses proceed in parallel. The router knows which accesses are from a miss or from a new request, and will attribute priority accordingly.

It may seem that by delaying accesses to neighborhoods during a path search may cause the search to not complete before the arrival of the external memory response. However, the analysis in Section IV shows otherwise. It conservatively assumes a 100 clock cycle latency of the external memory response, while modern micro-architectures use up to 250 clock cycles [56]. Therefore, delaying some path searches is not likely to reduce the probability of finding a path.

VII. RELATED WORK

State-of-the-art proposals on designing secure LLCs revolve around hardening the problem of changing the state of specific cached data, which results in a latency discrepancy. The state of the cached data can be changed through an eviction (set pressure [4], replacement policy [24]) or an invalidation [18], [57]. There are two state-of-the-art concepts for secure cache proposals: partitioned caches and randomized caches.

A. PARTITIONED CACHES

Partitioned caches gate access to parts of the cache when multiple processes are running, which results in a fight

for resources. Depending on which process managed to capture the largest partition of the cache, it may or may not be making the most out of the partition size. Other processes that need a larger partition are unable to grow, which may result in a performance bottleneck [26]. To avoid this, a common technique is to split the running processes into two domains: secure and insecure. Secure domains have access to a specialized portion of the cache with a special rule set, either a strictly reserved portion [28], [29], [30], [58], [59], [60] or a partial full-associative design [26]. Insecure domains have access to the whole or a portion of the cache with a secure domain aware rule set [26], [28]. Other proposals perform the process partitioning at the software level, where the operating system is able to reserve portions of the LLC to a particular process [61], [62]. This system can be used to thwart timing side-channel attacks [63], [64], [65]. CoDi$ does not use a partitioned-base defense (although shared pages are duplicated) and there is no performance penalty due to partition fighting.

B. RANDOMIZED CACHES

Current randomized cache proposals rely on the security properties of cryptographic primitives. Due to the latency requirements of the LLC, low-latency hash functions are used [50], [66]. The simplest defenses use a hash function with a unique ID [33], [67]. Other proposals choose not to use a unique ID but will change the key of a block-cipher periodically [31], [32]. Indeed, having full knowledge of collisions of the hash function is sufficient to break its security [27], [33], [43]. CoDi$ is a randomized cache that does not rely on a cryptographic hash function for security, thus the hit latency is not increased. The security of CoDi$ relies only on the attacker being unable to control the overall state of the cache, *i.e.*, to extract information the attacker would need to discern which cache miss caused which eviction.

VIII. CONCLUSION

The design of CoDi$ shows that by tying the local state of an address to the global state of the cache, one can build a timing side-channel secure cache. This is achieved by allowing addresses to be displaced between skews and in the same skew, and through the global selection of an eviction address. Unlike other randomized cache proposals, CoDi$ does not rely on the properties of cryptographic primitives, periodic key refreshes, or cache partitions to provide security. Since any address in the cache can be evicted through any path, *a successful attack requires the attacker to have full control over all paths to every eviction candidate in the cache*. Otherwise, the attacker will not be able to discern which cache miss caused which eviction. Moreover, the attacker cannot detect if an eviction was caused by noise or by a valid transmission. Due to this uncertainty, the cache state setup by the attacker does not enable it to know the origin of extracted information, as the attacker cache state may even no longer exist. Since the security model no longer depends solely on hash collisions, current state-of-the-art attacks for randomized caches do not work.

Area and energy estimates show that CoDi$, at worst, uses 45% less area, 79% less energy per hit, and 28% less energy per miss when compared to a full-associative cache. Experimental analysis shows that the miss-rate of CoDi$ increases by 0.2% and the IPC is similar to a full-associative cache. Furthermore, the results show a reduction of up to 5% in MPKI and an improvement of up to 1% in IPC, when compared to two state-of-the-art randomized cache proposals.

ACKNOWLEDGMENT

The authors would like to thank Diogo Marques and João Vieira for reviewing multiple drafts of this article.

REFERENCES

[1] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2019, pp. 1–19. [Online]. Available: https://spectreattack.com/spectre.pdf

[2] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading kernel memory from user space," in *Proc. 27th USENIX Secur. Symp. (USENIX Security)*, 2018, pp. 973–990. [Online]. Available: https://meltdownattack.com/meltdown.pdf

[3] T. Hornby. (2016). *Side-Channel Attacks on Everyday Applications: Distinguishing Inputs With Flush+Reload*. [Online]. Available: https://www.blackhat.com/docs/us-16/materials/us-16-Hornby-Side-Channel-Attacks-On-Everyday-Applications-wp.pdf

[4] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *Proc. IEEE Symp. Secur. Privacy*, May 2015, pp. 605–622. [Online]. Available: http://palms.ee.princeton.edu/system/files/SP_vfinal.pdf

[5] J. V. Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution," in *Proc. 27th USENIX Secur. Symp.* Berkeley, CA, USA: USENIX Association, Aug. 2018, pp. 1–19.

[6] J. Van Bulck, D. Moghimi, M. Schwarz, M. Lippi, M. Minkin, D. Genkin, Y. Yarom, B. Sunar, D. Gruss, and F. Piessens, "LVI: Hijacking transient execution through microarchitectural load value injection," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2020, pp. 54–72.

[7] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss, "ZombieLoad: Cross-privilege-boundary data sampling," in *Proc. CCS*, Nov. 2019, pp. 753–768.

[8] D. Gruss, C. Maurice, K. Wagner, and S. Mangard, "Flush+flush: A fast and stealthy cache attack," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, J. Caballero, U. Zurutuza, and R. J. Rodríguez, Eds. Cham, Switzerland: Springer, 2016, pp. 279–299. [Online]. Available: https://gruss.cc/files/flushflush.pdf

[9] A. Purnal, F. Turan, and I. Verbauwhede, "Prime+scope: Overcoming the observer effect for high-precision cache contention attacks," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Nov. 2021, pp. 2906–2920.

[10] E. M. Koruyeh, N. Khaled Khasawneh, C. Song, and N. Abu-Ghazaleh, "Spectre returns! speculation attacks using the return stack buffer," in *Proc. 12th USENIX Workshop Offensive Technol. (WOOT)*. Baltimore, MD, USA: USENIX Association, Aug. 2018, pp. 1–12.

[11] F. Liu, H. Wu, K. Mai, and R. B. Lee, "Newcache: Secure cache architecture thwarting cache side-channel attacks," *IEEE Micro*, vol. 36, no. 5, pp. 8–16, Sep./Oct. 2016. [Online]. Available: http://palms.ee.princeton.edu/system/files/07723806.pdf

[12] X. Zhang, Z. Yuan, R. Chang, and Y. Zhou, "Seeds of SEED: H2Cache: Building a hybrid randomized cache hierarchy for mitigating cache side-channel attacks," in *Proc. Int. Symp. Secure Private Execution Environ. Design (SEED)*, Sep. 2021, pp. 29–36.

[13] D. Brumley and D. Boneh, "Remote timing attacks are practical," *Comput. Netw.*, vol. 48, no. 5, pp. 701–716, 2005.

[14] B. B. Brumley and N. Tuveri, "Remote timing attacks are still practical," in *Computer Security—ESORICS 2011*, V. Atluri and C. Diaz, Eds. Berlin, Germany: Springer, 2011, pp. 355–371. [Online]. Available: https://eprint.iacr.org/2011/232.pdf

[15] G. Irazoqui, T. Eisenbarth, and B. Sunar, "S$A: A shared cache attack that works across cores and defies VM sandboxing—And its application to AES," in *Proc. IEEE Symp. Secur. Privacy*, May 2015, pp. 591–604. [Online]. Available: https://www.ieee-security.org/TC/SP2015/papers-archived/6949a591.pdf

[16] M. Neve and J.-P. Seifert, "Advances on access-driven cache attacks on AES," in *Selected Areas in Cryptography*, E. Biham and A. M. Youssef, Eds. Berlin, Germany: Springer, 2007, pp. 147–162.

[17] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: The case of AES," in *Topics in Cryptology—CT-RSA 2006*, D. Pointcheval, Ed. Berlin, Germany: Springer, 2006, pp. 1–20. [Online]. Available: https://www.cs.tau.ac.il/~tromer/papers/cache.pdf

[18] Y. Yarom and K. Falkner, "FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack," in *Proc. 23rd USENIX Secur. Symp. (USENIX Security)*. San Diego, CA, USA: USENIX Association, Aug. 2014, pp. 719–732. [Online]. Available: https://www.usenix.org/system/files/conference/usenixsecurity14/sec14-paper-yarom.pdf

[19] D. Gruss, C. Maurice, A. Fogh, M. Lipp, and S. Mangard, "Prefetch side-channel attacks: Bypassing SMAP and kernel ASLR," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.* New York, NY, USA: Association for Computing Machinery, Oct. 2016, pp. 368–379.

[20] E. Tromer, D. A. Osvik, and A. Shamir, "Efficient cache attacks on AES, and countermeasures," *J. Cryptol.*, vol. 23, no. 1, pp. 37–71, Jan. 2010. [Online]. Available: https://link.springer.com/content/pdf/10.1007%2Fs00145-009-9049-y.pdf

[21] O. Acıiçmez and W. Schindler, "A vulnerability in RSA implementations due to instruction cache analysis and its demonstration on OpenSSL," in *Topics in Cryptology—CT-RSA 2008*, T. Malkin, Ed. Berlin, Germany: Springer, 2008, pp. 256–273.

[22] C. Chen, T. Wang, Y. Kou, X. Chen, and X. Li, "Improvement of trace-driven I-Cache timing attack on the RSA algorithm," *J. Syst. Softw.*, vol. 86, no. 1, pp. 100–107, 2013.

[23] M. Yan, B. Gopireddy, T. Shull, and J. Torrellas, "Secure hierarchy-aware cache replacement policy (SHARP): Defending against cache-based side channel atacks," in *Proc. 44th Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2017, pp. 347–360. [Online]. Available: http://iacoma.cs.uiuc.edu/iacoma-papers/isca17_2.pdf

[24] S. Briongos, P. Malagon, J. M. Moya, and T. Eisenbarth, "RELOAD+REFRESH: Abusing cache replacement policies to perform stealthy cache attacks," in *Proc. 29th USENIX Secur. Symp. (USENIX Security)*. Boston, MA, USA: USENIX Association, Aug. 2020, pp. 1967–1984. [Online]. Available: https://www.usenix.org/system/files/sec20-briongos_0.pdf

[25] W. Xiong and J. Szefer, "Leaking information through cache LRU states," *CoRR*, vol. abs/1905.08348, pp. 1–13, May 2019.

[26] G. Dessouky, T. Frassetto, and A.-R. Sadeghi, "HYBCACHE: Hybrid side-channel-resilient caches for trusted execution environments," in *Proc. 29th USENIX Secur. Symp. (USENIX Security)*. Boston, MA, USA: USENIX Association, Aug. 2020, pp. 451–468.

[27] G. Saileshwar and M. Qureshi, "MIRAGE: Mitigating conflict-based cache attacks with a practical fully-associative design," in *Proc. 30th USENIX Secur. Symp. (USENIX Security)*. Berkeley, CA, USA: USENIX Association, Aug. 2021, pp. 1379–1396.

[28] L. Domnitser, A. Jaleel, J. Loew, N. Abu-Ghazaleh, and D. Ponomarev, "Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks," *ACM Trans. Archit. Code Optim.*, vol. 8, no. 4, pp. 1–21, Jan. 2012. [Online]. Available: http://palms.princeton.edu/system/files/hipeac12.pdf

[29] G. Dessouky, E. Stapf, P. Mahmoody, A. Gruler, and A.-R. Sadeghi, "Chunked-cache: On-demand and scalable cache isolation for security architectures," in *Proc. Netw. Distrib. Syst. Secur. Symp. (NDSS)*, 2022.

[30] Y. Wang, A. Ferraiuolo, D. Zhang, A. C. Myers, and G. E. Suh, "SecDCP: Secure dynamic cache partitioning for efficient timing channel protection," in *Proc. 53rd Annu. Design Autom. Conf.* New York, NY, USA: Association for Computing Machinery, Jun. 2016. [Online]. Available: http://www.cs.cornell.edu/andru/papers/dac16/dac16.pdf

[31] M. K. Qureshi, "CEASER: Mitigating conflict-based cache attacks via encrypted-address and remapping," in *Proc. 51st Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Oct. 2018, pp. 775–787. [Online]. Available: http://memlab.ece.gatech.edu/papers/MICRO_2018_2.pdf

[32] M. K. Qureshi, "New attacks and defense for encrypted-address cache," in *Proc. 46th Int. Symp. Comput. Archit.* New York, NY, USA: Association for Computing Machinery, Jun. 2019, pp. 360–371. [Online]. Available: http://memlab.ece.gatech.edu/papers/ISCA_2019_1.pdf

[33] M. Werner, T. Unterluggauer, L. Giner, M. Schwarz, D. Gruss, and S. Mangard, "ScatterCache: Thwarting cache attacks via cache set randomization," in *Proc. 28th USENIX Secur. Symp. (USENIX Security)*. Santa Clara, CA, USA: USENIX Association, Aug. 2019, pp. 675–692.

[34] A. Seznec, "A case for two-way skewed-associative caches," in *Proc. 20th Annu. Int. Symp. Comput. Archit. (ISCA)*. New York, NY, USA: Association for Computing Machinery, 1993, pp. 169–178. [Online]. Available: https://dl.acm.org/doi/pdf/10.1145/165123.165152

[35] J. L. Carter and M. N. Wegman, "Universal classes of hash functions," *J. Comput. Syst. Sci.*, vol. 18, no. 2, pp. 143–154, Apr. 1979.

[36] C. E. Shannon, "A mathematical theory of cryptography," in *Mathematical Theory of Cryptography*. Murray Hill, NJ, USA: Bell Telephone Labs, 1945.

[37] R. Pagh and F. F. Rodler, "Cuckoo hashing," in *Algorithms—ESA 2001*, F. M. A. D. Heide, Ed. Berlin, Germany: Springer, 2001, pp. 121–133.

[38] M. Herlihy, N. Shavit, and M. Tzafrir, "Hopscotch hashing," in *Proc. 22nd Int. Symp. Distrib. Comput. (DISC)*. Berlin, Germany: Springer-Verlag, 2008, pp. 350–364.

[39] D. Sanchez and C. Kozyrakis, "The ZCache: Decoupling ways and associativity," in *Proc. 43rd Annu. IEEE/ACM Int. Symp. Microarchitecture*. Washington, DC, USA: IEEE Computer Society, Dec. 2010, pp. 187–198. [Online]. Available: https://people.csail.mit.edu/sanchez/papers/2010.zcache.micro.pdf

[40] J. L. Henning, "SPEC CPU2006 benchmark descriptions," *ACM SIGARCH Comput. Archit. News*, vol. 34, no. 4, pp. 1–17, Sep. 2006.

[41] A. Purnal, L. Giner, D. Gruß, and I. Verbauwhede, "Systematic analysis of randomization-based protected cache architectures," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2021, pp. 987–1002.

[42] A. Purnal and I. Verbauwhede, "Advanced profiling for probabilistic prime+probe attacks and covert channels in scattercache," *CoRR*, vol. abs/1908.03383, pp. 1–8, Aug. 2019.

[43] W. Song, B. Li, Z. Xue, Z. Li, W. Wang, and P. Liu, "Randomized last-level caches are still vulnerable to cache side-channel attacks! But we can fix it," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2021, pp. 955–969.

[44] *BIOS and Kernel Developer's Guide (BKDG) for AMD Family 15h Models 70h-7Fh Processors*, AMD, Santa Clara, CA, USA, 2018.

[45] I. Mironov et al., "Hash functions: Theory, attacks, and applications," Microsoft Res., Silicon Valley Campus, Bengaluru, India, Tech. Rep. 187, 2005, pp. 1–22.

[46] G. Saileshwar, B. Wang, M. Qureshi, and P. J. Nair, "Randomized row-swap: Mitigating row hammer by breaking spatial correlation between aggressor and victim rows," in *Proc. 27th ACM Int. Conf. Architectural Support Program. Lang. Operating Syst.* New York, NY, USA: Association for Computing Machinery, 2022, pp. 1056–1069.

[47] D. Sanchez and C. Kozyrakis, "ZSim: Fast and accurate microarchitectural simulation of thousand-core systems," in *Proc. 40th Annu. Int. Symp. Comput. Archit.* New York, NY, USA: Association for Computing Machinery, Jun. 2013, pp. 475–486. [Online]. Available: http://people.csail.mit.edu/sanchez/papers/2013.zsim.isca.pdf

[48] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proc. ACM SIGPLAN Conf. Program. Lang. Design Implement.* New York, NY, USA: Association for Computing Machinery, Jun. 2005, pp. 190–200.

[49] J. Doweck, W.-F. Kao, A. K.-Y. Lu, J. Mandelblat, A. Rahatekar, L. Rappoport, E. Rotem, A. Yasin, and A. Yoaz, "Inside 6th-generation Intel core: New microarchitecture code-named skylake," *IEEE Micro*, vol. 37, no. 2, pp. 52–62, Mar. 2017.

[50] R. Avanzi, "The QARMA block cipher family. Almost MDS matrices over rings with zero divisors, nearly symmetric even-mansour constructions with non-involutory central rounds, and search heuristics for low-latency s-boxes," *IACR Trans. Symmetric Cryptol.*, vol. 2017, no. 1, pp. 4–44, Mar. 2017.

[51] *NAS Parallel Benchmarks*, NASA Adv. Supercomput. Division, Mountain View, CA, USA, 2020.

[52] L.-N. Pouchet, U. Bondugula, and T. Yuki, "PolyBench/C: The polyhedral benchmark suite," Tech. Rep.

[53] J. L. Carter and M. N. Wegman, "Universal classes of hash functions (extended abstract)," in *Proc. 9th Annu. ACM Symp. Theory Comput. (STOC)*. New York, NY, USA: Association for Computing Machinery, 1977, pp. 106–112.

[54] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "Cacti 6.0: A tool to model large caches," *HP Laboratories*, vol. 27, p. 28, Apr. 2009.

[55] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. Amsterdam, The Netherlands: Elsevier, 2011.

[56] M. Yan, R. Sprabery, B. Gopireddy, C. Fletcher, R. Campbell, and J. Torrellas, "Attack directories, not caches: Side channel attacks in a non-inclusive world," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2019, pp. 888–904. [Online]. Available: http://iacoma.cs.uiuc.edu/iacoma-papers/ssp19.pdf

[57] G. Irazoqui, T. Eisenbarth, and B. Sunar, "Cross processor cache attacks," in *Proc. 11th ACM Asia Conf. Comput. Commun. Secur.* New York, NY, USA: Association for Computing Machinery, May 2016, pp. 353–364. [Online]. Available: https://eprint.iacr.org/2015/1155.pdf

[58] V. Kiriansky, I. Lebedev, S. Amarasinghe, S. Devadas, and J. Emer, "DAWG: A defense against cache timing attacks in speculative execution processors," in *Proc. 51st Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Oct. 2018, pp. 974–987. [Online]. Available: https://eprint.iacr.org/2018/418.pdf

[59] K. T. Sundararajan, V. Porpodas, T. M. Jones, N. P. Topham, and B. Franke, "Cooperative partitioning: Energy-efficient cache partitioning for high-performance CMPs," in *Proc. IEEE Int. Symp. High-Perform. Comp Archit.*, Feb. 2012, pp. 1–12. [Online]. Available: https://www.cl.cam.ac.uk/~tmj32/papers/docs/sundararajan12-hpca.pdf

[60] G. Saileshwar, S. Kariyappa, and M. Qureshi, "Bespoke cache enclaves: Fine-grained and scalable isolation from cache side-channels via flexible set-partitioning," in *Proc. Int. Symp. Secure Private Execution Environ. Design (SEED)*, Sep. 2021, pp. 37–49.

[61] *AMD64 Technology Platform Quality of Service Extensions*, AMD, Santa Clara, CA, USA, 2018.

[62] A. Herdrich, E. Verplanke, P. Autee, R. Illikkal, C. Gianos, R. Singhal, and R. Iyer, "Cache QoS: From concept to reality in the Intel Xeon processor E5-2600 v3 product family," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, Mar. 2016, pp. 657–668.

[63] F. Liu, Q. Ge, Y. Yarom, F. Mckeen, C. Rozas, G. Heiser, and R. B. Lee, "CATalyst: Defeating last-level cache side channel attacks in cloud computing," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, Mar. 2016, pp. 406–418. [Online]. Available: http://palms.ee.princeton.edu/system/files/CATalyst_vfinal_correct.pdf

[64] T. Kim, M. Peinado, and G. Mainar-Ruiz, "STEALTHMEM: System-level protection against cache-based side channel attacks in the cloud," in *Proc. 21st USENIX Secur. Symp. (USENIX Security 12)*. Bellevue, WA, USA: USENIX Association, Aug. 2012, pp. 189–204. [Online]. Available: https://www.usenix.org/system/files/conference/usenixsecurity12/sec12-final79.pdf

[65] Z. Zhou, M. K. Reiter, and Y. Zhang, "A software approach to defeating side channels in last-level caches," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.* New York, NY, USA: Association for Computing Machinery, Oct. 2016, pp. 871–882. [Online]. Available: https://www.cs.unc.edu/ ziqiao/papers/ccs2016.pdf

[66] J. Borghoff, A. Canteaut, T. Güneysu, E. B. Kavun, M. Knezevic, L. R. Knudsen, G. Leander, V. Nikov, C. Paar, C. Rechberger, P. Rombouts, S. S. Thomsen, and T. Yalçın, "PRINCE—A low-latency block cipher for pervasive computing applications," in *Advances in Cryptology—ASIACRYPT 2012*, X. Wang and K. Sako, Eds. Berlin, Germany: Springer, 2012, pp. 208–225.

[67] S. Constable and T. Unterluggauer, "Seeds of SEED: A side-channel resilient cache skewed by a linear function over a Galois field," in *Proc. IEEE Int. Symp. Secure Private Execution Environ. Design (SEED)*, Sep. 2021, pp. 14–21.

[68] O. Weisse, J. V. Bulck, M. Minkin, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, R. Strackx, F. Thomas Wenisch, and Y. Yarom, "Foreshadow-NG: Breaking the virtual memory abstraction with transient out-of-order execution," Tech. Rep., 2018. [Online]. Available: https://foreshadowattack.eu/

LUÍS FIOLHAIS received the B.Sc. and M.Sc. degrees in electrical and computer engineering from Instituto Superior Técnico, Universidade de Lisboa, Portugal, in 2017. He is currently pursuing the Ph.D. degree with INESC-ID researching micro-architectural attacks and defenses. His research interests include computer architectures, VLSI, and computer security.

MANUEL GOULÃO received the Ph.D. degree in information security from Instituto Superior Técnico, Universidade de Lisboa, Portugal, in 2022. He is with the Security and Quantum Information Group, Instituto de Telecomunicações, Lisbon. His research interests include theoretical and applied cryptography and computer security.

LEONEL SOUSA (Senior Member, IEEE) received the Ph.D. degree in electrical and computer engineering from Instituto Superior Tecnico (IST), Universidade de Lisboa (UL), Lisbon, Portugal, in 1996. He is currently a Full Professor with UL and a Senior Researcher with the Research and Development Instituto de Engenharia de Sistemas e Computadores (INESC-ID). He has authored or coauthored more than 250 papers in journals and international conferences, and has edited four special issues of international journals. His research interests include VLSI architectures, computer architectures and arithmetic, parallel computing, and signal processing systems. He is a fellow of IET and a Distinguished Scientist of ACM.