IR-ORAM: Path Access Type Based Memory Intensity Reduction for Path-ORAM

Mehrnoosh Raoufi
Computer Science Department
University of Pittsburgh
Pittsburgh, USA
mraoufi@cs.pitt.edu

Youtao Zhang
Computer Science Department
University of Pittsburgh
Pittsburgh, USA
zhangyt@cs.pitt.edu

Jun Yang
Electrical and Computer Engineering Department
University of Pittsburgh
Pittsburgh, USA
juy9@pitt.edu

***Abstract*—Path ORAM is an effective ORAM (Oblivious RAM) primitive for protecting memory access patterns. Path ORAM converts each off-chip memory request from user program to tens to hundreds of memory accesses. While several schemes have been proposed to mitigate the total number of memory accesses, Path ORAM remains a highly memory intensive primitive that leads to large memory bandwidth occupation and performance degradation.**

In this paper, we propose IR-ORAM to reduce the memory intensity based on path access types in Path ORAM. Path accesses in Path ORAM, while being kept oblivious to ensure privacy protection, can be categorized to three types: paths for requested data blocks, paths for position map blocks, and dummy paths. We develop a set of techniques to reduce the memory intensity of each type while ensuring the obliviousness at the same time — we reduce the number of data blocks to access for each tree path, reduce the number of path accesses for position maps, and convert many dummy path accesses to early write-backs of dirty data in LLC. Our experimental results show that IR-ORAM achieves on average 42% performance improvement over the state-of-the-art while effectively enforcing the memory access obliviousness and the same level of security protection.

***Keywords*—memory systems; oblivious RAM; ORAM**

I. Introduction

Cloud computing has become a widely adopted computing paradigm for modern applications. While cloud computing can greatly improve computing flexibility and scalability, and reduce maintenance cost, there is a growing concern of its security and user privacy [7], [13], [15], [17], [19], [35]. Hardware-assisted security schemes, e.g., XOM [31] and Intel SGX [14], have been developed to effectively protect data secrecy and integrity for such computing environments. However, it remains a big challenge to hide data access patterns, in particular, for widely adopted direct-attached memory architecture in which memory addresses are communicated off-chip in cleartext. Leaking access patterns may reveal important user privacy and result in severe consequences [40]. Studies have shown that protecting user privacy at the highest level demands ORAM (oblivious RAM) [11], [12], an expensive crypto primitive that prevents leaking data access patterns from obfuscating memory accesses.

Path ORAM [27] is a popular ORAM implementation that organizes the user data to be protected in a tree structure and converts each user memory request to one or multiple tree path accesses. While Path ORAM reduces the memory access overhead from O(N) in traditional ORAM [11], [12] to O(logN) (where N is the number of data blocks of the protected memory space), it remains a highly memory intensive primitive that consists of path accesses of three types.

- **PT$_p$ path**. To determine the tree path to access, Path ORAM uses multiple levels of position map, a.k.a., PosMap, tables to map user addresses to path IDs. While an on-chip buffer can cache frequently used entries, Path ORAM still needs to generate many PosMap accesses.
- **PT$_d$ path**. To access a requested data block after knowing its path ID, Path ORAM accesses all tree nodes on the path from the leaf node to the tree root. All data blocks in these nodes are accessed to ensure secure protection.
- **PT$_m$ path**. To prevent timing channel attacks, Path ORAM needs to generate path accesses at a fixed rate, e.g., one path access per T cycles [9]. When it needs to generate a path access but there is no pending real request, Path ORAM constructs a dummy one to access a random tree path.

Since the high memory intensity has become the main obstacle that prevents Path ORAM from wide deployment, many schemes have been proposed to mitigate memory bandwidth usage and its impact. Maas *et al.* proposed to cache top tree levels on-chip to reduce the number of data blocks to access [22]. Nagarajan *et al.* proposed to create a smaller tree such that majority accesses can be satisfied by the smaller tree, which reduces the length of the tree and the number of blocks per node [23]. Zhang *et al.* proposed to exploit the dummy blocks of the same path to save shadow copies so that the processor can resume execution early [38]. Unfortunately, the memory intensity of Path ORAM remains high, which still incurs large performance degradation to the user applications.

In this paper, we propose IR-ORAM that proactively reduces the memory access intensity of each path type. **IR-ORAM consists of a set of three path-type-dependent schemes with a focus on intensity reduction, i.e., it reduces the number of each type of path accesses to improve the overall performance.** Our contributions are as follows.

- We propose *IR-Alloc*, a utilization-aware node size allocation strategy, to reduce the number of data blocks to access

for each path, i.e., the intensity of all types of paths. *IR-Alloc* exploits the observation that tree nodes at different levels exhibit significant space utilization difference. Here, the *space utilization* is defined as the portion of memory blocks that saves real data blocks (rather than dummy blocks). In particular, the middle level nodes show low utilization such that we can reduce the number of blocks allocated to these tree nodes, which effectively reduces the number of data blocks in each path while incurring minimized impacts on memory space and ORAM operation.

- We propose *IR-Stash* to reduce the number of PosMap accesses, i.e., the intensity of PT_p paths. Our utilization study reveals that the tree top mostly serves as an overflow buffer of the on-chip stash. However, existing schemes on buffering the tree top on-chip lead to either large space overhead or unnecessary PosMap accesses. We therefore develop *IR-Stash* that places top tree levels in a set-associative sub-stash and maintains its tree structure using a small table. *IR-Stash* helps to eliminate unnecessary PosMap accesses at low overhead.
- We propose *IR-DWB* to reduce the number of dummy path accesses, i.e., the intensity of PT_m paths. *IR-DWB* converts dummy paths to write-back operations of dirty LRU (least-recently-used) entries in the LLC (last level cache), which minimizes LLC replacement overhead while introducing no memory contention as that in traditional eager writeback cache designs.
- We evaluate the proposed techniques and study their effectiveness in memory intensity reduction. Our experimental results show that IR-ORAM achieves on average 42% performance improvement over the state-of-the-art while effectively enforcing the original memory access obliviousness and thus the same level of security protection.

In the rest of the paper, we briefly discuss the background in Section II. Section III describes the motivation. We elaborate the design details in Section IV. Section V and VI discuss the experimental methodology and the experiment, respectively. Section VII discusses additional related work. Section VIII concludes the paper.

II. Background

In this section, we first present the threat model and then briefly discuss Path ORAM and its optimizations that are related to our design.

A. Threat Model

In this paper, we adopt the same threat model as Freecursive ORAM [8] and other existing ORAM studies [23], [25], [27], [32], [38]. We focus on preventing information leakage from the memory access traces of applications running on *not-fully-trustworthy* cloud servers. In such environments, the only trusted component (i.e., trusted computing base (TCB)) is the processor, i.e., while the processor can faithfully execute the user application, all other components, including the OS, the memory modules, and the memory buses, are not trustworthy. To ensure high level data security, we need to protect data secrecy, data integrity, and data privacy for the secure applications running on the servers.

We assume that data secrecy and data integrity are protected with hardware-assisted security enhancements [10], [14], [29], [31], [36]. As an example, the recently released Intel SGX architecture saves encrypted user code and data in memory. They are decrypted when being brought into the processor [14]. A Merkle tree is built on the user data to prevent unauthorized changes [10]. Existing studies showed that the performance impacts of these enhancements are effectively mitigated with the trustworthy crypto hardware integrated in the processor.

In this paper, we assume the attackers have the physical access to the servers so that they can trace and analyze all the memory accesses. The servers adopt the direct-attached memory architecture [16] such that, while the data on data buses are transmitted in ciphertext, the data on address buses and command buses are in cleartext. To protect the memory access patterns, the baseline adopts the traditional Path ORAM implementation [27] and its recent enhancements, as elaborated in the following sections.

B. Path ORAM Basics

Path ORAM is a crypto primitive that protects memory access patterns through obfuscating memory accesses to untrusted external memory [27]. As shown in Fig. 1, Path ORAM organizes the untrusted memory space as a binary tree, referred to as *ORAM tree*. An on-chip ORAM controller converts each user memory request to a path access to the ORAM tree.

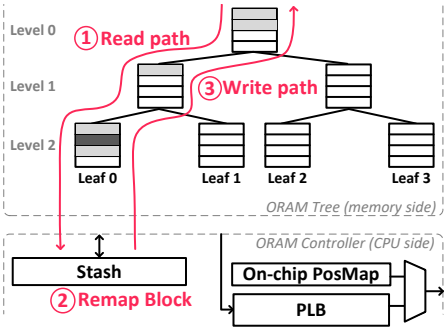


Fig. 1: An overview of Path ORAM ($L = 3$, $Z = 4$).

The ORAM tree has L levels ranging from 0 (the root) to L-1 (the leaves). Each tree node, referred to as a *bucket*, has Z slots to save data blocks, e.g., cache lines in this paper. The slots store either real data blocks or *dummy blocks*. Given all blocks are encrypted, the block types are indistinguishable.

The on-chip ORAM controller consists of control logic, stash, PosMap (position map table) and PLB (PosMap lookaside buffer). The PosMap is a lookup table that maps a block address *BlkAddr* in user address space to a unique *path ID* in the tree. The stash is a small fully associative on-chip buffer that temporarily keeps a number of data blocks (e.g., $100-200$ blocks) and their path IDs. Given PosMap could be big for a large user space, we may need multiple levels of PosMap and store these mapping entries in memory. We use PLB, a small on-chip buffer, to cache the frequently used PosMap entries

such that we can reduce the number of ORAM accesses for PosMap entries.

ORAM operations. Path ORAM converts a memory request with *BlkAddr*=a to one or more path accesses to the ORAM tree. Each path access consist of the following phases.

1) *Stash, PosMap, and PLB access phase.* Before accessing the ORAM tree, the ORAM controller searches the block in the stash and, simultaneously, translates a to a path ID l using PLB/PosMap. The block is returned to the user application if it is found in the stash. Otherwise, the ORAM controller starts the read path phase if the corresponding a-to-l mapping is found in PLB, or fetches the mapping from memory.
2) *Path read phase.* Given a path ID l, the ORAM controller reads all the blocks on l from the memory. It decrypts and authenticates the fetched blocks, and discards the dummy blocks and inserts the real blocks to the stash. This phase generates L×Z memory `read` accesses.
3) *Block remap phase.* After reading the path, the ORAM controller remaps block a to a random new path l'. It then updates the *PosMap* and the stash with the new map.
4) *Path write phase.* After returning the requested block to the user application, the ORAM controller writes data blocks except a back to path l. It searches the blocks in the stash and pushes the data blocks as low as possible in the ORAM tree. Dummy blocks are patched if not enough data blocks can be found. All blocks are encrypted and authenticated before being written to the memory. This phase generates L×Z memory `write` accesses.

Path ORAM may easily deplete the peak off-chip memory bandwidth [25], resulting in large performance degradation not only to itself but also to co-running applications [32]. In summary, the memory bandwidth consumption of Path ORAM has become the main obstacle that prevents its wide integration in modern computer systems.

C. Related ORAM Enhancements

Many schemes have been proposed to improve the performance of Path ORAM. We next discuss several closely related enhancements. Section VII discusses more related work.

PosMap is sensitive metadata and thus requires secure protection. Path ORAM may recursively create a new ORAM tree for the PosMap and then a new level of PosMap. The last level of PosMap is saved in an on-chip buffer. For better access obliviousness and performance, Fletcher *et al.* proposed *Freecursive* to merge all these ORAM trees such that PosMap and data accesses are non-distinguishable [8]. In this paper, we adopt Freecursive in the baseline. In our setting, we construct three levels of PosMap — PosMap_1 and PosMap_2 are merged in the main ORAM tree while PosMap_3 is saved completely in an on-chip buffer.

On average, Path ORAM/Freecursive introduces 8× slowdown over a non-secure execution in our setting. Given this large performance overhead, Path ORAM schemes are currently applied to small applications or application kernels that are highly security sensitive.

Given the number of accessed memory blocks for each path is linear to the path length, Maas *et al.* proposed to buffer a few top levels on-chip [22]. They chose to save the tree top in the stash — they saved up to three levels without incurring considerable stash management overhead. Later studies [23], [32] proposed to buffer ten or more levels, which effectively reduces the memory bandwidth demand. Given the stash is fully associative, maintaining a huge stash with all tree top blocks becomes expensive. The tree top can be kept in a standalone buffer that maintains the tree structure [23].

To achieve high security, Fletcher *et al.* proposed to defend timing channel attacks by issuing path accesses at fixed rate and pattern [9]. A dummy access is inserted if there is no real user request pending. When a Path ORAM implementation has different types of path accesses, it becomes more complicated. For example, Nagarajan *et al.* proposed to construct an extra smaller ORAM tree so that there exists two path lengths [23]. To prevent potential timing channels, they issue path accesses using a fixed pattern, e.g., one main tree access after every four path accesses to the smaller tree. Dummy path accesses of either type are inserted as needed.

III. Motivation

Since Path ORAM suffers mainly from frequent path accesses, we study the path types and the tree access patterns to reveal the opportunities for improvements.

A. The Types of Path Accesses

The preceding discussion reveals that there are three types of path accesses — PosMap paths, data paths, and dummy paths. They are referred to as $\mathbf{PT}_p$, $\mathbf{PT}_d$, and $\mathbf{PT}_m$ paths, respectively. To understand their importance, we conduct an experiment to evaluate the frequencies and summarize the results in Fig. 2. The experiment settings are in Section V. Here, $\texttt{PT}_p\texttt{(Pos1)}$ indicates the memory accesses for fetching PosMap_1 entries, i.e., the mapping from the requested data's block addresses to their ORAM path IDs. $\texttt{PT}_p\texttt{(Pos2)}$ indicates the memory accesses for fetching PosMap_2 entries, i.e., the mapping from PosMap_1 entry's block addresses to their corresponding path IDs. Note that the entire PosMap_3 is saved on-chip. $\texttt{PT}_d$ indicates the memory accesses for fetching the paths that contain the requested data blocks. $\texttt{PT}_m$ indicates the dummy paths that are inserted due to lacking real requests because we need to defend timing channel attacks.

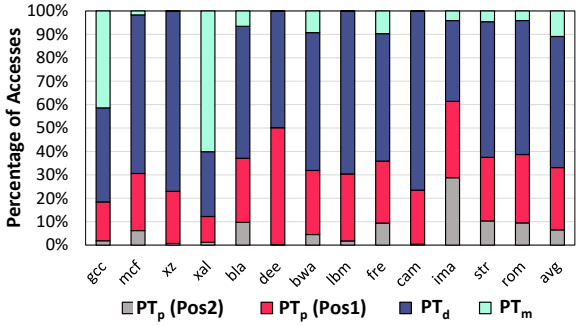


Fig. 2: The distribution of different path accesses.

From the figure, (1) while $\texttt{PT}_d$ accounts for 56% of the total memory accesses, $\texttt{PT}_p$ is non-negligible — they are around 33% of the total. Of these accesses, $\texttt{PT}_p\texttt{(Pos1)}$ is around 4× of $\texttt{PT}_p\texttt{(Pos2)}$, indicating there are many more PosMap_1 misses than PosMap_2 misses to PLB. (2) $\texttt{PT}_m$ accounts for a large portion as well. In this experiment, we set the inter-path time interval T to be the same (T=1000 cycles) for all benchmarks. Setting the same T value achieves the highest security protection across different benchmark programs.

Alternatively, previous studies have shown that it might be possible to set the T value according to the memory intensity of the application. By setting a larger T value, fewer dummy paths may be inserted but real requests may have to wait longer before being serviced, which becomes problematic for bursty memory requests. In addition, an attacker may observe the T value to guess the memory intensity, introducing a potential information leakage channel, e.g., a covert channel.

Note, even though Path ORAM has three path types, its memory access obliviousness is securely enforced, i.e., an attacker cannot determine the type of a particular path access outside of the TCB — *the PATH ORAM controller keeps issuing path accesses at one per T cycles*. This principle is important for analyzing the memory access obliviousness.

To summarize, we conclude that Path ORAM contains three types of path accesses. To effectively reduce its memory bandwidth demand, we may develop schemes to address the memory intensity of every type.

B. The Utilization of Tree Nodes

To prevent stash full and protocol failure, Path ORAM keeps sufficient dummy entries in the ORAM tree — a typical implementation uses around 50% of the protected space to hold real data [25], i.e., we store around 4GB user data in a 8GB ORAM tree, with all the rest being dummy data blocks.

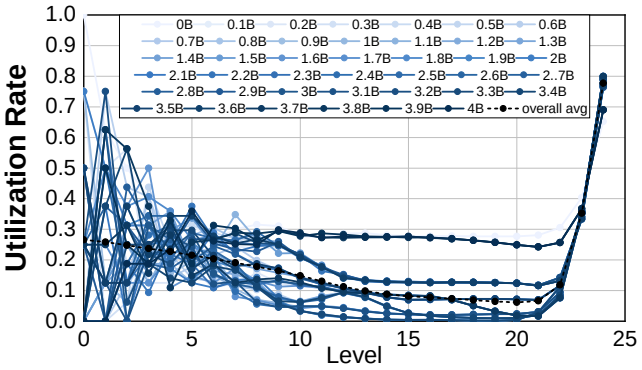


Fig. 3: The space utilization at different tree levels.

Given an ORAM tree consists of both (real) data blocks and dummy blocks, it is worthy to study the distribution of dummy blocks in the tree. Fig. 3 summarizes the results from an experiment for comparing the node space utilization (y-axis) at different levels (x-axis). We take snapshots at different execution times to illustrate the trend. Here, the *space utilization* at a level is defined as the ratio of all useful data blocks to the total allocated memory slots at that level. In this experiment, we protect a 8GB memory space with 4GB user data. We have Z=4, L=25. The block size is 64B.

To initialize the ORAM tree, we clear the tree and access all data blocks once in a random order. For each block, we follow the Path ORAM baseline to remap and write the block to the tree. We create three levels of PosMap mapping tables accordingly. While there are 64 million data blocks in the ORAM tree, we run the memory trace for four billion path accesses, which is sufficiently long to show the *normal* access behavior, as shown in [25], [27]. Due to its excessive length, we use a mix of path accesses from the benchmarks (trace range [0B-3.7B]) and the randomly generated memory accesses (trace range (3.7B, 4B]). In the figure, "XB" indicates the snapshot after executing X billion path accesses, i.e., "0B" is the snapshot right after the initialization. Note that the average line indicates the overall average and not the average of taken snapshots.

Fig. 3 presents the snapshots at different execution points for the typical setting (as shown in the experiment section). While a similar study was reported in [27], they focused on choosing different bucket sizes. Instead, we make the following observations that are new in the literature.

- The top levels (level 0 to around level 9) exhibit large utilization fluctuations. For example, at level 3, the utilization ranges from 11% to 50% and there is no clear stable range for the long execution at each level and across different levels. In general, the fluctuation tends to be more severe for the level that is closer to the tree root.
- The middle levels (level 10 to around level 21) exhibit two utilization ranges for different access patterns. For benchmark accesses that have patterns and data reuse (i.e., program memory traces), the utilization fluctuates at 20% and lower. For random accesses (i.e., synthesized memory traces), the utilization tends to be at around 30%. The utilization towards the end of the execution is around 30% because we place random traces at the end of the trace mix.
- The bottom levels (level 22 to 24) exhibit larger utilization than those of middle levels. In particular, the utilization of the last level tends to be around 70% for random accesses and 80% for patterned accesses. The utilization tends to grow towards that of the last level as they approach the last level.

When we fetch a tree path, each node contributes four data blocks indicating even memory bandwidth consumption across different levels. However, according to Fig. 3, we tend to get more dummy blocks from middle levels due to their low space utilization. In addition, a binary tree has significant capacity imbalance — the space of level l roughly equals to the space of all levels from 0 to l-1. While fetching top and middle levels consumes more memory bandwidth (e.g., 21 out of 25 levels), its space accounts for a small portion, e.g., top 21 levels occupy only 6% of the total space.

Fig. 4 compares the utilization of three different workloads. The results indicate that the utilization trend remains the same for individual workloads.

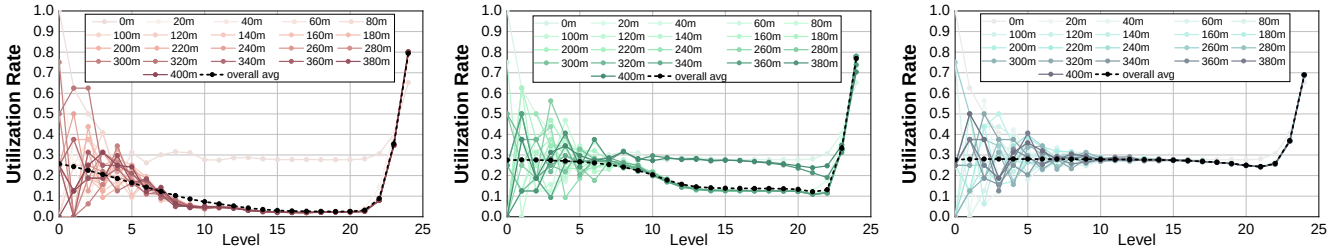


Fig. 4: The space utilization behavior per benchmark; gcc (left), lbm (middle), and a random trace (right).

In summary, there exists a significant mismatch of node space utilization, memory bandwidth, and space capacity across different tree levels.

C. The Block Migration Behavior

To understand the reason why an ORAM tree exhibits distinct utilization difference across different tree levels, we further study how a data block migrates in an ORAM tree.

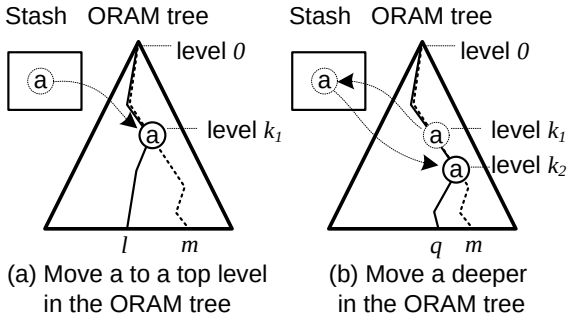


Fig. 5: The migration behavior after a leaves the stash.

In Path ORAM, each path access has two memory phases — a read phase and a write phase. During the write phase, we choose data blocks from the read phase, *pre-existing* data blocks in stash, and/or dummy blocks and place them in the path. A pre-existing block is a block that was in the stash before the read path phase. The observation is, *if we pick up a pre-existing data block, we tend to place it to a top level.* For example, in Fig. 5(a), a is a pre-existing block. Assume we read path l and write blocks back to l, and we pick up a (mapped to path m) from the stash and write it to level k_1. We observe that k_1 tends to be a small value, i.e., k_1 is close to the root. This is because (1) any two paths overlap (because the root belongs to all paths); and (2) two random paths tend to have small path overlap. The latter is true because two paths overlap, e.g., at level 15, only if they belong to the same subtree at level 15. However, there are 32K different subtrees at this level, making the overlap possibility very low. In contrast, there are only 8 subtrees at level 3, it has higher possibility for two paths to overlap at level 3. Of course, due to limited capacity at top levels, a pre-existing data block may not get the chance to be moved to the ORAM tree.

We also observe that *data blocks fetched from the read path are likely to be flushed to the same or lower levels*. Fig. 5(b) illustrates how it works. Assume a (with path ID m) was written to level k_1 (as in Fig. 5(a)). Now, we access another path q where path l and q overlap from level 0 (the root) to level k_2. we may not touch a if $k_1 > k_2$; and write to k_2 if $k_1 \leq k_2$. Due to the low utilization in middle levels, such write is likely to be successful.

We conduct an experiment to compare the node reuse at different levels. Fig. 6 summarizes the hit counts at different levels. From the figure, while top 10 levels account for less 0.01% of total ORAM space, the requested data blocks can be found in these levels for about 23% of total accesses.

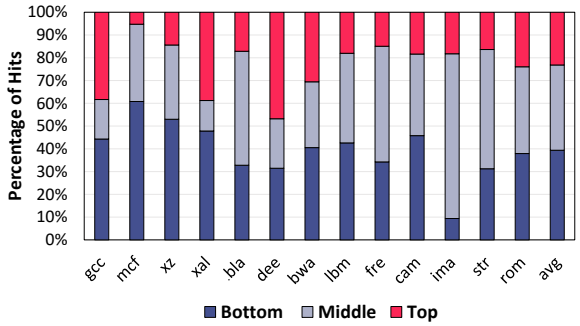


Fig. 6: Nodes at top levels have high reuse possibility.

Therefore, *the top levels of an ORAM tree serve as an overflow buffer of the on-chip stash*. An accessed data block may be buffered temporarily in the stash and then written to the top levels of the tree. As the execution proceeds, a hot block is likely to be brought back to the stash to reuse while a cold block gradually sinks towards the leaf nodes of the tree.

IV. The IR-ORAM Design

A. An Overview

In this paper, we develop IR-ORAM to exploit our findings to reduce the memory intensity of each type of path accesses.

- We develop *IR-Alloc* to reduce the bucket sizes of middle level tree nodes. By exploiting the low utilization of middle level nodes, IR-Alloc reduces the number of data blocks to access for each ORAM path and thus the memory intensity of all three types of paths.
- We develop *IR-Stash* to architect a double-indexed set-associative sub-stash to adapt to large tree top caching. It reduces the number of PosMap accesses and thus the memory intensity of PT_p paths.
- We develop *IR-DWB* to convert dummy path accesses to useful early write-back operations, which reduces the memory intensity of PT_m paths.

B. IR-Alloc: a Utilization-aware Node Size Allocator for Reducing Intensity of PT_p/PT_d /PT_m paths

Traditionally, an ORAM tree uses one Z value, i.e., all buckets have the same number of data blocks. However, our study reveals that nodes at middle levels have considerable low utilization, indicating that 70% or more fetched data from these levels are dummy blocks and thus discarded after fetching. For this reason, it would be beneficial to shrink the bucket size at these levels. As an example, if we shrink the bucket to half of the original, we would read 50% fewer blocks for buckets at these levels.

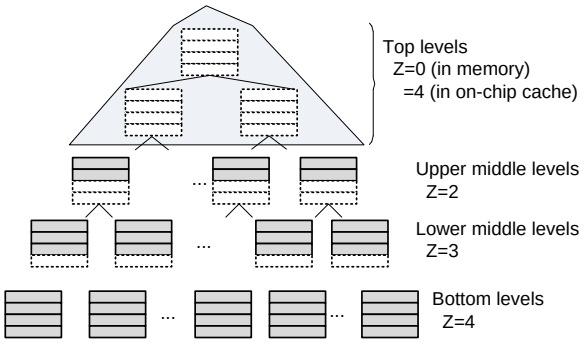


Fig. 7: The design of IR-Alloc scheme.

Fig. 7 illustrates how *IR-Alloc* works. It adopts an allocation strategy with more than two Z values: Z=2, 3, and 4 for tree level ranges [10, 16], [17, 19], and [20, 24], respectively. [1] For completeness, we set Z=0 for memory allocation for tree level range [0, 9]. We will elaborate the details in the next section. With this allocation strategy, *IR-Alloc* only needs to access 43 data blocks (=10×0+7×2+3×3+5×4) during one read (or write) path phase. As a comparison, we need to access 60 and 100 blocks, respectively, for the Path ORAM designs with and without a 10-level top tree cache.

We next study the design issues in *IR-Alloc*. (1) One issue is the reduction of the total ORAM space as there are fewer block slots. This is negligible because the allocation in Fig. 7 only leads to around 0.9% space reduction. This is because most of the space of a binary tree is from lower levels. For example, the space from top 20 levels account for around 3.1% of the total space. (2) Another issue is that, while the middle levels satisfy the overall space demand, a particular tree node may not have the space to hold the chosen data block. Such a tree node may be able to hold that block if adopting the baseline Path ORAM implementation. This is in general not a problem as the ORAM controller would save the data block to a higher tree level, and eventually increase stash usage. As we experience more path accesses, the block still have the chance to sink towards to the leaf node.

The background eviction. The original Path ORAM places blocks that cannot move to the ORAM tree in the stash temporally [27] and faces protocol failure if the stash overflows, i.e., it has insufficient space to hold the blocks after reading a path. Ren *et al.* introduce background eviction, which effectively converts the correctness problem to a performance/overhead trade-off [25].

Since IR-Alloc reduces the number of empty slots on each path, fewer blocks may be moved to the ORAM tree during the write path phase, which increases the possibility of stash overflow. While it does not lead to security concern, too many background evictions may degrade performance significantly. We will analyze its security in Section IV-E and study its performance impact in Section V.

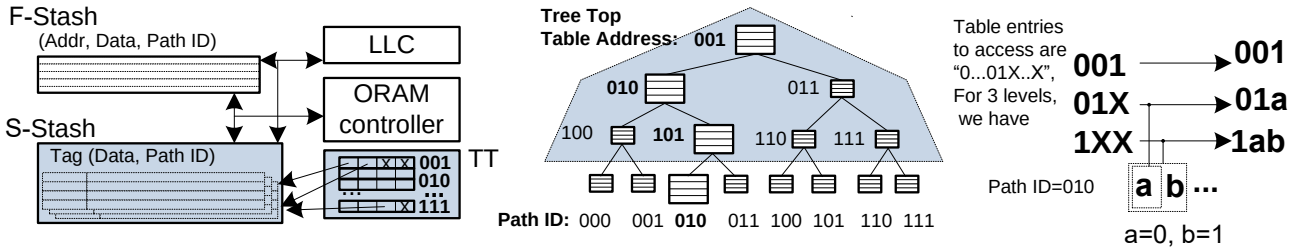
Choosing Z values. In this work, we adopt a greedy search algorithm to determine the Z values at different levels. This algorithm is based on empirical studies (as in the community and also shown in Fig. 3) that a random trace maximizes the utilization of stash entries as well as middle tree levels. Given an ORAM tree, we test run Path ORAM using random memory traces under two constraints: (1) the space reduction is within 1%; and (2) the increase of background evictions is within 15%. According to Fig. 3, random memory traces gives the worst case space utilization for middle levels, the focus of IR-Alloc. We observe that a 15% or less increase of background evictions for random traces exhibits negligible increase of background evictions for benchmark traces. We start with setting Z=3 for level 19 (depending on the tree size) and Z=4 for all other levels, and gradually shrink lower levels Z values for finding a local maximal in performance improvement. The algorithm depends only on the ORAM configuration and, in particular, not on applications. Once the ORAM configuration is determined, we just go through the search process once and make the choice applicable for all deployed systems. Therefore, the search overhead is negligible in general. Note that we may want to run the search twice, once for IR-Alloc and once for IR-Alloc+IR-Stash as the background eviction rate may differ.

C. IR-Stash: a Double Indexed Stash Implementation for Reducing Intensity of PT_p Paths

Caching top tree levels is an effective way for reducing memory intensity. Maas *et al.* proposed to merge the top three levels to the stash and slightly increase the stash size to hold these blocks [22]. However, a major issue of this design is its scalability — the stash needs to be expanded to hold (2^m-1)×Z more blocks if we cache top m levels. In addition, the stash needs to be fully associative. The stash may be accessed by the LLC using its block address (to determine if the requested block is in the stash), or by the ORAM controller using its path ID (to determine the blocks to expurge at write path phase). Therefore, it complicates the access if we make the stash a direct map (or a set associative) cache by indexing it using either the block address or the path ID. Unfortunately, integrating a fully associative buffer is expensive. For example, if we cache top 12 tree levels, the enlarged stash has at least 16K entries. The corresponding die area is about that of a 4MB 8-way set associative LLC. Thus, it becomes less preferable when we need to move more top tree levels on-chip.

An alternative design is to buffer tree top in a dedicated on-chip cache and access them according to the tree structure,

[1] Here we use the math range representation, i.e., level range [a, b] indicates levels a, a+1, a+2, ..., b.



(a) A set-associative stash to cache tree top (b) An example showing how to access tree top entries

Fig. 8: Exploit IR-Stash to effectively cache large tree top.

i.e., as they are in the memory [32], [39]. The stash can remain small (below 200 entries). In this design, the dedicated cache can only be accessed by the ORAM controller and thus is invisible to the LLC. Each ORAM path consists of two segments — top levels are in the buffer while the others are in the memory. The dedicated cache design harvests most of the benefits in caching. In particular, when reading a path, we search the buffer first. A buffer hit eliminates off-chip traffic and thus there is no need to remap.

A major issue with the dedicated cache design is the increased PosMap accesses. To determine if a data block is in the dedicated cache, the LLC needs to know the path ID of the block, which demands finding its Posmap entry. This access is necessary if the block is in the memory. However, if the block is in the cache, the PosMap access is a waste. Based on the discussion in Section III-C, the top tree levels have a small size but a higher reuse possibility, which increases a non-negligible number of PosMap accesses. A PosMap access, if missed in PLB, results in a full path access, which significantly degrades the Path ORAM performance.

The IR-Stash Design. Fig. 8 illustrates our *IR-Stash* design. Intuitively, we include a large sub-stash for buffering the tree top entries and make it double-indexed to facilitate both LLC and ORAM accesses. The reason why IR-Stash can reduce the intensity of PT_p paths is that, if the tree top is indexed only by block addresses, a large number of PosMap accesses would be required to support ORAM path accesses. IR-Stash consists of two sub components.

(1) A small fully-associative stash *F-Stash* that has around 200 entries. *F-Stash* is the same as the traditional stash.

(2) A set-associative stash *S-Stash* that keeps blocks from the tree top. *S-Stash* is **double-indexed**. For the cache organization, it is indexed using the block address (in user space).

S-Stash is also indexed by a small pointer table TT, which helps to keep the tree structure of the blocks in *S-Stash*. Each entry in TT corresponds to a bucket and saves four pointers pointing to the data entries in *S-Stash* that save the data blocks contents and their path IDs. We skip the code with all zeros, assign the code "00..01" as the table address of the root node, and then continue the table address assignment level-by-level according to the tree structure.

In the example in Fig. 8(b), we cache top three levels and illustrate their table addresses.

We next describe how IR-Stash supports the accesses from both LLC and the ORAM controller. When LLC issues a request for a data block, it uses its block address to search both sub-stashes in parallel. Given *F-Stash* is small and fully associative, and *S-Stash* is set indexed by block addresses, the access is fast and incurs low access overhead. **A hit in either F-stash or S-Stash returns the block immediately and thus incurs no path access or path remapping.**

When the ORAM controller needs to access the tree top using a path ID, it follows the same Path ORAM protocol. The only difference is that the tree top is stored on-chip and thus uses TT table to identify the corresponding entries in *S-Stash*. To read a path, the ORAM controller reads the memory portion of the path and then the on-chip portion. For the on-chip portion, we need to access multiple buckets and access each bucket using its table address. By employing the coding strategy as discussed above, we can infer the tables addresses of these buckets. Note that maintaining TT is necessary because *S-Stash* is indexed by block address (in user space) which is different from the physical address of block location in the tree.

Assume the path to access is "$a_1a_2...a_i....$" ($1 \leq i \leq L$-1)" and we cache top t levels on-chip, we need to access t buckets and their table addresses are:

$$\overbrace{00...00}^{\text{(t-1) zeros}}1,\ \overbrace{00...00}^{\text{(t-2) zeros}}1a_1,....,\ \overbrace{0}^{\text{1 zero}}1a_1...a_{t-2},\ \ 1a_1...a_{t-1}$$

For each table entry, we follow its four pointers to fetch the block contents and their path IDs in *S-Stash*. To write a path, we follow the Path ORAM protocol to search *F-Stash* and choose the data blocks to write back at each level. To improve caching effectiveness and avoid address conflicts, *S-Stash* indexes the blocks using MD5 of their addresses. Our experiments show that it evenly distributes the blocks.

When the ORAM controller fills in new blocks in the treetop buckets, we enter the chosen blocks in S-Stash and update the pointers accordingly. If a block from *F-Stash* cannot be moved to *S-Stash* because the target cache set is full, we skip picking this block for this round and get it flushed to *S-Stash* or a memory bucket at a later time.

At context switch, we flush the entries in F-Stash to the ORAM tree. Since the entries in S-Stash are cached tree bucket entries, they are encrypted, authenticated, and then written back to their corresponding memory locations. The TT table is then discarded. We rebuild the table to resume the execution.

D. IR-DWB: Converting Dummy Path Accesses for Reducing Intensity of PT_m paths

To mitigate the performance impact of dummy path accesses, we propose *IR-DWB* to convert them to useful memory accesses. *IR-DWB* converts dummy accesses into free early write-back of dirty blocks in LLC. Intuitively, *IR-DWB* searches for the dirty LLC entries that are likely to be written back in the near future, writes them to the memory of these entries, and marks these entries as clean so that it incurs low overhead when they are selected for replacement. Since dummy accesses are exploited for this matter, these early write-backs occur at no extra cost in terms of performance.

Fig. 9 illustrates how *IR-DWB* works. It consists of two main subtasks: (1) finding the appropriate dirty LLC entry; and (2) writing the dirty entry to memory. For the first subtask, we keep a register Ptr that points to the dirty LRU entry of one LLC cache set. We round-robin across all sets and search for the LRU entry when the LLC is idle. If the LRU entry of the current set is not dirty, we proceed to the next cache set. If the pointed entry is accessed and thus no longer an LRU entry, we clear Ptr (even if it is locked as discussed next) and proceed to the next cache set. If no entry can be found, we pause the search for 1000 cycles and restart from a random set. To implement a round-robin search, we used a small state-machine similar to autonomous eager writeback in [18] that is also adopted by [28], [34]. Alternatively, candidates can be chosen by maintaining a queue as in eager queue scheme in [18]. The latter approach can be adopted in case hardware overhead is a tight constraint.

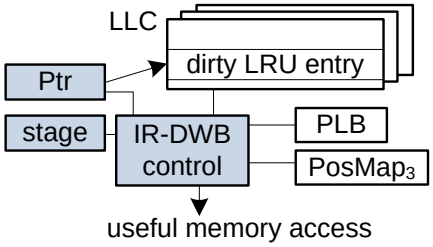


Fig. 9: The design of IR-DWB scheme.

For the second subtask, we need up to three ORAM path accesses due to PosMap and data accesses, i.e., two ORAM path accesses for PosMap_1 and PosMap_2, respectively, and one path access for the dirty data block. For this purpose, we keep a register `Stage` to indicate if the corresponding LLC entry is ready to be written to the memory. We set `Stage`=3 if neither $PosMap_1$ or $PosMap_2$ mapping can be found in PLB; `Stage`=2 if $PosMap_2$ is a PLB hit while $PosMap_1$ is a miss; and `Stage`=1 if both can be found in PLB.

To defend timing channel attacks, Path ORAM issues path accesses at fixed rate and inserts dummy paths when there is no real request. IR-DWB optimizes the implementation as follows. When it is time to issue a dummy path access, *IR-DWB* checks if `Stage`=0 (indicating no LRU entry write-back is in progress). If `Stage`$\neq$0, *IR-DWB* continues the unfinished dirty entry flush; otherwise, it locks Ptr and proceeds to flush the dirty LRU entry pointed by Ptr. Depending on if we need PosMap accesses, we set `Stage`=3 for accessing $PosMap_2$ and `Stage`=2 for accessing $PosMap_1$; and `Stage`=1 if both PosMap entries are ready so that we can write the dirty data block. We decrement `Stage` after each path access and mark the corresponding LLC entry as clean when `Stage`=0. During this processing, if the dirty LLC entry pointed by Ptr is no longer a LRU entry, we abort the early eviction by setting `Stage`=0; or if the entry is chosen as a victim entry, we abort the early eviction by setting `Stage`=0 and perform the normal eviction instead.

From the discussion, for maximized benefit, IR-DWB requires three dummy path accesses to write the selected LLC entry back to the memory. In practice, we gain benefits if we have one or two dummy path accesses and thus can only partially service the request.

Delayed Block Remapping. IR-DWB currently works with the traditional LLC eviction strategy, i.e., a data block is remapped after its access; it is then placed in the LLC [8], [27], [37]. An LLC-evicted data block, if being dirty, becomes a new memory access. Alternatively, Nagarajan *et al.* [23] proposed a delayed data block remapping policy [23]. It works as follows. After accessing a data block, the ORAM controller discards its mapping, which eliminates the data block from the ORAM tree. The data block is added back to the ORAM tree when it is evicted from the LLC. Under this policy, the writeback block uses the ORAM-removed block in stash and thus shall not increase stash pressure. However, there is a limitation, i.e., it demands PosMap accesses at write-back time. Given the corresponding PosMap entry for servicing the initial access may not be in the PLB, it needs up to two extra full path accesses for PosMap entries. The LLC and memory are not inclusive under this policy.

Comparing the two data block remapping policies, the delayed remapping tends to introduce extra PosMap access overhead when most of evicted data blocks from LLC are clean. As shown in the experiment section, while the delayed remapping improves the overall baseline performance, it may slowdown the read intensive benchmarks by more than 50%.

To integrate IR-DWB with delay data block remapping, we may proactively conduct block remapping for LRU entries in LLC, and convert dummy paths to PosMap accesses to eliminate the overhead at write-back. We may need extra table for the generated remapping and thus leave it to future work.

Comparison. *IR-DWB* shares the similarity with *eager writeback* [18] for speeding up the traditional write-back LLC. Both schemes evict the dirty LLC entries before they are chosen for replacement. However, they differ as follows. (1) *eager writeback* tends to increase off-chip memory bandwidth demand. *IR-DWB* only exploits dummy path accesses. Since dummy path accesses consume the bandwidth anyway, *IR-DWB* does not introduce extra memory bandwidth demand. (2) *eager writeback* may degrade the program execution as an ongoing writeback request may block a user request that otherwise can be issued. *IR-DWB* does not hurt the current or the next request as a dummy path access needs to finish

before the ORAM controller may issue another path access. Converting the dummy access does not degrade the execution. (3) one *eager writeback* can be serviced by one extra memory access while one *IR-DWB* may need up to three dummy paths.

E. Security and Correctness Analysis

Security Guarantee. Path ORAM is a security primitive that protects user privacy through memory address obfuscation. It achieves memory obliviousness with two uniformity.

(1) **Path accesses are not distinguishable**. Even though there exists read or write user requests, and ORAM path accesses can be categorized as three types (data block access, dummy path access, and PosMap access), all path accesses look the same. An attacker outside of TCB cannot distinguish either the memory request type or the path access type.
Note, each path consists of one bucket access (or Z block accesses) to each tree level. Given the memory addresses are in cleartext, the tree access is non-oblivious, i.e., an attacker knows exactly when and what tree nodes are accessed.

(2) **Access intensity is not distinguishable**. By enforcing timing attack protection, the ORAM controller issues one path access every T cycles. An attacker outside of TCB cannot infer the path access type or application behavior. In particular, if timing attack protection is not enforced, the first path access of a burst of several path accesses is more likely to be a PosMap access.

In this section, we show that IR-ORAM enforces the above uniformity and thus the same level of security protection. While IR-Alloc reduces the number of blocks read from some tree levels, all paths are kept the same. Each individual path fetches the same number of block from a particular tree level so that we cannot distinguish the type of a particular path from the rest of all others. Similarly, eliminating access tree top nodes in IR-Stash and converting dummy paths to useful paths in IR-DWB keep the obliviousness of path accesses.

The non-uniformity introduced in IR-Alloc, e.g., we fetch two blocks from level 12 and four blocks from level 23, leaks no sensitive information. This is because memory addresses are in cleartext, and the block-to-tree-level mapping is public information in the original Path ORAM design. Since it is well known how the bucket sizes are adjusted, exposing node level non-uniformity has no security concerns. For the top tree cache design, we will not start off-chip memory accesses until we know if the requested block is in the on-chip sub-stashes and prevent potential information leakage.

In summary, the ORAM paths in IR-ORAM are kept the same pattern and at the fixed rate. Thus, it ensures the same level of security protection as that in the original Path ORAM.

Correctness Guarantee. IR-ORAM does not introduce correctness issue either. IR-ORAM changes the way how the stash and the ORAM tree are constructed, which increases the possibility of stash overflow. In the basic Path ORAM implementation, the protocol fails if a stash overflow happens. Ren *et al.* introduced background eviction [25], which effectively converts the protocol correctness problem to a performance/overhead trade-off. In IR-ORAM, we enable background eviction if there are more blocks than a threshold after any write path phase. In summary, IR-ORAM does not introduce correctness issue to Path ORAM.

V. Experiment Methodology

To evaluate IR-ORAM, we used a trace-based simulator USIMM for cycle-accurate DRAM memory simulation [6]. This is similar to the setting that was adopted in recent Path ORAM studies [23], [32]. As shown in Table I, we modeled a 4-issue OoO (out-of-order) 3.2GHz processor. There are two levels of data cache with the LLC (last level cache) being 8-way set associative 2MB. We modeled the ORAM tree following the typical setting [8], [23], [38] — we protect 8GB memory with 4GB user data. We use Z=4, L=25 for baseline.

To collect the trace for evaluation, we used Pin tool [21] on SPEC CPU2017 suite [1]. For each program, we skipped the warmup phase and then collected the trace that covers 2M L1 cache misses. We also picked a few traces obtained from PARSEC suite [5], [6]. Table II lists the L2 misses per kilo instruction (MPKI) for all the benchmarks we picked.

TABLE I: System configuration.

Processor Configuration	
Processor Fetch Width/ ROB Size	4 / 128
Memory Channels	4
DRAM Clk Frequency	800 MHz
L1 D-cache	2-way 256KB
L2 cache (LLC)	8-way 2MB
ORAM Configuration	
Protected space and user data	8GB/4GB
ORAM tree levels	25
Bucket size/Block size	4 / 64B
Stash entries	200
Dedicated tree top cache	256KB (4K entries)
On-chip PLB / PosMap	64KB / 512KB

TABLE II: Evaluated benchmarks.

Suite	Benchmark	read MPKI	write MPKI	Benchmark	read MPKI	write MPKI
SPEC	gcc	0.1	0.3	bwa	0.0	20.7
	mcf	19.5	0.1	lbm	0.0	45.3
	xz	24.9	29.6	cam	0.01	8.8
	xal	0.05	0.1	ima	0.3	2.9
	dee	0.0	5.7	rom	0.02	23.0
PARSEC	bla	2.6	0.4	fre	2.1	0.4
	str	2.7	0.5			

VI. Experimental Results

We implemented and compared the following schemes.

- `Baseline`: this is the traditional Path ORAM implementation [27] that adopts Freecursive [8] and has ten top tree levels cached in dedicated on-chip cache. It also adopts the subtree layout to improve row buffer hits and background eviction to prevent stash overflow [25].

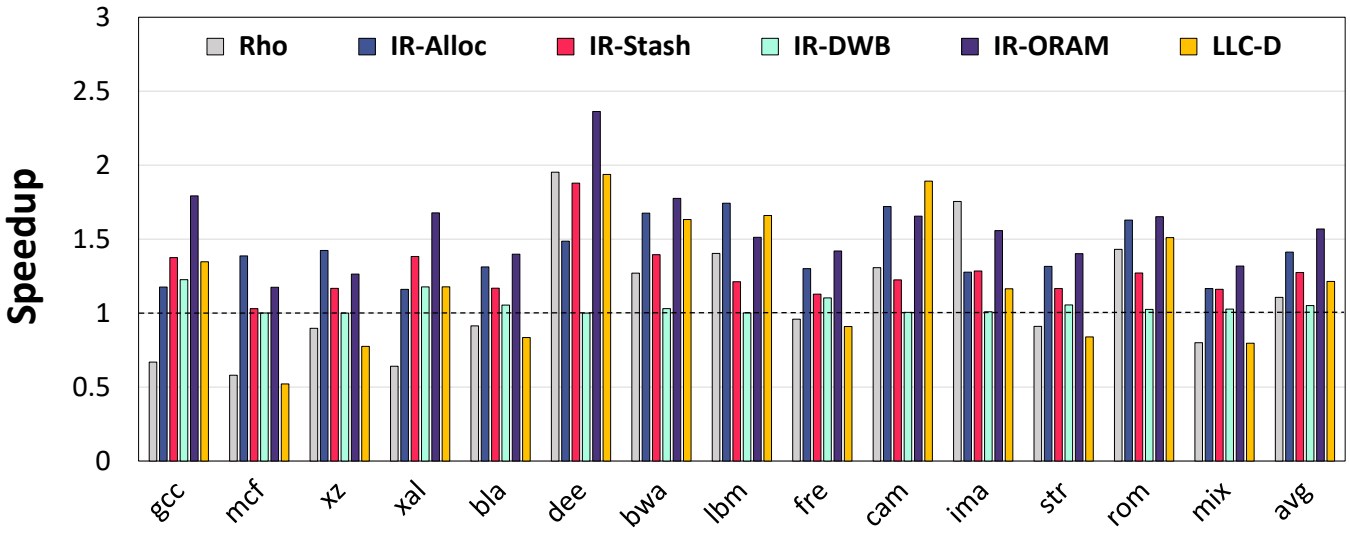


Fig. 10: The performance comparison of different schemes.

- `Rho`: it is the ρ design [23] over `Baseline`. `Rho` implements a smaller tree and several other optimizations. We chose the best setting (L=19, Z=2) for the small tree, included other optimizations, and enforced the defense for timing channel attacks (we used 1:2, i.e., one main tree access per two accesses to the smaller tree).
- `IR-Alloc`: it implements IR-Alloc over `Baseline`. We set Z=1 for tree level range [10,15], and Z=2 for [16,18].
- `IR-Stash`: it implements IR-Stash over `Baseline`. For *S-Stash*, we tested different set associativities and choose 4-way set associative in this paper.
- `IR-DWB`: it implements IR-DWB over `Baseline`.
- `IR-ORAM`: it integrates all three designs in our paper. It is built on top of `Baseline`. (It sets Z to 2 and 3 for tree level ranges [10,16] and [17,19], respectively.)
- `LLC-D`: it adopts the delayed data block remapping policy [23] on top of `Baseline`.
- `IR-Stash+IR-Alloc` (LLC-D as baseline): it is `IR-Alloc` and `IR-Stash` on top of `LLC-D`.

A. Performance Comparison

Fig. 10 compares the performance of different schemes. The results are normalized to `Baseline`. mix bar indicates mix trace of 3 different benchmarks. From the figure, `Rho` achieves an average of 11% improvement. It exhibits a large degradation on mcf. This is due to the mechanism integrated for defending timing channel attacks, which inserts many dummy paths and offsets the benefit from accessing the smaller tree. This reduction may be mitigated if making the defense application-specific (currently we used 1:2 ratio).

For our schemes, `IR-Alloc` achieves on average 41% improvement over `Baseline`. The improvement comes mainly from the reduced number of data blocks to access for each path. Since we reduce the memory intensity of all path types, the improvements are stable across all benchmark programs. `IR-Stash` achieves on average 27% improvement over `Baseline`. Given both `Baseline` and `IR-Stash` cache top ten levels of the tree, the improvement comes mainly from the reduction of PosMap accesses. `IR-DWB` achieves on average 5% performance improvement. When there were more dummy path accesses, e.g., gcc in the figure, we found more opportunities for conversion, which helped to achieve higher improvement. For benchmarks having few dummy path accesses, e.g., cam and dee, `IR-DWB` was rarely activated, which led to close to zero performance impact.

When enabling all three proposed schemes, `IR-ORAM` achieves on average 57% improvement over `Baseline`, or 42% improvement over `Rho`.

`LLC-D` improves `Baseline` for most of the benchmarks due to their high dirty eviction rate. However, a read intensive benchmark like mcf experiences 1.9× slowdown. Fig. 11 illustrates the speedup of `IR-Stash+IR-Alloc` over a baseline that adopts `LLC-D`. Our two schemes can effectively improve a baseline with `LLC-D` adopted by 72% on average. We observe a high speedup of 1.63× for mcf. This is because, with `LLC-D` baseline, the number of hits in the tree top triples for this benchmark such that `IR-Stash` finds more opportunities to reduce the number of PT_p path accesses.

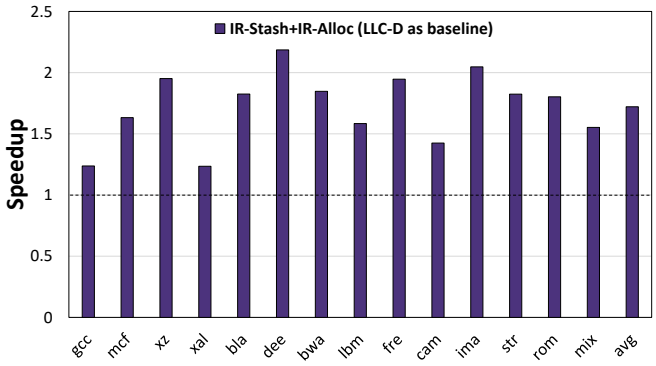


Fig. 11: The performance comparison with LLC-D as baseline.

Since `IR-Stash` and `IR-Alloc` are orthogonal to timing channel protection feature, we also measured their speedup without having timing channel protection. Our results showed

that IR-Alloc achieves slightly smaller speedup compared to when timing channel protection was enabled (40% vs 41% in Fig. 10). This is expected because with timing channel protection being enabled some background eviction accesses are reduced due to existence of inevitable dummy accesses.

By developing path type dependent techniques, IR-ORAM effectively reduces the memory intensity of the Path ORAM.

B. IR-Alloc Overflow

While IR-Alloc changes the Z values for two tree level ranges, the discussion in Section IV-B indicates that this selection is not unique — we may choose different Z values for different tree level ranges, and all such selections may give good performance improvements.

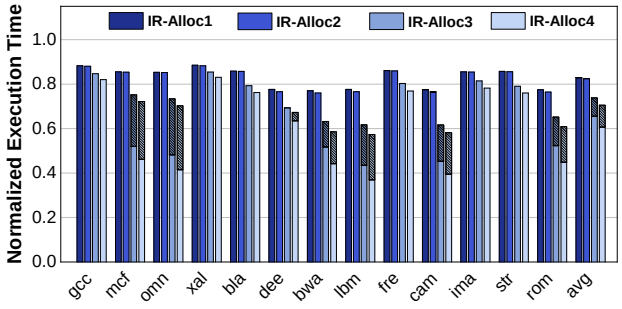


Fig. 12: Design exploration of IR-Alloc scheme.

To study the selection of Z values, we composed four configurations that set Z values for different tree level ranges as follows. PL indicates the number of data blocks we need to fetch per path. For all configurations, the memory shrinks below 1%. Of course, there are more configurations available.

- **IR-Alloc1: Z=2** for L10∼16, **Z=3** for L17∼19 .... **PL=43**
- **IR-Alloc2: Z=2** for L10∼16, **Z=2** for L17∼18 .... **PL=42**
- **IR-Alloc3: Z=1** for L10∼14, **Z=2** for L15∼18 .... **PL=37**
- **IR-Alloc4: Z=1** for L10∼15, **Z=2** for L16∼18 .... **PL=36**

Fig. 12 compares the performance of different IR-Alloc configurations. The results are normalized to execution time of Baseline. For each bar, the shaded portion indicates the time spent on background eviction. IR-Alloc4 is the same as IR-Alloc in Fig. 10. From the figure, in general, we tend to achieve larger performance improvement by reducing the number of data blocks to access per ORAM path. In addition, aggressively reducing the number of data blocks tends to incur large time in background eviction.

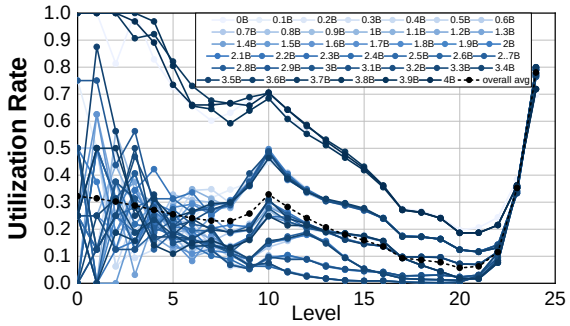


Fig. 13: Level utilization with IR-Alloc.

To study background eviction, Fig. 13 shows the level utilization experiment similar to that in Section III-B for IR-Alloc. Snapshots were taken at different times of the execution using the same memory trace mix as in Fig. 3.

From the figure, we observed that, for memory accesses from benchmarks, the top and middle tree levels show high space utilization ratios than those before adopting IR-Alloc, i.e., the utilization ratios in Fig. 3. However, they are still low, meaning in general, the background eviction is still low.

For randomized traces, the utilization ratios are much higher, i.e., more than 50% utilization. In particular, we rarely found empty slots for tree nodes between level 0 and level 3. This indicates there is a high possibility of stash overflow. However, most benchmark programs have stable working sets and IR-Alloc achieves performance improvements over the Path ORAM implementation.

C. PosMap Reduction

We next investigated the effectiveness of IR-Stash in reducing position map accesses. Fig. 14 reports the normalized PosMap accesses of IR-Stash over Baseline. From the figure, IR-Stash dramatically reduces the number of PosMap accesses — on average, the number of PosMap accesses in IR-Stash are 49% of those in Baseline.

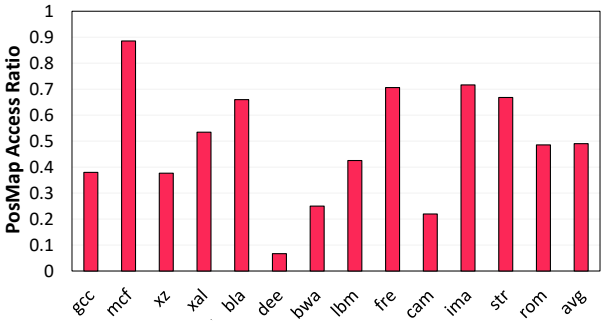


Fig. 14: Comparing PosMap accesses with Baseline.

For benchmarks that have large reduction, e.g., 94% for dee, IR-Stash achieves large improvement of 87%. For the one with small reduction, e.g., mcf, it achieves low improvement.

D. Dummy Path Accesses

IR-DWB is designed to exploit dummy accesses in timing channel protection mode for early write-backs. Fig. 10 shows its speedup. We also investigated its effectiveness in converting dummy accesses. Fig. 15 illustrates the percentage of each path

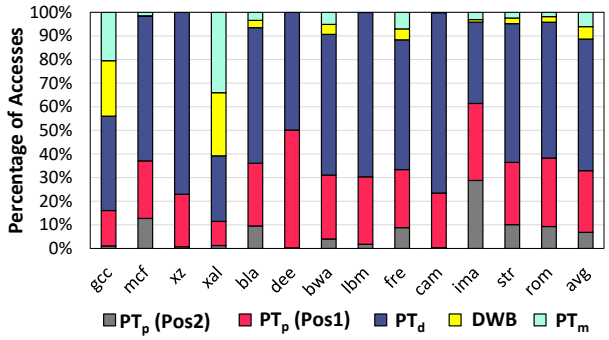


Fig. 15: Access type distribution in IR-DWB.

access type. On average, IR-DWB reduces the percentage of dummy accesses from 11% to 6%.

E. Scalability Analysis

To evaluate the scalability of IR-Alloc, we used different sizes of protected memory, i.e., 2GB (L=24) and 8GB (L=26), and summarized the results in Fig. 16. For each configuration, we applied the Z finding algorithm accordingly to find the appropriate Z value at each level. We used the random traces as they set the performance lower bound while exhibiting high probability in background eviction. Fig. 16 compares the speedup of IR-Alloc over Baseline for different memory sizes. The x-axis indicates the size of user data which is half of the entire protected space. We conducted the experiment for 13 different random traces and reported the average speedup. The standard deviation of different speedup results is low (=0.0001) as random traces lack locality.

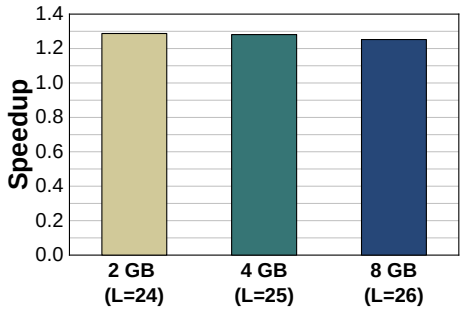


Fig. 16: The scalability analysis of IR-Alloc.

Impact of block size. When adopting larger blocks, e.g., by a cloud server with remote clients, the PosMap becomes smaller, which may be stored completely within TCB. This eliminates the potentials that IR-ORAM can explore from IR-Stash. However, the benefits from IR-Alloc remain untouched.

F. Overheads

Space overhead. By reducing the bucket sizes of selected tree levels, IR-Alloc reduces the total memory space. However, this reduction is negligible, the different IR-Alloc configurations in Section VI-B keep the space loss below 1%.

The IR-Stash design, comparing to the dedicated tree top cache design, keeps the tree structure in a small table, which saves $(2^{10}-1)\times 4$ pointers and each pointer is of 12 bits. The total size is 6KB. In addition, it saves the tag array that the dedicated tree top cache may not need. This overhead is modest comparing to the enlarged stash design.

Energy overhead. The energy overheads comes from (1) the extra stash evictions introduced by IR-Alloc; (2) the extra table lookups in IR-Stash; and (3) the extra LLC/PLB accesses for finding the IR-DWB candidates. Given the energy consumption of Path ORAM comes mainly from memory accesses, the on-chip activities are negligible. For example, one access to 256KB cache is around 0.6nJ while an access to 1GB memory is about 40nJ [4]. The more extra stash evictions, the higher energy overhead the IR-ORAM may have. In our experiments, the number of extra stash evictions is low, incurring less than 5% of total energy consumption. Our energy saving in the memory systems is proportional to performance improvement, i.e., about 57% over Baseline.

VII. Related Work

Ren *et al.* proposed Ring ORAM to reduce memory bandwidth at path access time with background tree reshuffle [24]. IR-ORAM adjusts the construction of the ORAM tree and thus is orthogonal and can be integrated to achieve better performance. Yu *et al.* proposed PrORAM to improve Path ORAM performance by constructing superblocks for prefetching [37]. Zhang *et al.* proposed to eliminate accessing overlapped portion of consecutive paths [39]. Wang *et al.* proposed to mitigate the interference of Path ORAM on co-running processes on the same server [32]. Path ORAM was also enhanced for its adoption for cloud services [20], [26].

By adopting secure memory architectures, memory access patterns on the buses can be effectively protected with hardware enhancements [2], [3]. Wang *et al.* proposed to achieve high level privacy protection by adopting Path ORAM for emerging BOB (buffer-on-board) memory architecture [33]. In addition to defending user privacy, hardware-assisted security enhancements are developed to mitigate the performance impact of data authentication [30].

VIII. Conclusions

In this paper, we propose IR-ORAM, a Path ORAM optimization that consists of a set of techniques, to mitigate the high memory intensity of Path ORAM. In particular, we propose IR-Alloc to reduce the number of data blocks that we need to access for each tree path; IR-Stash to buffer top tree levels, which hybrids the extended stash and dedicated cache designs to achieve effective PosMap access reduction; IR-DWB to convert a significant portion of dummy path accesses to write back dirty LRU entries in LLC. On average, IR-ORAM achieves 42% performance improvement over the state-of-the-art while effectively enforcing the memory access obliviousness and the same level of security protection.

Acknowledgements

We thank the anonymous reviewers for their constructive suggestions. The work is supported in part by NSF CCF# 2011146 and NSF CCF# 1910413.

References

[1] *SPEC CPU 2017 Benchmark Suite*, 2017. [Online]. Available: https://www.spec.org/cpu2017

[2] S. Aga and S. Narayanasamy, "Invisimem: Smart memory defenses for memory bus side channel," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, 2017.

[3] A. Awad, Y. Wang, D. Shands, and Y. Solihin, "Obfusmem: A low-overhead access obfuscation for trusted memories," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture*, 2017.

[4] R. Balasubramonian, A. B. Kahng, N. Muralimanohar, A. Shafiee, and V. Srinivas, "Cacti 7: New tools for interconnect exploration in innovative off-chip memories," *ACM Transactions on Architecture and Code Optimization*, 2017.

[5] C. Bienia, "Benchmarking modern multiprocessors," Ph.D. dissertation, Princeton University, 2011.

[6] N. Chatterjee, R. Balasubramonian, M. Shevgoor, S. H. Pugsley, A. N. Udipi, A. Shafiee, K. Sudan, M. Awasthi, and Z. Chishti, "Usimm : the utah simulated memory module," 2012.

[7] I. Damgård, S. Meldgaard, and J. B. Nielsen, "Perfectly secure oblivious ram without random oracles," in *Proceedings of the 8th Conference on Theory of Cryptography*, 2011.

[8] C. W. Fletcher, L. Ren, A. Kwon, M. van Dijk, and S. Devadas, "Freecursive oram: [nearly] free recursion and integrity verification for position-based oblivious ram," in *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2015.

[9] C. W. Fletchery, L. Ren, X. Yu, M. Van Dijk, O. Khan, and S. Devadas, "Suppressing the oblivious ram timing channel while making information leakage and program efficiency trade-offs," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture*, 2014.

[10] B. Gassend, E. Suh, D. Clarke, M. Van Dijk, and S. Devadas, "Caches and merkle trees for efficient memory authentication," in *Proceedings of 9th International Symposium on High Performance Computer Architecture*, 2003.

[11] O. Goldreich, "Towards a theory of software protection and simulation by oblivious rams," in *Proceedings of the 19th Annual ACM Symposium on Theory of Computing*, 1987.

[12] O. Goldreich and R. Ostrovsky, "Software protection and simulation on oblivious rams," *J. ACM*, vol. 43, 1996.

[13] M. T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia, "Privacy-preserving group data access via stateless oblivious ram simulation," in *Proceedings of the 23rd Annual ACM-SIAM Symposium on Discrete Algorithms*, 2012.

[14] Intel, *Intel Software Guard Extensions*, 2014. [Online]. Available: https://software.intel.com/en-us/sgx

[15] M. S. Islam, M. Kuzu, and M. Kantarcioglu, "Access pattern disclosure on searchable encryption: Ramification, attack and mitigation," in *19th Annual Network and Distributed System Security Symposium*, 2012.

[16] B. Jacob, S. Ng, and D. Wang, *Memory Systems: Cache, DRAM, Disk*, 2007.

[17] L. M. Kaufman, "Data security in the world of cloud computing," *IEEE Security Privacy*, 2009.

[18] H.-H. S. Lee, G. S. Tyson, and M. K. Farrens, "Eager writeback- a technique for improving bandwidth utilization," in *Proceedings 33rd Annual IEEE/ACM International Symposium on Microarchitecture*, 2000.

[19] C. Liu, M. Hicks, and E. Shi, "Memory trace oblivious program execution," in *2013 IEEE 26th Computer Security Foundations Symposium*, 2013.

[20] L. Liu, R. Wang, Y. Zhang, and J. Yang, "H-oram: A cacheable oram interface for efficient 1/o accesses," in *2019 56th ACM/IEEE Design Automation Conference*, 2019.

[21] 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 *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2005.

[22] M. Maas, E. Love, E. Stefanov, M. Tiwari, E. Shi, K. Asanovic, J. Kubiatowicz, and D. Song, "Phantom: Practical oblivious computation in a secure processor," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, 2013.

[23] C. Nagarajan, A. Shafiee, R. Balasubramonian, and M. Tiwari, "ρ: Relaxed hierarchical oram," in *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019.

[24] L. Ren, C. W. Fletcher, A. Kwon, E. Stefanov, E. Shi, M. van Dijk, and S. Devadas, "Ring ORAM: closing the gap between small and large client storage oblivious RAM," *IACR Cryptol. ePrint Arch.*, 2014.

[25] L. Ren, X. Yu, C. W. Fletcher, M. van Dijk, and S. Devadas, "Design space exploration and optimization of path oblivious ram in secure processors," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, 2013.

[26] S. Sasy, S. Gorbunov, and C. W. Fletcher, "Zerotrace : Oblivious memory primitives from intel SGX," in *25th Annual Network and Distributed System Security Symposium*, 2018.

[27] E. Stefanov, M. van Dijk, E. Shi, C. W. Fletcher, L. Ren, X. Yu, and S. Devadas, "Path oram: An extremely simple oblivious ram protocol," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, 2013.

[28] J. Stuecheli, D. Kaseridis, D. Daly, H. C. Hunter, and L. K. John, "The virtual write queue: coordinating dram and last-level cache policies," 2010.

[29] G. E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas, "Aegis: Architecture for tamper-evident and tamper-resistant processing," in *Proceedings of the 17th Annual International Conference on Supercomputing*, 2003.

[30] M. Taassori, A. Shafiee, and R. Balasubramonian, "VAULT: reducing paging overheads in SGX with efficient integrity verification structures," in *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems*, X. Shen, J. Tuck, R. Bianchini, and V. Sarkar, Eds., 2018.

[31] D. L. C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz, "Architectural support for copy and tamper resistant software," in *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2000.

[32] R. Wang, Y. Zhang, and J. Yang, "Cooperative path-oram for effective memory bandwidth sharing in server settings," in *2017 IEEE International Symposium on High Performance Computer Architecture*, 2017.

[33] R. Wang, Y. Zhang, and J. Yang, "D-oram: Path-oram delegation for low execution interference on cloud servers with untrusted memory," in *2018 IEEE International Symposium on High Performance Computer Architecture*, 2018.

[34] Z. Wang, S. M. Khan, and D. A. Jiménez, "Improving writeback efficiency with decoupled last-write prediction," in *2012 39th Annual International Symposium on Computer Architecture (ISCA)*, 2012.

[35] P. Williams and R. Sion, "Single round access privacy on outsourced storage," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, 2012.

[36] J. Yang, Y. Zhang, and L. Gao, "Fast secure processor for inhibiting software piracy and tampering," in *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, 2003.

[37] X. Yu, S. K. Haider, L. Ren, C. Fletcher, A. Kwon, M. van Dijk, and S. Devadas, "Proram: Dynamic prefetcher for oblivious ram," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, 2015.

[38] X. Zhang, G. Sun, P. Xie, C. Zhang, Y. Liu, L. Wei, Q. Xu, and C. J. Xue, "Shadow block: Accelerating oram accesses with data duplication," in *51st Annual IEEE/ACM International Symposium on Microarchitecture*, 2018.

[39] X. Zhang, G. Sun, C. Zhang, W. Zhang, Y. Liang, T. Wang, Y. Chen, and J. Di, "Fork path: Improving efficiency of oram by removing redundant memory accesses," in *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture*, 2015.

[40] X. Zhuang, T. Zhang, and S. Pande, "Hide: An infrastructure for efficiently protecting information leakage on the address bus," in *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2004.