2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA) | 978-1-6654-7652-2/23/$31.00 ©2023 IEEE | DOI: 10.1109/HPCA56546.2023.10071064

AB-ORAM: Constructing Adjustable Buckets for Space Reduction in Ring ORAM

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

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

Xulong Tang
Computer Science Department
University of Pittsburgh
Pittsburgh, PA, USA
xulongtang@pitt.edu

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

***Abstract*—Ring ORAM (Oblivious RAM) is a secure primitive that mitigates the large performance degradation of ORAM through reduced *online* memory bandwidth demand, i.e., the number of memory accesses at servicing a real memory request. Ring ORAM requires 4× or more of the protected data space to enable the optimization and thus presents high capacity pressure on modern memory systems. While recent studies strive to reduce its space consumption through bucket compaction, the large space consumption remains a major design challenge for Ring ORAM.**

In this paper, we propose AB-ORAM to reduce the space capacity demand in Ring ORAM. AB-ORAM identifies two inefficient use of memory space in Ring ORAM: (i) accessed blocks hold useless data until the next reshuffle operation; and (ii) large buckets provide a diminishing performance benefit for tree levels close to the leaves. AB-ORAM then proposes two schemes to exploit the optimization opportunities, respectively. Specifically, it reclaims accessed blocks early by allocating them to buckets that need a reshuffle; and shrinks the bucket size for tree level close to the leaves for a better space/performance trade-off. We evaluate the proposed AB-ORAM design and compare it to the state-of-the-art. Our results show that AB-ORAM achieves an average of 36% space reduction over the state-of-the-art while introducing very low performance overhead.

***Index Terms*—Oblivious RAM, ORAM, security, memory access pattern, space efficiency.**

I. Introduction

Modern computer systems widely adopt the detached-memory architecture, i.e., the processor chip integrates a memory controller on-chip and sends memory addresses and device commands in cleartext on memory buses [19]. Even if the user data may be secured with strong encryption and authentication schemes, e.g., AES encryption [12], Merkle tree authentication [15], it is possible to leak sensitive information from access patterns in memory addresses [35], [39]. To ensure high-level protection of user privacy, it is necessary to adopt an expensive ORAM primitive that obfuscates memory requests from the user program [16], [17].

Ring ORAM is a recently proposed secure primitive for mitigating the large performance degradation of ORAM [25]. Ring ORAM is built on top of Path ORAM [28], an ORAM primitive that organizes data blocks in a binary tree structure and converts each user memory request to two path accesses in the tree, which incurs large performance degradation, i.e., O(logN) complexity where N is the number of to-be-protected data blocks. Ring ORAM optimizes the protocol by differentiating two types of accesses: online and offline accesses. The former refers to those servicing the real user requests while the latter refers to those for protocol maintenance. While Ring ORAM has the same overall complexity as that of Path ORAM, i.e., O(logN), it fetches one data block from each tree bucket for online accesses, representing $\frac{1}{Z}$ memory bandwidth requirement over Path ORAM, where Z is the number of data blocks in each tree node. With special hardware support, the memory bandwidth requirement for online accesses can be further reduced to O(1).

A major concern of Ring ORAM is its low space utilization. Path ORAM needs to double the memory space such that there are sufficient empty slots spreading across the ORAM tree and it has low possibility to remap a data block to a path with no empty slot [28]. This leads to 50% **space utilization**. The space utilization is defined as the size of the real user data over the size of the ORAM tree. Ring ORAM has even lower space utilization as it allocates more dummy blocks. For a typical setting [25], a 12-entry tree bucket keeps (1) five blocks for block remapping. Based on the above discussion, on average, there are 2.5 blocks holding real data while the rest holds dummy data; and (2) seven dummy blocks for Ring ORAM operations. This represents a 2.5/12= 21% space utilization.

Given memory resource is precious for modern computers, the low space utilization tends to introduce large performance and energy consumption overheads. Cao *et. al.* addressed the space utilization issue with *bucket compaction (CB)*, a design that shrinks the bucket size and utilizes a portion of real blocks as reserved dummies when needed [6]. CB prevents the stash overflow possibility with more frequent background eviction, a performance/overhead trade-off optimization proposed for Path ORAM [26]. Unfortunately, space utilization with bucket compaction may be improved to 31%, which remains a major design obstacle for Ring ORAM adoption. For this matter, some studies try to improve the performance of Path ORAM without increasing the space demand. For instance, IR-ORAM [23] is the latest optimization proposed upon Path ORAM that improves the performance while it maintains 50% space utilization. However, it still has lower performance than Ring ORAM. In this work, our goal is to achieve optimum space

utilization and performance simultaneously.

In this paper, we propose AB-ORAM to improve the space utilization of Ring ORAM implementation while maintaining the high performance benefit of Ring ORAM. The proposed AB-ORAM focuses on **space reduction**. We define space reduction as the reduced total size of the ORAM tree. Note that this reduction only affects the dummy part of the ORAM tree, and the real part, i.e., user data, remains intact. As such, AB-ORAM also effectively improves the space utilization (= user data/ ORAM tree size) as the ORAM tree size is reduced. The contributions of AB-ORAM are summarized as follows.

- AB-ORAM exploits two inefficient use of memory space in Ring ORAM: (i) a data block becomes a *dead block* after its first access and holds useless data till the next bucket reshuffle or path eviction; (ii) larger buckets that contain more dummy blocks help to support more path accesses till the next expensive bucket reshuffle or path eviction. However, for tree levels close to the leaves, its performance benefit diminishes fast while space demand increases dramatically.
- AB-ORAM addresses the inefficient use of memory space with optimized bucket allocation. AB-ORAM dynamically tracks dead blocks and adaptively allocates them to buckets that demand reshuffle, i.e., those due to bucket reshuffle or path eviction operations. This helps to reclaim dead blocks early and thus reduces the overall space demand. By exploiting the space/performance trade-off at different levels, AB-ORAM adopts a statically fixed but non-uniform bucket space allocation strategy. It decreases the number of dummy blocks for the levels close to the leaves.
- We evaluate the proposed AB-ORAM design and compare it to the state-of-the-art. Our results show that AB-ORAM achieves on average 36% space reduction over the state-of-the-art while introducing very low performance overhead.

II. Threat Model

We use the same threat model as those in prior ORAM studies [25], [28]. Our discussion is based on a standalone secure processor while the design is applicable to cloud setting with a secure server and remote clients. For the standalone secure processor, we assume that only the processor can be fully trustworthy, i.e., the trusted computing base (TCB) includes the processor only. The program code and data are stored in ciphertext in memory. An on-chip secure engine encrypts data before writing to memory and decrypts after fetching from memory. The data are also authenticated to ensure data integrity. Prior studies have shown that hardware-assisted security enhancements can effectively reduce the encryption and authentication overheads [15], [29], [32]. To prevent memory traces from leaking sensitive information, the baseline configuration adopts Ring ORAM.

III. Background

While Ring ORAM and Path ORAM share the same protocol complexity, Ring ORAM reduces the memory bandwidth for online accesses and thus has better performance.

A. Path ORAM Basics

Given Ring ORAM is built on top of Path ORAM, we next briefly discuss how Path ORAM works. More details can be found in [28]. Path ORAM achieves O(logN) protocol complexity by organizing the to-be-protected memory space as a binary tree. Each tree node, referred to as a bucket, contains Z' slots each of which holds a block of typically cacheline size. Each slot may contain either real (user data) or dummy block. Each user data block is randomly mapped to a path on the ORAM tree. Given a memory request for address A from the user program, Path ORAM translates A to its obfuscated path ID l, and services the request with two path accesses of l: read path phase and write path phase. Path ORAM accesses $L \times Z'$ blocks in each phase, where L is the number of ORAM tree levels. Path ORAM includes an on-chip ORAM controller that contains a *stash* and a *position map*. The stash buffers blocks read from the tree during path accesses while the position map holds frequently used mapping between data blocks and tree path IDs.

Path ORAM access has three types of operations. Assume we are to access block A.

(1) **read path:** The ORAM controller first looks up the position map to identify the path l on which block A resides, and then reads all the blocks on l from the memory. It decrypts and authenticates the fetched blocks. It keeps the real blocks in the stash and discards the dummy blocks.

(2) **block remap:** After the read path, block A is present in the stash. The ORAM controller remaps it to a random new path. It then updates the position map with this new mapping. Block A is then sent to the user program.

(3) **write path:** The ORAM controller writes data blocks except A back to l. It searches the entire stash and starts the write-back from the leaf level to the root. Dummy blocks might be written to a tree node if there are not enough blocks found for that node. Note that all blocks are encrypted and authenticated prior to write-back.

The protocol fails if a data block is mapped to a path that has no empty slot. To prevent protocol failures, Path ORAM uses only half of the slots to store real data blocks. Thus, the space utilization for Path ORAM is 50% [25], [26], [28].

B. Ring ORAM Basics

Ring ORAM [25] was developed on top of Path ORAM [28]. It achieves performance improvement by reducing the memory bandwidth requirement for online accesses to $\frac{1}{Z'}$ of that in Path ORAM. An *online access* is referred to as the operation that services the memory request of the user program.

Ring ORAM also organizes the to-be-protected memory as a binary tree. Assume we construct an ORAM tree with L levels and each bucket has Z slots. Then, the ORAM tree can hold $Z \times (2^L - 1)$ blocks. Ring ORAM reserves S slots in each bucket, (or $S \times (2^L - 1)$ for the entire tree), for holding dummy blocks only. The remaining Z' slots in each bucket may hold either real data blocks or dummy blocks (Z=Z'+S). Ring ORAM uses around half of Z' space for real data blocks

TABLE I: Organization of bucket metadata in Ring ORAM and AB-ORAM.

	Metadata Field	**AB-ORAM** (bit)	**Ring ORAM** (bit)	**Function**
Block-related	`count`	$1 \times log(S)$	$1 \times log(S)$	Number of times the bucket has been touched since the last refresh
	`addr`	$Z' \times log(N_{Block})$	$Z' \times log(N_{Block})$	Address for each real block
	`label`	$Z' \times (L+1)$	$Z' \times (L+1)$	The path ID of each real block
	`ptr`	$Z' \times log(Z)$	$Z' \times log(Z)$	Offset in the bucket for each real block
	`valid`	$Z \times 1$	$Z \times 1$	Indicates whether the corresponding block is valid
	`remote`	$R \times 1$	-	Indicates whether the corresponding block is located at a remote location
	`remoteAddr`	$R \times log(N_{Bucket})$	-	Address of the bucket in which the corresponding block is remotely allocated
	`remoteInd`	$R \times log(Z)$	-	Offset in the bucket of the remotely allocated block
	`dynamicS`	$log(S)$	-	The current S value of the bucket (based on the last allocation)
Slot-related	`status`	$Z \times 2$	-	Indicates the slot status (`REFRESHED, ALLOCATED, DEAD`)

to facilitate random path remapping, similar as that in Path ORAM. Fig. 1 depicts Ring ORAM tree organization.

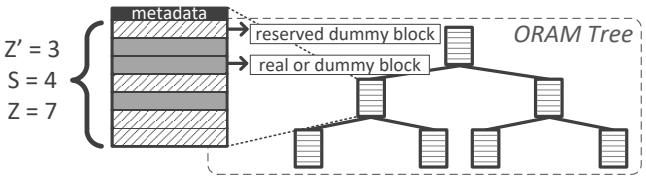


Fig. 1: Ring ORAM tree organization ($L = 3$, $Z' = 3$, $S = 4$, and $Z = 7$).

Like Path ORAM, Ring ORAM maintains a position map that maps each data block to a path ID. It also includes a stash to buffer data blocks loaded from the memory, and optionally a treetop cache for performance improvement [21], [24]. The path access in Path ORAM, is responsible for servicing the user request, maintaining the tree, and depleting the stash. Ring ORAM, on the other hand, differentiates between online and offline accesses. The former is to service the user program request while the latter refers to maintenance operations. Ring ORAM supports three main operations as follows.

- ***ReadPath:*** this operation is referred to as *online access*, i.e. to service the user program request. To access block A, the ORAM controller first determines its mapped path l and then conducts a two-step access.
  (1) metadata access: The ORAM controller loads the metadata from a separate small tree for all buckets along l. It then identifies the location of block A, i.e., a particular slot in one bucket, and then determines one valid dummy block from each of the other buckets along l. Metadata is updated/written back at the end of the Ring ORAM access.
  (2) block access: The ORAM controller reads one block from each bucket along the path. The block A is added to the stash while all other blocks are dummy blocks and thus discarded. The bucket location of each block is invalidated and the information gets updated in the metadata.
  ReadPath differs from path accesses in Path ORAM in that it only reads one block per bucket, thereby reducing the memory bandwidth requirement compared to Path ORAM.
- ***EvictPath:*** it is a background operation that gets triggered after every A online accesses. Each trigger chooses a path using reverse-lexicographic order to reshuffle. It i) reads all remaining valid blocks from the buckets along the selected path, ii) refills the buckets with loaded blocks as well as those in the stash, and iii) writes the new contents to the buckets in memory. The data are encrypted and authenticated and, as part of the operation, the metadata are also updated. The role of this operation is to lower the stash occupancy and push the data blocks to levels close to tree leaves. It is similar to path access in Path ORAM but requires no specific block to access.
- ***EarlyReshuffle:*** it gets triggered after a *readPath* operation, for a particular bucket if it accumulates S *readPath* operations after the last bucket write. To complete the operation, the ORAM controller reads the corresponding bucket into the stash, reshuffles and writes it back to the tree. Each bucket reshuffle includes Z' reads (from valid slots) and Z writes (to all slots). Note that a residue block in the stash might be piggy-backed during a bucket reshuffle.

Note that *readPath* is considered online access, and services the user program request. Whereas, *evictPath*, and *earlyReshuffle* are offline accesses and responsible for maintaining the tree and depleting the stash.

Ring ORAM maintains metadata for each bucket, as listed in Table I, to facilitate the protocol operation. When accessing a bucket with *readPath*, we increment its `count` and invalidate the corresponding slot. The `addr` and `ptr` fields determine at what slot each real block resides in the bucket.

Ren *et al.* explored the design space for choosing Z', S, Z, and A [25]. For a typical setting for secure processor setting, we use $Z' = 5$, $S = 7$, $Z = 12$, $A = 5$, as shown in [25].

Space utilization. For Ring ORAM, its space utilization is $(Z' \times 50\%)/Z$, or about 21% for the above typical setting.

C. Ring ORAM with Bucket Compaction

Cao *et. al.* proposed to shrink the bucket size while keeping Z' and S parameters intact by introducing the concept of overlap [6]. In this scheme, the bucket size is Z, and Z' blocks are dedicated to real blocks. Then, S can have a value of $Z - Z' + Y$, where Y is the number of blocks for overlap. In this way, when all dummy reserved blocks of a bucket are used, a block from the portion dedicated to real blocks can be returned to the processor. That block is called a green block and may be either dummy or real. In case it is real, it has to remain in the stash. This can increase the chance of stash overflow. To address this issue, they proposed to generate dummy accesses if the stash occupancy reaches a threshold.

Thus, dummy insertion continues until *evictPath* operation frees up the stash below the threshold. If we consider the typical setting of Ring ORAM, $Z' = 5$, $S = 7$, $Z = 12$ as a baseline, by applying the bucket compaction scheme with $Y = 4$, the allocation will be $Z = 8$, $Z' = 5$ and $S = 3$. In this paper, we built our design on top of this state-of-the-art.

D. Broad Impact of Space Reduction

Studies have shown that securing memory access patterns demands ORAM [16], [17] as simple obfuscation [39] tends to provide limited security protection. Unfortunately, all ORAM protocols have high performance overheads, in particular, they introduce orders of magnitude more memory accesses. Of different ORAM protocols, Ring ORAM has low online memory bandwidth request, making it a promising protocol for practical deployment.

However, Ring ORAM's space utilization is low, which increases memory contention with co-running applications. As main memory is one of the most precious system resources in modern systems, reducing space demand can effectively make better use of main memory resource, leading to improved system performance, throughput, and power/energy consumption.

IV. Motivation

We made two key observations regarding the low utilization of memory space in Ring ORAM. In the following, we elaborate these observations and discuss how to exploit them to improve the space efficiency. The first observation is the existence of dead blocks in the ORAM tree. The second observation is that there exists a space/performance trade-off on the bucket size at different tree levels.

A. Studying Dead Blocks

A bucket slot in Ring ORAM tree can be accessed at most once between any two reshuffles. The ORAM controller marks the slot as *invalid* after its *readPath* but does not reclaim the space until a later *evictPath* or *earlyReshuffle*. In this paper, the invalid bucket slots are referred to as *dead blocks*.

We conduct an experiment to track the total number of dead blocks for different benchmarks and summarize the results in Fig. 2. The settings are listed in Section VII. The X-axis shows the program execution in the total number of online accesses. The Y-axis shows the snapshot of the total number of dead blocks. From the figure, the number of dead blocks increases quickly at the beginning of the execution and stabilizes after 30M online accesses. Due to high similarity of the results, the figure only reports the results from three individual benchmarks and the average of all benchmarks.

Dead blocks are generated from online accesses at a stable rate, i.e., every *readPath* generates L dead blocks, where L is the tree height. The elimination of dead blocks, i.e., reclaiming invalid blocks through *evictPath* and *earlyReshuffle* operations, exhibits low rate at the beginning of the execution, and then stabilizes for the rest of program execution. *EarlyReshuffle* always gets triggered when a bucket accumulates S dead

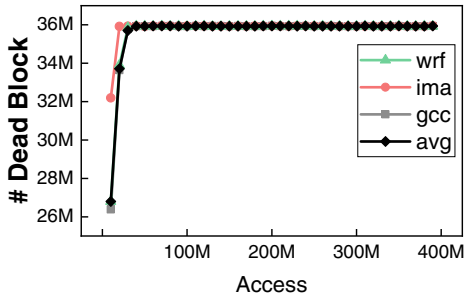


Fig. 2: Dead blocks over time for different benchmarks.

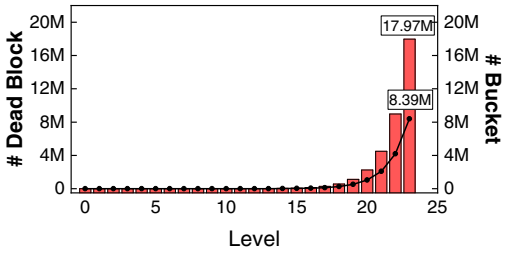


Fig. 3: Dead blocks across the levels.

blocks. The chance is low at the beginning of the execution. For the path determined by *evictPath*, there are few dead blocks along the path at the beginning of the execution so that only few dead blocks can be reclaimed. With more online accesses, the dead blocks spread across all paths such that picking up any path has around L dead blocks to reclaim. This helps to stabilize the total number dead blocks.

For the 24-level ORAM tree in Fig. 2, the dead blocks account for around 18% (=36M/(12$\times$(2^{24}-1)) of the total ORAM space. That is, around 18% of the allocated space is wasted at any time after entering the stable execution stage.

Fig. 3 reports the numbers of existing dead blocks at different tree levels after running 400 million traces. At each tree level (X-axis), the bar shows the number of dead blocks (Y-axis to the left) and the line dot denotes the number of buckets at that level (Y-axis to the right). As shown in the figure, the last level contains 17.9 million dead blocks. Given that the last level has about 8 million buckets, on average, there are 2.1 dead blocks per bucket.

B. Studying Space/Performance Trade-off

In this section, we study the trade-off between space and performance in Ring ORAM. As discussed before, there are S reserved dummy blocks allocated for each bucket in Ring ORAM. During the *readPath*, only one bucket returns a real block while all other buckets along with the path return dummy blocks. Any bucket, if having been accessed S times, may run out of dummy blocks and thus needs to be reshuffled, i.e., triggering an *earlyReshuffle* on this bucket. Thus, the larger the S value is, the less number of *earlyReshuffle* operations the Ring ORAM would need. Of course, a larger S value results in more space wasted in saving dummy data. In addition, a larger S value leads to more blocks in each path,

making it more expensive to complete *evictPath*. Accordingly, if we reduce the S value, the number of *earlyReshuffles* increases but at the same time the cost of *evictPaths* decreases.

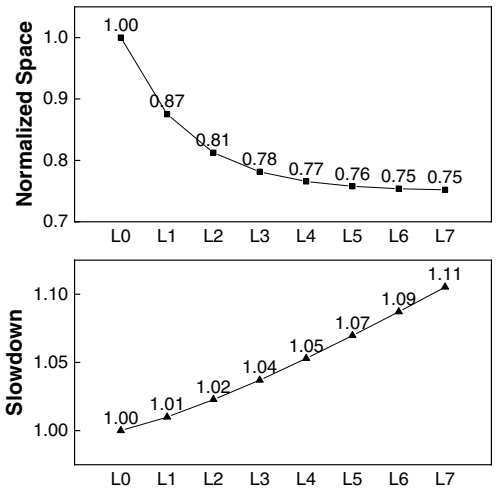


Fig. 4: Space demand across different path length normalized to the baseline (top), slowdown over the baseline (bottom).

We conduct an experiment to expose the space/performance trade-off. In Fig. 4, we compare the space and performance impacts when reducing the S values for the bottom seven levels (that account for 99% of the entire capacity). L-x means reducing the S value by three for the last x levels. The baseline adopts the typical setting [25], $Z = 12$, $Z' = 5$, and $S = 7$. From the figure, as we reduce the S values for more levels, the space demand reduces while the performance degrades. When the bucket size is reduced so is the path length, hence, the path eviction incurs less number of memory accesses. The space saving stabilizes after reducing the last three levels. In the experiment, there is a large increase of the number of *earlyReshuffles*. However, the reduction from conducting cheaper *evictPaths* benefits more. As figure shows, the execution time grows linearly whereas the space reduction is in logarithmic scale. Hence, by shrinking the S value for the levels close to the leaves, we can achieve a significant space reduction while incurring a low performance overhead.

V. The AB-ORAM Design

A. Overview

In this section, we elaborate on our AB-ORAM design for reducing the space demand for Ring ORAM. AB-ORAM consists of two designs — one is to early reclaim the dead blocks while the other is to reduce the number of dummy blocks for bottom levels with a non-uniform S value setting.

Dead Block Reclaim. To reclaim dead blocks, we adopt a new block allocation mechanism called remote allocation. To enable remote allocation, we first construct a FIFO queue to track recently generated dead blocks at each bottom tree level and update the metadata accordingly. We then reduce the initial S value, i.e., the number of reserved dummy blocks, for the buckets at bottom levels and extend the S value to its original using the space reclaimed from dead blocks. Given a binary tree doubles its size with every extra level, reducing the S value at the bottom levels can effectively reduce the space demand for Ring ORAM.

Non-uniform S Setting. By exploiting the performance and space-saving trade-off at the bottom levels, we propose to reduce the S value for bottom tree levels. While such a design increases the number of *earlyReshuffle* operations and slightly degrades the performance, it achieves large space savings.

For the first scheme, we need remote allocation and altering S value. While the second scheme only involves altering S value. Thus, we organize the rest of this section as follows. First, we discuss the remote allocation. Then, we discuss how we can exploit altering S value for each of our schemes.

B. Remote Allocation

1) One Extra Level of Address Mapping: Ring ORAM consists of three levels of address mapping, as shown in Fig. 5(a). A memory address of a user request is first mapped to a path ID in the ORAM tree. This mapping is randomized and secured by the ORAM protocol, i.e, encrypted as position map. Given a path ID, we can use the well-known tree organization knowledge to translate it into a list of tree buckets. Depending on the operation, we may get all block addresses for the selected buckets (for *evictPath* and *earlyReshuffle*) or one block address per bucket (for *readPath*). The latter is determined by the metadata, which is also secured and part of the Ring ORAM protocol. Given a tree block address, the OS translates it to the storage location, i.e., the physical addresses in the main memory. From the figure, mapping from a path ID to all its related tree buckets/blocks, and from a tree block to the address in the memory is not protected and is thus known to the attackers.

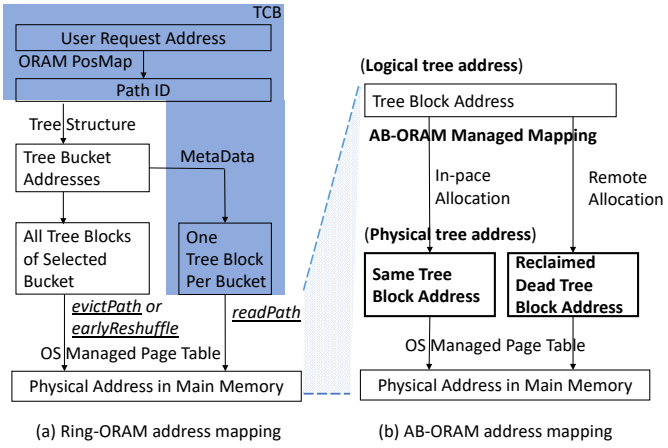


Fig. 5: AB-ORAM adds one more level of address mapping in the insecure domain.

To reuse the dead blocks, AB-ORAM introduces one more level of address mapping, as shown in Fig. 5(b). The translation from a memory address of a user request to a list of tree block addresses is kept the same as that in the

baseline. To facilitate the discussion, we differentiate two tree block addresses — *logical tree addresses* and *physical tree addresses*. Conceptually, the logical tree address of a data block determines where the block should stay according to tree organization. The physical tree address determines its actual location due to space availability. AB-ORAM saves this mapping in bucket metadata, however, the mapping is kept in cleartext and thus known to the attackers.

Based on if a block's physical tree address is the same as its logical tree address, we have two types of data allocation.

- In-place allocation: This refers to the case when a block's physical tree address is the same as its logical tree address. For example, all blocks in the baseline implementation. In AB-ORAM, the block in the top and middle levels keeps the same logical and physical tree addresses.
- Remote allocation: This refers to the case when a block's physical tree address differs from its logical tree address. After fetching a tree block for *readPath*, AB-ORAM marks the corresponding block as dead and may reclaim it by allocating it to a different logical tree block. In Ring ORAM, both *evictPath* and *earlyReshuffle* need to write reshuffled secure data back to the memory. They may demand space allocation and set up the corresponding mapping for remote allocation. The mapping is kept in the metadata in cleartext.

2) Tracking Dead Blocks: To enable remote allocation, we need to dynamically identify the dead blocks in the ORAM tree so that we can reuse them later. This consists of two subtasks. One is to collect the addresses of dead blocks while the other is to mark the status of tree blocks in metadata.

Tracking Queues. In Ring ORAM, dead blocks are generated from *readPath* operations — each *readPath* fetches L blocks such that all of them become *dead* after the access. While all L blocks can be potentially claimed, our study in Section VIII-D reveals that the dead blocks in the top and middle levels tend to have short lifetimes and they account for a small portion of the total memory space. As such, we skip these levels and maintain several FIFO queues with one for each bottom level, referred to as *DeadQ* queues, to track dead blocks at each corresponding level. Each entry in the queue maintains two fields that define the physical location of a dead block (or an empty slot): {`slotAddr`, `slotInd`}.

AB-ORAM maintains all the *DeadQ* queues inside the processor, i.e., on-chip. However, this information does not need to be secured. Our security analysis shall prove that it does not compromise security protection if the attacker knows this information. Given the limited on-chip space, each *DeadQ* maintains a small number of entries, e.g., 1000 entries. Since they are FIFO queues, the maintenance cost is low. The design goal of the *DeadQ* is not to track all dead blocks at each level. Instead, it is to collect a good amount of dead blocks so that we can exploit dead blocks to meet part of the space allocation demands for the following *evictPath* and *earlyReshuffle* operations. We keep one queue for each level because according to our analysis the lifetime of dead blocks from different levels exhibit orders of magnitude difference as shown in Fig. 12.

Tracking Metadata. Since we add one level of address mapping, we need to precisely know if a tree block adopts in-place allocation or remote allocation. AB-ORAM achieves this by keeping per block status information in the metadata of each bucket. Table I details the organization of bucket metadata in Ring ORAM and AB-ORAM. For clarity, we divide metadata fields into two categories, *block-related* and *slot-related*. The block-related metadata of a bucket contains information about the blocks that have been mapped to this bucket. Whereas the slot-related metadata indicates information about the slot itself, i.e. the physical location.

AB-ORAM adds five pieces of metadata to Ring ORAM: four block-related (`remote`, `remoteAddr`, `remoteInd`, and `dynamicS`) and one slot-related (`status`). The `remote` flag indicates whether the corresponding block adopts in-place or remote allocation, i.e., if the logical and physical locations of the block are the same. In the case that they are not the same, `remoteAddr` and `remoteInd` identify the physical location of the block in the tree.

The `status` of all slots is initially `REFRESHED` once they are written to the ORAM tree. When a block is accessed during the *readPath*, it must be invalidated according to the Ring ORAM protocol. Thus, its `valid` flag is turned off. At this point, the `status` of the slot that contains this block becomes `DEAD` since it is carrying a dead block. A slot is marked as `ALLOCATED` when it is added to AB-ORAM's *DeadQ*.

Table I states the size of metadata fields in terms of ORAM parameter. Note that N_{Block} and N_{Bucket} are the number of real blocks and the total number of buckets in the ORAM tree, respectively. R indicates the maximum number of slots that AB-ORAM allows remote allocation per bucket. All other parameters are identical to what is introduced in Section III-B.

Tracking procedures. In summary, we have the following two lightweight procedures.

- `markDEAD()`: it marks the `status` of a slot as `DEAD` when its occupant block turns dead (i.e. `valid` = 0). Note that the block may be either in-place-allocated or remote-allocated. It is invoked at the metadata access of *readPath*.
- `gatherDEADs()`: it adds information of all the `DEAD` slots along a path into the corresponding *DeadQ* with the format of {`slotAddr` , `slotInd`}. Then, it marks `status` of that slot as `ALLOCATED` so that no one else will use it. It is invoked at the metadata access of *readPath*.

3) Remote Allocation: In Ring ORAM, only *evictPath* and *earlyReshuffle* can write data blocks to the main memory. All data blocks take in-place allocation in the baseline Ring ORAM implementation.

Next, we illustrate how to remotely allocate a data block (with logical tree address A) at a bottom tree level.

(1). We first dequeue a *DEAD* block from its corresponding *DeadQ*. Let us assume the tree block address of it is T.

(2). We then write block A in block T. Note that no access or update on metadata for block T is needed. Since block T is looked up from the *DeadQ*, its `status` has already been updated as `ALLOCATED` (at the time it was queued).

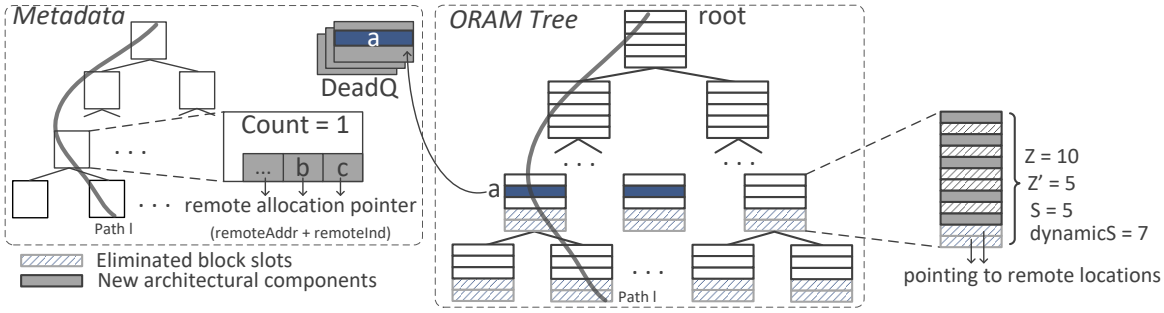


Fig. 6: An overview of remote allocation in AB-ORAM.

(3). We update the block-related metadata of block A. The `remote` flag is raised and `remoteAddr` and `remoteInd` are set to point to block T. All other block-related metadata pieces are updated as they are in the Ring ORAM.

Fig. 6, demonstrates the remote allocation. In the figure, after accessing block a on path l, it is marked as dead and added to the *DeadQ*. Block b and c are remotely allocated and the mapping identifies the physical tree addresses.

C. Altering the S Value for Space Savings

We next elaborate on how to change the S value for better space savings. It consists of two designs. One is to extend the S value to take advantage of the remote allocation. The other is to set the non-uniform S values for more space savings.

1) Extending the S Value with Remote Allocation: While remote allocation can early reclaim the space occupied by dead blocks, the baseline tree allocation actually cannot exploit its benefit — in the baseline, every logical tree block has its physical tree block allocated so that there is no need to "reuse" the space of dead blocks.

There exist two alternative strategies to exploit remote allocation. Based on the typical ORAM setting $Z' = 5$, $S = 7$, $Z = 12$, $A = 5$, assume we apply remote allocation to the bottom levels, taking level 23 as an example.

(1). We allocate 12 blocks ($Z = 5 + 7 = 12$, $S = 7$) for each bucket at this level. At runtime, based on the generation rate of dead blocks, we can extend the bucket size to $Z = 5 + 9 = 14$, $S = 9$, i.e., each bucket can sustain 9 *readPath* accesses, instead of 7 *readPath* accesses, before triggering an *earlyReshuffle*.

(2). We allocate 10 blocks ($Z = 5 + 5 = 10$, $S = 5$) for each bucket at this level. At runtime, we recover the bucket size to $Z = 5 + 7 = 12$, $S = 7$. Each bucket can still sustain 7 *readPath* accesses, even though we physically only allocate 5 reserved dummy blocks.

For either strategy, after extending the S value, each bucket has two fewer physical blocks, e.g., strategy (1) allocates 12 entries for one bucket but tries to use it as a 14-entry bucket. We, therefore, allocate two logical blocks to the reclaimed dead blocks. The two strategies have different emphases — the first strategy saves no space comparing the baseline. However, it reduces the number of *earlyReshuffle* operations and thus potentially improves the performance. The second strategy focuses on space savings. It uses smaller space for the initial tree allocation but tries to achieve the same effectiveness as the baseline. Given this paper focuses on space savings, we adopt the second strategy in AB-ORAM.

AB-ORAM initializes the ORAM tree with the bucket size for bottom tree levels being set as $Z - r$ where r denotes the applied space reduction. Based on the study in Section IV, we identify r as 2 for the baseline setting. The r value depends on the choices of Z, Z', and S values, i.e, it depends on the ORAM tree, while being independent of the secure applications. After one complete round of tree reshuffle, i.e, applying *evictPath* to every path, we start to extend S values for the bottom levels. The number of sustained *readPath* accesses before *earlyReshuffle* is also updated to the new value.

In the design, when the *DeadQ* is empty, we may skip extending the S value for a bucket at the bottom level. This may happen either at *evictPath* or *earlyReshuffle* time. To facilitate the transition and the empty-queue case, we keep a counter `dynamicS` for each bucket, as in Table I. It tracks the number of *readPath* that it can sustain, `dynamicS` is extended to $S + 2$ only for the buckets that allocate their two logical tree blocks in reclaimed dead blocks.

2) The Non-uniform S Value: The motivational study in Section IV revealed that, if we choose to reduce the S value for a number of bottom tree levels, there exists a trade-off between performance overhead and space savings. Therefore, we can take a non-uniform S value design that sets S_1 and S ($0 \leq S_1 < S$) values for the bottom k tree levels and for the other tree levels, respectively. Here S is the same as the baseline while S_1 is a smaller value. By reducing the S values for k bottom levels, we tend to suffer from a small performance degradation but achieve space savings.

Fig. 4 in Section IV shows that the space savings are about to saturate at $L3$ while the performance loss is low (about 4%). The result was for $S_1 = S - 3$. However, if we take a different S_1 value, the trend of space savings stay the same while the performance loss may vary. We, therefore, set $k = 3$ or $k = 2$ and adjust S_1 according to the space utilization. Our experiment study shows that S_1 can be set as S-2 and S-1 for the baseline Ring ORAM and an optimized implementation that makes better use of space, respectively.

We propose two designs in the paper — (1) Initially allocating smaller buckets (with smaller S value) for several bottom tree levels and extending the S value at runtime; (2) Assigning a smaller S value for several bottom tree levels. While both designs shrink the S value, they differ from each other as they have different design goals. The former allocates fewer physical tree blocks for these buckets and exploits dead blocks to mitigate the physical space reduction, i.e., recover to the same S value and sustain the same number of *readPath* accesses as those of the baseline. The performance overhead comes from accessing remote blocks for a subset of tree entries. The latter permanently reduces the S value for the bottom tree levels such that these buckets can sustain fewer *readPath* accesses. The performance overhead comes from increased *earlyReshuffle* operations. By shrinking the S value, the latter design improves the *evictPath* performance.

D. Comparison to the State-of-the-arts

Table II summarizes the latest optimizations proposed on ORAM. One of the key improvements in IR-ORAM [23] is to reduce the path access overhead by shrinking the Z value for the middle levels. IR-ORAM reveals that middle levels are under-utilized. Besides, they account for a small portion of the entire capacity. Thus, shrinking the buckets of these levels improves the performance without affecting the capacity. However, it increases the probability of the stash overflow, hence, it may incur more background evictions than the baseline. IR-ORAM was proposed to optimize Path ORAM [28] while the principal can be adopted to optimize the portion of Z' entries of tree buckets in Ring ORAM.

Bucket Compaction (CB) [6] was proposed to shrink the bucket size for Ring ORAM by reducing the S value. This too reduces the path access overhead so that *evictPath* operation costs less. Unlike IR-ORAM, CB applies the shrinking to buckets of all levels so it reduces the space demand effectively. Note that this reduction only affects the number of reserved dummy blocks so the capacity of the tree for storing real data blocks remains intact. Like IR-ORAM, it may increase the number of background evictions because real blocks may be returned to the stash instead of dummy blocks.

As discussed, we develop two schemes in this paper, i) extending the S value for bottom levels by reclaiming dead blocks, and ii) setting non-uniform S values for the ORAM tree. For the former, we shrink the S value like CB but we compensate for the performance impact by extending the S value via remote allocation. Since remote allocation means address redirection, it may incur a slight increase in memory block accesses due to lower row buffer hit in DRAM DIMMs. Our experimental results show that this overhead is negligible. The latter design shrinks the bucket size of levels close to the leaves, which helps to reduce the overhead of each *evictPath* operation. This increases the number of *earlyReshuffle* operations for those levels. However, the reshuffle operations are off the critical path and thus exhibit a low impact on performance.

More importantly, the proposed two schemes are orthogonal to both IR-ORAM and bucket compaction. By adopting our schemes on top of IR-ORAM or CB, we are able to further reduce the space demand of Ring ORAM. The space utilization of the fully optimized Ring ORAM implementation is made comparable to that of Path ORAM, i.e., around 50%.

VI. Security and Correctness

In this section, we show that AB-ORAM ensures the same level of security guarantee as that of the baseline Ring ORAM.

A. Remote Allocation is Secure

To prove remote allocation is secure, we refer to the address mapping enhancement in Fig. 5 in Section V-B1 The key observation is that the added address mapping is outside of the secure domain, that is, we utilize public knowledge to improve space utilization and leak no secure information.

To utilize the space of dead blocks, AB-ORAM tracks the generation of dead blocks, early reclaims dead blocks, queries the status of selected blocks, and shrinks/extends the S value for the bottom tree levels. Tracking the generation of dead blocks leaks no secure information as it is public knowledge — attackers, without performing AB-ORAM, can conduct the same information collection, i.e., the blocks accessed by *readPath* become dead. We maintain a queue for each level and enqueue/dequeue in cleartext. Given a dead block, it is well-known if it is queued, or skipped as the queue is full. Collecting such information reveals no secure information.

Reclaiming dead blocks leaks no secure information as, to reclaim dead blocks, we dequeue them from *DeadQ* and exploit them as a buffer to store the data of corresponding logical tree blocks. The mapping is known to the public. Such an extra mapping is secure because if it is not, we can construct a simple attack to the baseline Ring ORAM. Conceptually, the decision on choosing a particular logical tree block from a bucket at *readPath* is secure. The address mapping introduced in AB-ORAM does not change the above decision and kicks in only when we know the address of a logical tree block.

Querying the status of data blocks is secure as the query is integrated with the metadata access in the baseline. The metadata access is performed before each *readPath* and thus AB-ORAM does not introduce extra protocol access steps.

In the same way that real and dummy blocks are indistinguishable in Ring ORAM, their dead and reused versions are indistinguishable in AB-ORAM. Therefore, an attacker cannot infer anything about a block being real or dummy by collecting a dictionary of all remote mappings in AB-ORAM. Also, the temporal locality would not affect this matter either because if it would, one could guess the type of the accessed block based on its location along the path in Ring ORAM.

B. Altering the S Value is Secure

According to the security analysis in the baseline Ring ORAM [25], the three types of operations can be divided into two groups: (1) user-application-dependent *readPath*; (2) maintenance-oriented *earlyReshuffle*, and *evictPath*. Ring ORAM ensures all *readPath* accesses are indistinguishable, while the knowledge about when to perform maintenance

TABLE II: Summary of the state-of-the-art ORAM implementations.

	Ring ORAM [25]	IR-ORAM [23]	Bucket Compaction [6]	This work	
				Dead block reclaim	Non-uniform S value
Space demand	-	improved	improved	improved	improved
Online access	-	-	-	slight more	-
Bucket reshuffle	-	-	-	slight more	more
Path eviction	-	-	improved	slight more	improved
Background eviction	-	more	more	-	-

operations is well-known and leaks no secure information. We next prove that, by altering the S value (in dead block reclaim and non-uniform S design), these operations remain secure.

For *readPath*, altering the S value has no impact on Ring ORAM mapping (i.e., creating/accessing PosMap entries). Consequently, all *readPaths* remain indistinguishable — each of them initiates a metadata access and then fetches one block from each bucket along the path. Each bucket contains at least one dummy slot to support one more *readPath*. Therefore, no secure information leaks from collecting *readPath* operations.

For *earlyReshuffle*, it is public knowledge about when to be triggered on a particular bucket. Altering S is advertised as cleartext `dynamicS` such that the same strategy is applied to trigger *earlyReshuffle*, which leaks no secure information.

For *evictPath*, it is triggered at a fixed interval, i.e., once for every five *readPath* accesses and uses the fixed reverse-lexicographic order, which leaks no secure information.

For the special case discussed in Section V-C1, i.e., when *DeadQ* is empty, we may skip extending the S value, i.e., allocating $Z = 5 + 5 = 10$ entries instead of $Z = 5 + 7 = 12$ entries for a bucket at the bottom level. This results in assigning `dynamicS`$= 5$ for this bucket. This is secure as this is public knowledge, similar to assigning a cleartext $S = 7$ in the baseline. The slight security difference here is, now an attacker guesses if any of the first 5 accesses to a 10-entry bucket contains real data (instead of guessing from 7 accesses to a 12-entry bucket). For the non-uniform S value design, the security indication is, assume we reduce S to 3, an attacker guesses if any of the first 3 accesses to an 8-entry bucket contains real data. Given this guess needs to be combined with all bucket accesses along the path, i.e., only one bucket returns the real data, this security difference is negligible as we demonstrate empirically in the following experiment.

C. Empirical Security Analysis

We set up an experiment to demonstrate how AB-ORAM preserves the same security guarantee as Ring ORAM. We simulate one billion traces of 17 benchmarks in this experiment. All other configurations are listed in Section VII. We measured the success rate of an attacker guessing the real block during *readPath*. The attacker guesses one block out of L blocks randomly. Fig. 7 indicates the success rate, i.e., the number of correct guesses by the attacker over the total number of *readPaths*. On average, the baseline exhibits a success rate of 0.041665, while for AB-ORAM, it is 0.041670. As shown in the figure, AB-ORAM closely follows the baseline. As one would expect, the success rate for all applications is around 0.041666 (= $1/24$), which highlights that path accesses are indistinguishable in Ring ORAM, which is preserved by AB-ORAM as well.

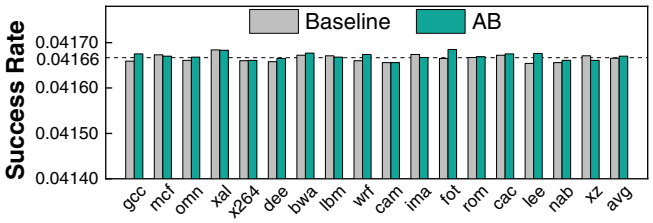


Fig. 7: Empirical study on AB-ORAM security implication.

D. Correctness

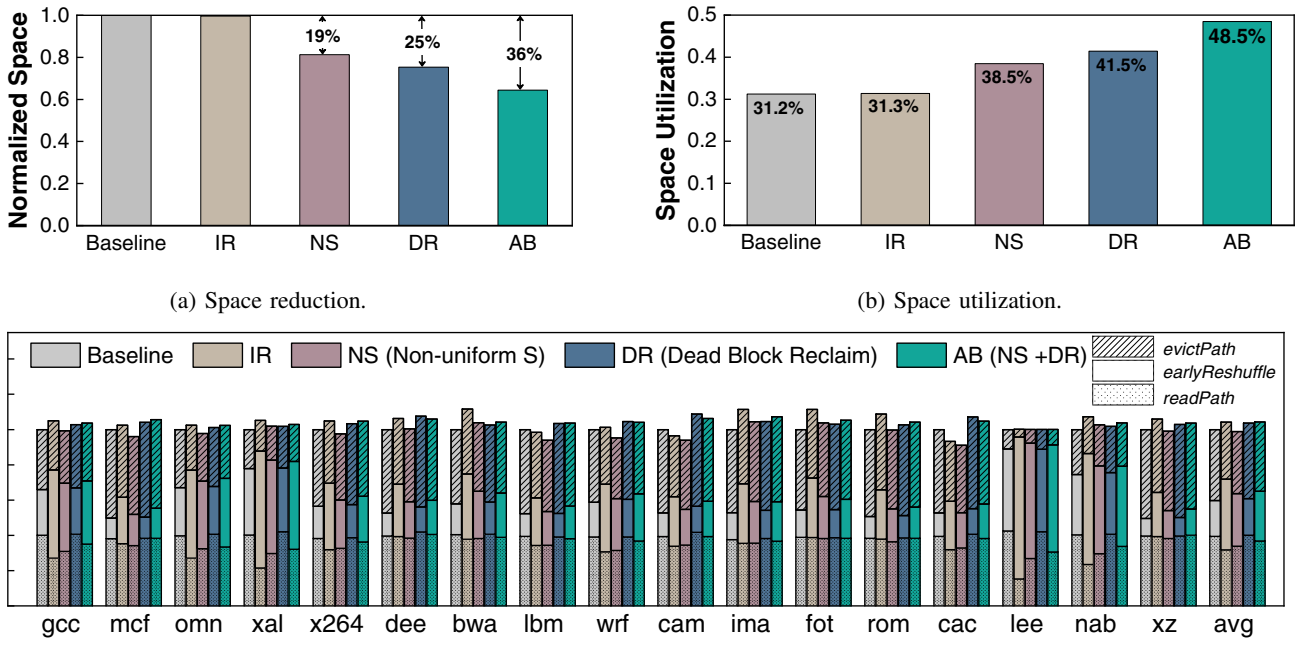
Ring ORAM follows Path ORAM and fails if a data block is mapped to a path that contains no dummy entry for the Z' portion. Given AB-ORAM does not change the Z' value for the ORAM tree, AB-ORAM can save the same number of data blocks in each bucket. The address mapping from the user address to the tree path remains the same. AB-ORAM fetches one real data block from one path, the same as the baseline. As such, AB-ORAM introduces no correctness issue.

VII. Experimental Methodology

To evaluate AB-ORAM, we used USIMM [7], a widely-adopted trace-driven cycle-accurate DRAM simulator in the literature to evaluate ORAM schemes. Table III lists the configuration details. We used the Pin tool [10] to collect traces from SPEC CPU2017 suite [1]. We used traces with 40 million memory accesses from each benchmark. For each trace, the first 38 million accesses were used to warm up the ORAM tree, and the last two million were fed to USIMM for DRAM access simulation. While not reported, we ran longer traces and the results remained stable. Table IV lists the benchmarks and their LLC misses per kilo instruction (MPKI) in the experiments.

We modeled a Ring ORAM tree with 24 levels, $Z = 12$, and $Z' = 5$, $S = 7$. We integrated Bucket Compaction [6] in the baseline, as set $Y = 4$, $Z = 8$, $Z' = 5$, and $S = 3$, i.e., the tree occupies $(2^{24} - 1)\times 8\times 64\text{B} = 8\text{GB}$ memory space. Following the prior work [6], [22], [23], [28], [36], [37], the protected user data occupies around 50% of all Z' entries in buckets, that is, $2^{(24-1)}\times 5\times 50\%\times 64\text{B} = 2.5\text{GB}$. We adopted a tree cache that saves the top 10 levels on-chip [23].

We implement and evaluate the following schemes.



(a) Space reduction.

(b) Space utilization.

(c) Performance overhead with the breakdown of operations.

Fig. 8: Space saving and performance overhead comparison of different schemes.

- `Baseline`: It implements Ring ORAM [25] and integrates Bucket Compaction [6], with $Y = 4$, $Z = 8$, $Z' = 5$, and $S = 3$. Note, all other schemes are built on top of this.
- `IR`: It implements IR-ORAM utilization optimization [23]. It sets $Z' = 4$ for levels in range [L10, L18], and $Y = 3$.
- `DR` (Dead Block Reclaim): It sets the bucket size to $Z = 6$ ($Z' = 5$, $S = 1$) for [L18, L23], then extends S value by 2 via remote allocation.
- `NS` (Non-uniform S): It sets the bucket size to $Z = 6$ ($Z' = 5$, $S = 1$) for [L22, L23].
- `AB`: It combines `DR` and `NS`. It sets $Z = 6$ ($Z' = 5$, $S = 1$) for [L18, L20], and $Z = 5$ ($Z' = 5$, $S = 0$) for [L21, L23].

TABLE III: System configuration.

Processor Configuration	
Processor Fetch Width/ ROB Size	4 / 256
Memory Channels	4
DRAM Clk Frequency	800 MHz
L1 / L2 D-cache	4-way 64KB / 8-way 256KB
L3 cache (LLC)	16-way 2MB
ORAM Configuration	
ORAM tree levels	24
Bucket size/ Block size	4 / 64B
Stash entries	300
Dedicated tree top cache	256KB (4K entries)
On-chip PLB / PosMap	64KB / 512KB

VIII. Experimental Results

In this section, we discuss the result of evaluated schemes described in Section VII.

TABLE IV: Evaluated benchmarks.

Integer Benchmark	read MPKI	write MPKI	Float Benchmark	read MPKI	write MPKI
gcc	0.1	0.5	bwa	0.0	4.1
mcf	28.2	0.2	lbm	0	15.3
omn	0.3	0.06	wrf	0.1	1.0
xal	0.1	0.2	cam	0.0	7.1
x264	1.6	2.1	ima	0.2	2.1
dee	0.0	14.7	fot	0.03	1.56
xz	0	15.5	rom	0.0	13.7
lee	0.01	0.01	nab	0.1	0.2
			cac	0.0	5.4

A. Main Results of Space and Performance

Fig. 8 presents the main result of our evaluation. Fig. 8a reports the total space consumption of different schemes normalized over `Baseline`. Fig. 8b shows the space utilization. Fig. 8c compares the normalized execution time for different schemes, with results normalized over `Baseline`.

From the figure, `IR` exhibits 4% performance slowdown because it uses $Y = 3$ to avoid a large increase of background evictions, but smaller Y causes more reshuffles than `Baseline`, which has $Y = 4$. It has a negligible impact on space demand as it only shrinks the middle levels of the ORAM tree. `DR` lowers the space demand to 75% of `Baseline`, i.e., the state-of-the-art Ring ORAM implementation. `DR` achieves 25% space reduction, leading to a space utilization of 41.5%. We track the reclaimed blocks and observe that `DR` early reclaims most of the dead blocks in the system. `DR` is 3% slower than `Baseline`, with the

overhead coming mainly from the enlarged buckets at the bottom levels. Reshuffling a bucket with the extended S value is more expensive than doing an original bucket. NS reduces the space demand of Baseline by 19% with comparable execution time. NS increases the number of *earlyReshuffles* but reduces the cost of *evictPath*. When combining DR and NS, AB-ORAM achieves 36% space reduction over Baseline while having around 4% performance overhead. The combined scheme achieves further space savings over DR and NS, indicating DR and NS are designs that improve space utilization from different directions. AB improves the space utilization of Baseline from 31.2% to 48.5% which is very close to 50%.

Note that the bucket size reduction in IR-ORAM [23] originally was proposed upon Path ORAM, and in that setting, it benefits the performance. Because bucket size reduction is more significant in Path ORAM as it has a much smaller bucket size (Z=4). In addition, IR in the presence of the bucket compaction optimization in Ring ORAM incurs more dummy accesses due to stash overflow.

Fig. 9 reports the bandwidth impact of our approach. From the figure, the extra bandwidth demand is negligible. On average, AB increases the bandwidth usage by 1%.

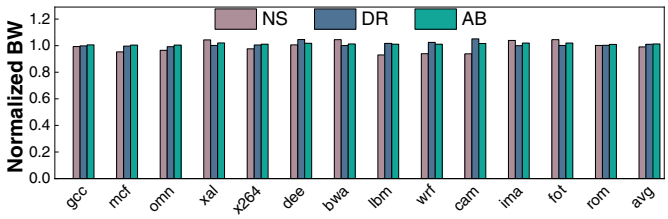


Fig. 9: Bandwidth impact of AB-ORAM.

B. Bucket Reshuffle Impact

Fig. 10 compares the number of reshuffles across the different levels for different schemes. DR has the closest number of reshuffles compared to Baseline due to S extension. NS increases the number of reshuffles for [L22, L23] where the S value is reduced by 2. AB compared to NS has more reshuffles for L21 and fewer reshuffles for L22 and L23. This is because we use L3-S1 in AB that shrinks S by 1 for [L21, L23].

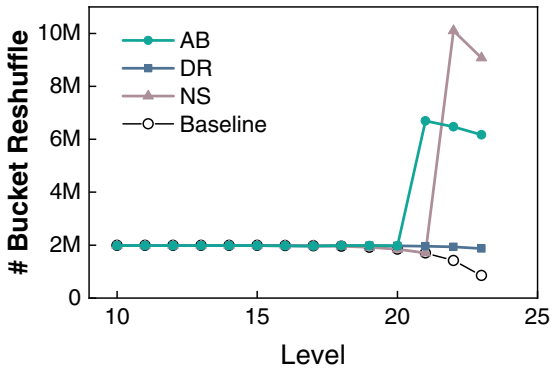
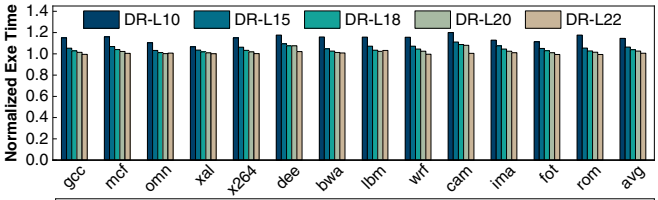


Fig. 10: Comparing number of reshuffles across the levels.

C. DR Sensitivity Analysis

Fig. 11 denotes the result of a sensitivity analysis of DR scheme across the level choice. DR-L18 in Fig. 11 is the same as DR in Fig. 8. Top levels are less desirable for remote allocation due to their low contribution to space demand. For instance, the top 17 levels account for less than 1% of the space, while their reshuffle number contributes equally to performance as other levels.



Normalized Space Demand				
DR-L10	DR-L15	DR-L18	DR-L20	DR-L22
50.00%	50.03%	50.26%	51.04%	54.16%

Fig. 11: Sensitivity analysis of DR to the number of levels.

D. Dead Block Lifetime Analysis

We studied the lifetime of dead blocks. A dead block's lifetime is defined as how long it has been invalid. In Fig. 12 the X-axis indicates the tree levels while the Y-axis indicates the lifetime of dead blocks in terms of the number of online accesses. We report the minimum, average, and maximum lifetime of all dead blocks at each level. The three lines are the average of all benchmarks.

From the figure, the dead blocks from buckets above level 18 (i.e., closer to the root) tend to have a lifetime close to zero, indicating most of these dead blocks get reclaimed in a very short period of time. However, for dead blocks close to the leaves, the average lifetime is large, indicating that dead blocks at these levels tend to be *invalid* for a long duration.

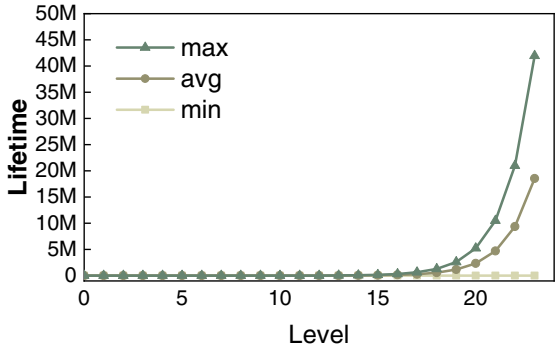
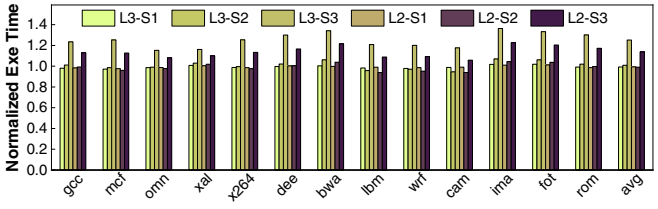


Fig. 12: Dead blocks lifetime across tree levels.

E. NS Design Exploration

To determine the settings for NS, we studied different configurations and summarized the results in Fig. 13. In this figure, Ly-Sx means that, for the last y levels, NS shrinks the S value by x. From the figure, an aggressive configuration, e.g., L3-S3, has large performance degradation. Note, it differs from Fig. 4 because CB is the baseline here. We therefore chose L2-S2 for NS and L3-S1 for DS+NS, i.e., AB.



Normalized Space Demand					
L3-S1	L3-S2	L3-S3	L2-S1	L2-S2	L2-S3
59.4%	52.1%	44.8%	60.4%	54.2%	48.0%

Fig. 13: Design exploration of NS.

F. Remote Allocation Effectiveness

Fig. 14 indicates the ratio of extended S values over the total number of bucket allocations. As shown in Fig. 3, there are abundant dead blocks available at each level. Therefore, `DR` is able to extend almost all of the bucket allocations after gathering enough dead blocks in *DeadQ*. In contrast, when `NS` is also enabled, there are fewer dead blocks available at a time. Thus, `AB` has a lower extending ratio of 74%. Note, this ratio remains the same across different applications as the dead block availability is not application dependant.

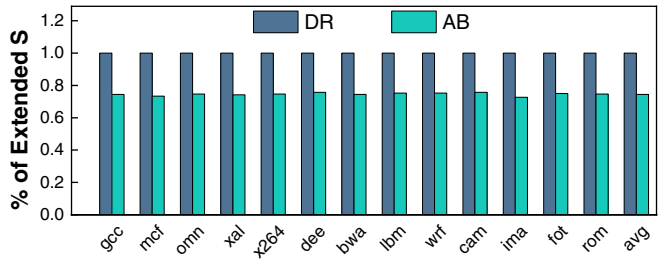


Fig. 14: AB-ORAM capability for extending the S value.

G. Generalizability Over Different Applications

To assess the generalizability, we repeat the experiment with applications from another benchmark suite; PARSEC [5], [7]. Fig. 15 reports the space and performance results. Our space saving remains the same as it is not application dependent. `NS` has a similar execution time as `Baseline`. `DR` and `AB`, on average, incur 3% and 4% performance overheads, respectively.

	NS	DR	AB
Space Reduction	19%	25%	36%
Space Utilization	38.5%	41.5%	48.5%

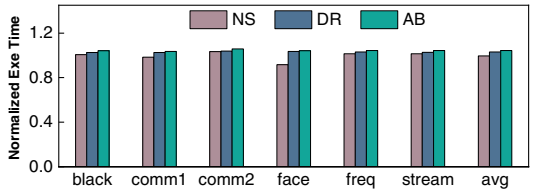


Fig. 15: Generalizability analysis of AB-ORAM.

H. Storage Overhead

On-chip Overhead. Given we track dead blocks for 6 levels, and *DeadQs* are empirically set to have 1000 entries, the on-chip space is 21KB, which is small for modern processors.

Memory Overhead. For Ring ORAM, the bucket metadata takes 33B that fits into a block (i.e. 64B). To avoid incurring any performance penalty during the metadata access phase we keep the extra added metadata by AB-ORAM less than a block size (33B + 28B) by setting $R = 6$ in Table I.

IX. Related Work

Che *et. al.* proposed a channel imbalance-aware scheduler [8] to minimize the channel imbalance for read requests in Ring ORAM. Devadas *et. al.* proposed Onion ORAM [11] to construct a constant bandwidth blowup scheme. It leverages poly-logarithmic server computation to avoid the logarithmic lower bound on ORAM bandwidth blowup. Chen *et. al.* implemented Onion Ring ORAM [9] to outperform logarithmic-bandwidth ORAM such as Ring ORAM. Hoang *et. al.* developed a distributed ORAM scheme [18] to achieve client storage efficiency in addition to efficient client-server bandwidth. Many optimizations were proposed for Path ORAM with some being applicable to Ring ORAM [13], [14], [21], [22], [33], [34], [36], [37], [38].

Recent studies have proposed to adopt ORAM for protecting data storage servers [20], [27]. Memory access patterns can be effectively protected by hardware enhancements [2], [4]. Hardware-assisted security schemes are also developed to mitigate the impact of data authentication on performance [31].

Skewed Merkle Tree was proposed to reduce the number of memory accesses [30], [40]. A skewed tree has imbalanced left and right subtrees such that path lengths may vary. AB-ORAM keeps the full binary tree and only reduces the size of the nodes at some levels, i.e., paths are of the same length.

Radix Path [3] was proposed on top of Path ORAM to reduce the space demand. In this scheme, the root node is expanded to a 1000-entry node to buffer nodes of other levels and all other buckets are reduced in size. It requires a very expensive background eviction to avoid path overflow.

X. Conclusion

In this paper, we propose AB-ORAM to address the space inefficiency of Ring ORAM. AB-ORAM reduces the space demand by reclaiming the dead blocks space via remote allocation. It furthers the space reduction by setting a non-uniform bucket size across the tree levels. It shrinks the buckets close to the leaves. AB-ORAM is orthogonal to the latest optimization of Ring ORAM and effectively lowers the overall space demand. On average, AB-ORAM achieves 36% space reduction over the state-of-the-art while introducing very low performance overhead.

Acknowledgements

We thank all anonymous reviewers for their constructive suggestions. This work is supported in part by NSF grants #2011146, #1910413, #2154973, #1725657, and a startup funding from the University of Pittsburgh.

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] K. S. Al-Saleh and A. Belghith, "Radix path: A reduced bucket size oram for secure cloud storage," *IEEE Access*, 2019.

[4] 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.

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

[6] D. Cao, M. Zhang, H. Lu, X. Ye, D. Fan, Y. Che, and R. Wang, "Streamline ring oram accesses through spatial and temporal optimization," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2021.

[7] 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.

[8] Y. Che, Y. Hong, and R. Wang, "Imbalance-aware scheduler for fast and secure ring oram data retrieval," in *2019 IEEE 37th International Conference on Computer Design (ICCD)*, 2019.

[9] H. Chen, I. Chillotti, and L. Ren, "Onion ring oram: Efficient constant bandwidth oblivious ram from (leveled) tfhe," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019.

[10] I. Corp., *Pin - A Dynamic Binary Instrumentation Tool*, 2012. [Online]. Available: https://software.intel.com/content/www/us/en/develop/articles/pin-a-dynamic-binary-instrumentation-tool.html

[11] S. Devadas, M. van Dijk, C. W. Fletcher, L. Ren, E. Shi, and D. Wichs, "Onion oram: A constant bandwidth blowup oblivious ram," in *TCC*, 2016.

[12] M. Dworkin, E. Barker, J. Nechvatal, J. Foti, L. Bassham, E. Roback, and J. Dray, "Advanced encryption standard (aes)," 2001.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

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

[18] T. Hoang, C. D. Ozkaptan, A. A. Yavuz, J. Guajardo, and T. Nguyen, "S^3oram: A computation-efficient and constant client bandwidth blowup oram with shamir secret sharing," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017.

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

[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] 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.

[22] C. N., 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.

[23] M. Raoufi, Y. Zhang, and J. Yang, "IR-ORAM: Path access type based memory intensity reduction for path-oram," in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2022.

[24] L. Ren, C. Fletcher, A. Kwon, E. Stefanov, E. Shi, M. van Dijk, and S. Devadas, "Constants count: Practical improvements to oblivious RAM," in *24th USENIX Security Symposium (USENIX Security 15)*, 2015.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[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] J. Szefer and S. Biedermann, "Towards fast hardware memory integrity checking with skewed merkle trees," in *Proceedings of the Third Workshop on Hardware and Architectural Support for Security and Privacy*, 2014.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] Y. Xu, W. Cui, and M. Peinado, "Controlled-channel attacks: Deterministic side channels for untrusted operating systems," in *2015 IEEE Symposium on Security and Privacy*, 2015.

[36] 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.

[37] 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.

[38] 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.

[39] 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.

[40] Y. Zou and M. Lin, "Fast: A frequency-aware skewed merkle tree for fpga-secured embedded systems," in *2019 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, 2019.