rORAM: Efficient Range ORAM with $\mathbb{O}(\log^2 N)$ Locality

Anrin Chakraborti*, Adam J. Aviv†, Seung Geol Choi†, Travis Mayberry†, Daniel S. Roche†, Radu Sion*
*Stony Brook University, {anchakrabort, sion}@cs.stonybrook.edu
†United States Naval Academy, {aviv, choi, mayberry, roche}@usna.edu

***Abstract*—Oblivious RAM protocols (ORAMs) allow a client to access data from an untrusted storage device without revealing to that device any information about their access pattern. Typically this is accomplished through random shuffling of the data such that the storage device cannot determine where individual blocks are located, resulting in a highly randomized access pattern. Storage devices however, are typically optimized for *sequential* access. A large number of random disk seeks during standard ORAM operation induce a substantial overhead.**

In this paper, we introduce rORAM, an ORAM specifically suited for accessing ranges of *sequentially logical blocks* while *minimizing the number of random physical disk seeks*. rORAM obtains significantly better asymptotic efficiency than prior designs (Asharov et al., ePrint 2017, Demertzis et al., CRYPTO 2018) reducing *both* the number of seeks and communication complexity by a multiplicative factor of $\mathbb{O}(\log N)$. An rORAM prototype is 30-50x times faster than Path ORAM for similar range-query workloads on local HDDs, 30x faster for local SSDs, and 10x faster for network block devices. rORAM's novel disk layout can also speed up standard ORAM constructions, e.g., resulting in a 2x faster Path ORAM variant. Importantly, experiments demonstrate suitability for real world applications – rORAM is up to 5x faster running a file server and up to 11x faster running a range-query intensive video server workloads compared to standard Path ORAM.

I. Introduction

ORAM. An attacker viewing communications or tracking accesses of a storage user can determine a wealth of private information. Data encryption does not prevent this since access patterns often reveal nearly as much as the data contents themselves [28]. Oblivious RAM (ORAM) [27, 38, 41] aims to solve this by making access patterns indistinguishable to an adversary observing reads/writes to untrusted storage.

Typically, ORAM performance metrics have focused on communication overhead, or bandwidth, loosely describing the number of additional data reads/writes needed to perform a single access [27]. More recently, other metrics have included local computation complexity and round complexity. Numerous ORAM constructions [10, 38, 40, 41, 42, 45] have been proposed and studied optimizing these performance measures.

Data locality and range ORAM. One important measure, so far largely overlooked, is *data locality*, the spatial locality of data in storage, where related data is stored adjacent in memory rows or blocks on disk. Due to caching effects at all levels of the memory hierarchy, it has long been understood that taking advantage of spatial locality can have significant performance benefits. In particular, a single cache miss overhead is more costly than executing 100 instructions. Disk seek overhead costs (e.g., time) often exceed 10000 times the bandwidth cost of reading a single sequential block from that disk [17]. This observation has led to the development of efficient data structures and algorithms which improve performance by optimizing data locality in storage [18].

However, in the case of ORAMs, the randomization necessary to ensure privacy seems to be in direct conflict with data locality. Even for a single access, a typical ORAM requires many non-sequential accesses to the untrusted data store. Even worse, the upper-layer (e.g., file systems) generating optimized accesses with high degree of locality to the underlying storage, gains no benefit when using a standard ORAM to interface with a physical store. This is because in ORAMs, the physical locations have no correlation with their logical addresses.

To address this, recently, Demertzis et al. [21] considered locality for ORAMs in the context of searchable encryption and provided an ORAM construction with $\mathbb{O}(1)$ disk seeks and $\mathbb{O}(N^{1/3}\log^2 N)$ blocks communication complexity. This is a logarithmic improvement over Path ORAM in the number of seeks (Table I), but the significant bandwidth blowup renders the construction suitable only for very specific applications.

Further, in [21], range accesses – a key use case where locality stemming from upper-layer accesses (e.g., in a file system) should be heavily leveraged – are still inefficient. For a range of size r, the number of disk seeks required is $\mathbb{O}(r)$ (note the dependence on the range size).

Asharov et al. [9] specifically considered the issue of supporting efficient range queries for ORAMs, by *making disk seeks independent of the range size*. They show that ORAM range query locality directly conflicts with standard ORAM security requirements. By definition, a standard ORAM must not reveal whether a client requests any r random items or a contiguous region of length r. An ORAM protocol that provides both locality and security must necessarily incur prohibitive bandwidth overhead.

Asharov et al. further observed that carefully relaxing the traditional ORAM security definition to match realistic scenarios allows for significantly more efficient solutions. Specifically, if leaking the rough size (i.e., $\lceil \log_2 r \rceil$) of each accessed range is acceptable, it is possible to design a range ORAM construction with $\mathbb{O}(\log^3 N \cdot (\log\log N)^2)$ seeks per operation, *independent of the length r of the range* (Table I).

For comparison, consider that Path ORAM needs $\mathbb{O}(r \cdot \log^2 N)$ seeks for r sequential accesses, where the dependency

Network and Distributed Systems Security (NDSS) Symposium 2019
24-27 February 2019, San Diego, CA, USA
ISBN 1-891562-55-X
https://dx.doi.org/10.14722/ndss.2019.23320
www.ndss-symposium.org

on r stems mainly from a lack of locality. [9] does better asymptotically when $r = \Omega(\log N \cdot (\log\log N)^2)$, at the cost of $\mathbb{O}(\log N)$ times higher communication complexity.

Our work. Unfortunately, this reduction in the number of seeks comes at the cost of significant bandwidth overhead, which is often times much more expensive, especially when data is outsourced to remote servers.

To mitigate this, we ask the following important question:

Can we construct an efficient range ORAM scheme with data locality, while ensuring that accesses to a small range is asymptotically as fast as the traditional ORAMs?

rORAM answers affirmatively and provides a highly efficient range query mechanism with locality, with $\mathbb{O}(\log^2 N)$ seek and $\mathbb{O}(r \cdot \log^2 N)$ non-amortized communication complexity, $\mathbb{O}(\log N)$ times more efficient than existing work. Importantly, note that for singleton ranges, rORAM has the same asymptotic bandwidth requirements as standard Path ORAM [41] with a server-side position map!

A. Security & Application Setting

Security of range ORAM. At first glance, it may seem that allowing range size leaks seriously weakens ORAM security. However, in most practical cases, this leak already exists inherent to the deployment.

For example, a typical file system running on top of an ORAM issues a majority of its accesses in tightly time-adjacent bursts of sequential block ranges. This immediately leaks the range sizes to any underlying untrusted storage. In fact, explicitly hiding range sizes may be futile if the underlying storage is already aware that a file system runs on top of it. In the following, we detail several other considerations that strengthen the case for accepting range size leaks.

First, consider that rORAM leaks only the rough length of a range, where the actually queried range size is always a power of 2, i.e., $i = \lceil \log_2 r \rceil$. Thus for any user-desired range length, e.g, $\{1, 2, 3, \ldots, \mathsf{maxlen}\}$, the leakage profile will be $\{\lceil \log_2 1 \rceil, \lceil \log_2 2 \rceil, \lceil \log_2 3 \rceil, \ldots, \lceil \log_2 \mathsf{maxlen} \rceil\}$. This leakage contains $\mathbb{O}(\log_2 N)$ different values and can be understood as $\mathbb{O}(\log_2 N)$ different possible padding lengths. Of course, padding any arbitrary length r always to a fixed N provides the best security, but it also greatly increases communication costs. Having variable padding lengths provides a tuning knob to trade off between efficiency and security.

More importantly, even with fixed-length padding (the most secure option), standard ORAMs leak significant information, mainly through the timing channel discussed above, not captured by the ORAM security definition. As discussed, for a typical file system deployed on top of a standard ORAM block device, accesses to different files/metadata/etc. are highly correlated and can be determined accurately using timing information on the number of blocks requested within a given time window. No practically viable solution exists for this leak.

Application setting. Many applications, such as searchable encryption, are well suited to less strict ORAM security guarantees. Weaker ORAMs have been previously used to design efficient dynamic searchable encryption schemes [32]. Range ORAMs are particularly useful in this setting [21].

Further, as shown by experiments (Section VII), rORAM is extremely well suited for deployment with traditional file systems (e.g., ext4 file server). File systems typically generate requests of variable sizes for both reading/writing files and updating metadata. To achieve acceptable I/O throughputs, the underlying ORAM block device needs to support efficient queries for arbitrarily-sized ranges of sequential blocks.

Accordingly, rORAM is designed to efficiently execute range queries of variable sizes. This significantly speeds up file system operation and for large file applications (e.g., a video server), the gains are even more noticeable. For example, rORAM features a 5x speedup over Path ORAM running a typical file server and an 11x speedup for a video server application running on a local HDD.

In summary, rORAM generalizes standard ORAMs that do not specifically support range queries, and provides an easy-to-tune tradeoff between performance and security:

Applications querying only singleton ranges achieve the same security guarantees as on a traditional ORAM at similar costs. Applications querying entire ranges get significant performance increase at an easily quantifiable security cost, namely leaking the size of the range.

B. rORAM Highlights

Locality-aware disk layout and batch writes. As we will see later, a main rORAM building block is a modified version of Path ORAM. We first introduce a new technique for reducing the number of seeks in a Path ORAM.

Tree-based ORAMs, such as Path ORAM, update data in the server-side tree through an *eviction* operation, which reads a specific path in its entirety and writes back as many blocks as possible from the client-local *stash* along the path. To prevent overflows, consecutive eviction paths are chosen with minimum overlap, usually in bit-reversed lexicographical ordering[1] of the leaf identifiers [26].

However, this has a detrimental effect on the number of seeks required when evictions are performed in batches, since successively chosen eviction paths are topologically distant from each other in the tree. Specifically, when tree nodes are stored at random locations, the number of seeks required to batch b evictions is $\mathbb{O}(b \cdot \log N)$ (note the dependence on b).

Since range queries write back multiple blocks to the tree, a more efficient batching mechanism is desirable. By design, rORAM enables many *evictions* to execute with very few seeks – the number of seeks is independent of the number of evictions performed. To this end, *rORAM disk layout ensures that tree nodes accessed in successive evictions are physically located next to each other on the storage device*.

In particular, the paths (i.e., corresponding to leaf nodes) in the ORAM tree are labeled in bit-reversed lexicographic ordering. Then, the physical buckets at each level of the tree

[1]In bit-reverse ordering, numbers are ordered by treating the leftmost bit as the least significant bit. For example, the sequence of 3-bit-reversed number ordering is 000, 100, 010, 110, . . ., 111; that is, 0, 4, 2, 6, . . ., 7 in decimal.

	Seeks	Bandwidth	Server Space	Client Storage	Leakage
rORAM (this work)	$\mathbb{O}(\log^2 N)$	$\mathbb{O}(r \cdot \log^2 N)$	$\mathbb{O}(N \log N)$	$\mathbb{O}(L \cdot \lambda)$	$\lceil \log_2 r \rceil$
Asharov et al. (Range ORAM) [9]	$\mathbb{O}(\log^3 N \cdot (\log\log N)^2)$, amort.	$\mathbb{O}(r \cdot \log^3 N)$, amort.	$\mathbb{O}(N \log N)$	$\mathbb{O}(L \cdot \lambda)$	$\lceil \log_2 r \rceil$
Path ORAM (rec. PM) [41]	$\mathbb{O}(r \cdot \log^2 N)$	$\mathbb{O}(r \cdot \log^2 N)$	$\mathbb{O}(N)$	$\mathbb{O}(\lambda)$	none
Path ORAM (local PM) [41]	$\mathbb{O}(r \cdot \log N)$	$\mathbb{O}(r \cdot \log N)$	$\mathbb{O}(N)$	$\mathbb{O}(N)$	none
Demertzis et al. [21]	$\mathbb{O}(r)$	$\mathbb{O}(r \cdot N^{1/3} \cdot \log^2 N)$	$\mathbb{O}(N)$	$\mathbb{O}(N^{1/3} \log^2 N)$	none

TABLE I

PERFORMANCE COMPARISON FOR A CLIENT ACCESSING A REGION OF r CONTIGUOUS BLOCKS. L IS THE MAXIMUM RANGE SIZE SUPPORTED BY THE RANGE ORAM SCHEMES AND λ IS THE SECURITY PARAMETER. ALL COMPLEXITIES ARE IN TERMS OF NUMBER OF BLOCKS.

are stored adjacent to each other, following the same bit-reversed ordering (see Figure 2 and Figure 3).

As a result, when performing b evictions together, buckets can be fetched (and written back) *level-by-level*. Due to the physical layout of the tree, the $b \mod 2^i$ buckets required from level i, will be adjacent to each other on physical storage and can be read with only 1 seek. Effectively, b evictions now require in total $\mathbb{O}(\log N)$ seeks, *independent of* b.

Interestingly, since rORAM applies this technique to Path ORAM itself, it achieves better efficiency not only for range query applications but also for standard Path ORAM performance. Experiments show a 2x speedup for standard Path ORAM equipped with the disk-aware layout and batched evictions (Section VII).

Multiple sub-ORAMs with different data locality. rORAM deploys $\mathbb{O}(\log N)$ separate Path ORAM based sub-ORAMs, each of which contains a copy of the *same* data at all times. Further, each sub-ORAM is optimized to serve a different range size with minimal disk seeks.

Specifically, the ith sub-ORAM is optimized for access to a contiguous range of length r where $\lceil \log_2 r \rceil = i$. This leaks the rough size of the given range, but it allows the sub-ORAM to be highly efficient in serving the query.

Locality-sensitive mapping. The locality-aware disk layout only ensures that multiple batched evictions incur a small number of disk seeks independent of the batch size. However the layout does not provide any seek-related guarantees when *querying* for multiple blocks from different random paths in a Path ORAM tree. This is because Path ORAMs place blocks along random tree paths, regardless of the block address. As a result, fetching a logically related range of blocks still requires fetching multiple random paths from the tree, unavoidably incurring a large number of seeks. Specifically, if each node of the tree is stored at a random location on disk, fetching a range of size r requires $\mathbb{O}(r \cdot \log N)$ seeks.

To mitigate this, rORAM introduces a novel block mapping scheme consistent with the locality-aware disk layout. In particular, the scheme places the first block in a range onto a random path in the tree, and *all subsequent blocks in the range are placed along paths that are stored adjacent to each other on disk*. Now, reading a range of size r requires only $\mathbb{O}(\log N)$ seeks! Critically, as we will show, with an appropriately-sized stash, the mapping provides standard privacy assurances.

Efficient management of multiple position maps. Note that since rORAM duplicates data across multiple sub-ORAMs, updates to a block in one sub-ORAM should be reflected to the other sub-ORAMs as well to ensure consistency. This is challenging because the sub-ORAMs are initialized with their own random seeds, and hence the location of a particular block in one sub-ORAM does not immediately provide its location in the other sub-ORAMs.

Further, in the presence of a stateless or limited-storage client, the position maps associating logical to physical addresses in each of the sub-ORAM trees must also be stored obliviously on the server. This is a well-known problem with tree-based ORAMs like Path ORAM. Locating and updating a block in all the sub-ORAMs using their corresponding position maps will unfortunately result in $\mathbb{O}(\log^3 N)$ seeks and communication overhead.

To tackle this, rORAM introduces a new *distributed position map*. First, each block in each sub-ORAM stores additional information allowing rORAM to immediately look up the physical location of that same block in the other ORAMs, not unlike the case of pointer-based oblivious data structures [44]. Second, note that if a client accesses a range using the ith sub-ORAM, the positions of the blocks in the accessed range need to be updated *only in the* i*th sub-ORAM*. The range needs to be evicted to the other trees as well, but there is no need to refresh the positions in the other sub-ORAMs since they are still hidden and look random to the adversary – the range has not been *read* from those trees, and the (deterministic) eviction schedule is independent of the positions.

As a result, rORAM needs only $\mathbb{O}(\log^2 N)$ seeks for the position map accesses and updates per operation, matching the asymptotic cost of the data access itself.

Simplicity of construction & evaluation. A key advantage of rORAM is the simplicity of implementation and deployment. Prior solutions [9] are amortized and use complex building blocks (such as locality-friendly oblivious sort etc.)

rORAM mechanisms can be implemented with simple yet effective modifications to existing tree-based ORAM designs. rORAM is evaluated in detail for real workloads, and compared against standard ORAMs. rORAM is 30-50x times faster than Path ORAM for similar range query workloads on local HDDs, 30x faster for local SSDs, and 10x faster for network block devices. Further, the rORAM locality-aware physical layout can be deployed independently to speed-up standard Path ORAM by a factor of 2x. Finally, application benchmarks demonstrate that rORAM is up to 5x faster running a file server and up to 11x faster running a range-query intensive video server workloads compared to Path ORAM.

C. Summary

Based on the construction presented herein, rORAM makes the following contributions:

1) A new oblivious range ORAM construction that optimizes data locality and is faster by a factor of $\mathbb{O}(\log N)$ over previous results [9] (Table I).

2) A new locality-preserving sub-ORAM design based on bit-reversed lexicographic ordering that achieves: i) a highly-optimized physical disk layout for efficiently batching evictions, and ii) an efficient tree paths mapping mechanism for ensuring data locality in range queries.
3) A new distributed position map construction for efficiently locating block replicas in multiple ORAMs.
4) An open-source implementation of rORAM. To the best of our knowledge, rORAM is the first implementation of a Range ORAM construction.
5) Micro-benchmarks showing significant performance increase for range-query workloads compared to standard ORAMs. For example, rORAM is 30-50x faster than Path ORAM for range queries of size $\geq 2^{10}$ blocks on local HDDs, 30x faster for local SSDs, and 10x faster for network block devices.
6) Application benchmarks showing suitability for real world applications: rORAM is up to 5x faster running a file server and up to 11x faster running a range-query intensive video server across several platforms.

II. Prior Work

Oblivious RAM (ORAM) and applications. ORAM protects the access pattern so that it is infeasible to guess which operation is occurring and on which item. Since the seminal work by Goldreich and Ostrovsky [27], many works have focused on improving ORAM efficiency (e.g., [10, 34, 38, 42, 45]).

ORAM plays as an important tool to achieve secure cloud storage [31, 39] and secure multi-party computation [22, 30, 42] and secure processors [24, 29]. There also have been works to hide the access pattern of protocols accessing individual data structures, e.g., maps, priority queues, stacks, and queues and graph algorithms on the cloud server [35, 44].

Locality in searchable encryption. Data locality has been a useful metric for evaluating searchable symmetric encryption [8, 14, 19]. In these models, the client stores their data remotely and encrypted, but the server can perform searches upon the data (e.g., a keyword search) without revealing the plain text. While related, searchable symmetric encryption does not protect against access patterns, e.g., revealing whether the same data item has been accessed multiple times.

ORAMs with locality. In the closest related work to this one, Asharov et al. [9] first introduced the weaker security model for range ORAMs by which the size of the range is leaked to provide data locality. Their construction, built on top of a hierarchical ORAM construction [27], also makes use $\mathbb{O}(\log N)$ series of ORAMs by which each ORAM forms the layer in the tree. Locality is achieved by storing the ranges on each level as increasingly larger blocks of size 2^i. They show that the number of seeks per access is $\mathbb{O}(\log^3 N \cdot (\log\log N)^2)$.

Further, [9] proposes a more relaxed definition for File ORAMs where the sizes of individual files are revealed per access. This results in better asymptotic performance at the cost of less access flexibility. First, File ORAMs do not provide any opportunity for padding accesses in contrast to the variable-length padding options available for range ORAMs – access to individual files/metadata of different sizes are always distinguishable and multiple accesses to the same file can be linked using the file size. Second, File ORAMs cannot support efficient arbitrary-sized reads/edits to portions of large files – files are always accessed in their entirety. In this work, we adapt the security setting and flexibility of range ORAMs but with a more efficient construction. We primarily compare our work against the range ORAM construction in [9].

Data locality has been used previously as a performance metric in the setting of write-only ORAMs. In this security model, reads are assumed to be unobservable by an attacker, but writes to data can be observed and must be obfuscated. First introduced by Blass et al. [11] in the context of protecting hidden volumes, a randomized procedure was used to achieve obliviousness. Later, Roche et al. [36] showed that write-obliviousness can be achieved with deterministic, sequential writing patterns. However, the data locality of reads was not evaluated, and depends largely on the write pattern itself.

Improvements for the position map have been produced by using temporal locality. FreeCursive [24] employs a PosMap Lookaside Buffer (PLB) to reduce the overhead of using a position map. While leveraging *temporal* locality in the position map, this work does not provide *spatial* data locality.

ORAMs have also been used to expand searchable encryption with locality. Work by Demertzis et al. [21] proposed a hierarchical square-root ORAM [27] to support searchable encryption. This scheme makes use of locality-preserving version of Melbourne Shuffle [33] to achieve $\mathbb{O}(1)$ seeks, but requires $\mathbb{O}(N^{1/3}\log^2 N)$ communication and local storage with higher server storage. It also does not support range queries naturally, which adds a multiplicative cost to communications and seeks.

III. Background & Security Definitions

ORAM. An Oblivious RAM (ORAM) protocol allows a client to store and manipulate an array of N blocks on an untrusted, honest-but-curious server without revealing the data or access patterns to the server. Specifically, the logical array of N blocks is indirectly stored into a specialized back-end data structure on the server, and an ORAM scheme specifies an access protocol that implements each logical access with a sequence of physical accesses to that back-end structure. An ORAM scheme is secure if for any two sequences of logical accesses of the same length, the physical accesses produced by the protocol are computationally indistinguishable.

More formally, let $\vec{y} = (y_1, y_2, \ldots)$ denote a sequence of operations, where each y_i is a $\mathsf{Read}(a_i)$ or a $\mathsf{Write}(a_i, d_i)$; here, $a_i \in [0, N)$ denotes the logical address of the block being read or written, and d_i denotes a block of data being written. For an ORAM scheme Π, let $\mathsf{Access}^{\Pi}(\vec{y})$ denote the physical access pattern that its access protocol produces for the logical access sequence $\vec{y}$. We say the scheme Π is *secure* if for any two sequences of operations $\vec{x}$ and $\vec{y}$ of the same length, it holds

$$\mathsf{Access}^{\Pi}(\vec{x}) \quad \approx_c \quad \mathsf{Access}^{\Pi}(\vec{y}),$$

where $\approx_c$ denotes computational indistinguishability (with respect to the security parameter λ).

A. Range ORAM and Locality

In this work, we study ORAMs specifically suited for accessing sequential ranges of data. This requires a slightly different security definition to capture the fact that range ORAMs access ranges of blocks instead of just single blocks.

Let let $\vec{y} = (y_1, y_2, \ldots)$ denote a sequence of operations, where each y_i represents an access to a range of sequential blocks. Let y_i be either $\mathsf{ReadRange}(a_i, \ell_i)$ or $\mathsf{WriteRange}(a_i, \ell_i, d_1, \ldots, d_{\ell_i})$. Here, a_i refers to a logical block as before, but additionally ℓ_i indicates the *number of sequential blocks to access* starting with a_i. $d_1, \ldots, d_{\ell_i}$ are the blocks of data to be written to the logical addresses $a_i, a_i + 1, \ldots, a_i + \ell_i$.

Let $\mathtt{len}(y_i)$ signify the length ℓ_i for of the range access y_i. A Range ORAM scheme Π is secure if for any two sequences of operations $\vec{x}$ and $\vec{y}$ of the same length, subject to the following constraint:

$$\forall i : \lceil \log_2(\mathtt{len}(y_i)) \rceil = \lceil \log_2(\mathtt{len}(x_i)) \rceil$$

it holds that

$$\mathsf{Access}^{\Pi}(\vec{x}) \quad \approx_c \quad \mathsf{Access}^{\Pi}(\vec{y}),$$

where $\approx_c$ denotes computational indistinguishability (with respect to the security parameter λ).

Informally, this means that a Range ORAM can leak the rough size of the ranges that are being accessed by each operation. That is, the length of two accesses only needs to be within $(2^k, 2^{k+1}]$ for some k in order for them to be indistinguishable. In other words, $\mathbb{O}(\log \ell_i)$ bits are leaked per access, which is the *order of magnitude* of the range.

Locality and seeks. Locality of an algorithm is well defined in prior works [9, 19]. Informally, this is *the number of seeks* required on the storage medium during the execution of that algorithm. If an ORAM algorithm performs accesses to the physical storage at the addresses $\vec{z} = (z_1, z_2, \ldots)$, in that order, then a *seek* is defined as an index i such that $z_{i+1} \neq z_i + 1$. The total number of seeks across $\vec{z}$ is the *locality* of the algorithm.

B. Path ORAM

One of the most efficient ORAM constructions currently known is Path ORAM, presented in the seminal work of Stefanov et al. [41]. Path ORAM works by storing data blocks in a complete binary tree with N leaf nodes (or buckets). Each bucket in the tree has space for a small constant number of blocks, denoted Z. During initialization, leaf buckets are numbered 0 to $N - 1$ and blocks are each given random tags (or positions) from the range $[0, N)$. In addition, there is a single small *stash* area which holds some blocks temporarily. The tree maintains an *invariant* that if a block has tag p, it will exist *either in the stash or somewhere along the path from the root of the tree to the pth leaf node*.

Data access and eviction. In order to retrieve a block, the client must first determine the path position tag t for the block. This is done by maintaining a map, called the *position map*, that relates logical block addresses to their random positions. Once the tag t has been found, the client retrieves the entire path from root to the tth leaf node and stores the buckets on this path locally. The requested logical block is accessed by scanning the retrieved buckets. The chosen block is then assigned to a new random leaf node (i.e. a tag), and its tag is updated accordingly and stored back in the position map. Finally the updated block itself is appended to the stash area. Note that this occurs invariably as the tag is reassigned for every access in order to hide the access pattern.

Because the stash has a fixed size, eventually it is necessary to *evict* blocks from the stash back to the tree buckets. In the simplest setting, every data access (which involves appending one new block to stash) is followed immediately by rewriting, or *evicting*, along a single path in the tree. In this step, the client picks a path in the tree (either randomly or deterministically using bit-reversed ordering [26, 41]) and retrieves the buckets for that path from the storage device. The existing blocks in that path are then re-ordered, along with the blocks in stash, so that every block is stored as far down in the path as it can go, subject to the invariant and the size of the buckets. Any block which still does not fit in the path is stored back to the stash area.

Bucket and stash size. Because each bucket has a fixed size, as does the stash area, it is possible for the scheme to "break" by running out of room in the stash, a situation referred to as *stash overflow*. The original Path ORAM used a random eviction strategy and showed that if the bucket size is at least $Z \geq 5$, the probability of stash overflow decreases exponentially in the stash size [41]. The Ring ORAM construction improved this further, demonstrating that $Z \geq 3$ is sufficient with a *deterministic* eviction strategy [34].

Position map. The map storing each block's tag can be quite large; it is $N \log_2 N$ bits long. If the client is not capable of storing the map locally, it can be stored recursively in a series of $\mathbb{O}(\log N)$ smaller ORAMs on the server. Alternatively, one can use an oblivious data structure (more specifically, an oblivious trie [36, 41]) to store the position map. In either solution, the total communication overhead for a single access with Path ORAM is $\mathbb{O}(\log^2 N)$.

Access Complexity. ORAMs are typically evaluated in terms of *bandwidth* – the number of *data blocks* that are downloaded/uploaded in order to complete one logical request. Path ORAM features an overall bandwidth of $\mathbb{O}(\log N)$ data blocks, where N is the total number of blocks in the ORAM. This asymptotic bound holds only under the *large block size assumption* when the data blocks size is $\Omega(\log^2 N)$ bits.

rORAM has the same large block size assumption and all access complexities reported in this paper indicate the number of physical blocks that are accessed overall for fulfilling a particular logical request.

Seeks. If Path ORAM is used as a Range ORAM to retrieve a sequential range of blocks, each block is stored along a random path, and it would require $\mathbb{O}(r \cdot \log N)$ seeks to fetch data blocks, where r is the number of blocks in the range. If the position map is stored server-side recursively, the position map access will additionally require $\mathbb{O}(r \cdot \log^2 N)$ seeks. A locality-friendly ORAM, as we achieve here, should require a number of seeks *independent of the range size r*.

IV. rORAM Construction Overview

In this section, we describe the basic, core construction details for rORAM. We start with multiple Path ORAM trees as building blocks, designating each ORAM for range queries of a specific size. This allows us to design and optimize a particular sub-ORAM for range queries of a given size.

To this end, we show how to achieve data locality for both queries and evictions for each of the sub-ORAMs. This is the result of two key insights – i) using a locality-aware disk layout that dramatically reduces the number of seeks required for performing multiple evictions in batches, and ii) a novel locality-sensitive block mapping mechanism which reduces seeks when querying for blocks in a range.

Finally, we describe a novel distributed position map scheme for efficient query and update of the multiple sub-ORAMs.

A. Core Construction

Multiple ORAMs each covering a subset of ranges. We use multiple sub-ORAMs to store ranges of a specific length, as with the prior work [9]. Let N be the total number of blocks stored in the rORAM, and $L \leq N$ be a parameter indicating the maximum range size that will be supported. Then the rORAM construction makes use of $\ell + 1$ Path ORAMs, where $\ell = \lceil \log_2 L \rceil$; these individual Path ORAMs are labeled $R_0, R_1, \ldots, R_\ell$. An access on ORAM R_i will always access *exactly* 2^i blocks (see Figure 1) which are logically sequential in the range. Note that R_0 is a Path ORAM as it would normally be constructed, with a range size of just one block.

Within a given ORAM R_i, N data blocks are partitioned into ranges of length 2^i, and let r_i^j denote the jth range in R_i, i.e., $r_i^j := [j \cdot 2^i, (j+1) \cdot 2^i)$. Each ORAM R_i is specifically tailored so that contiguous ranges of length 2^i are located close to each other on storage. The tradeoff is that ranges can only be queried in their entirety, consequently $\ell + 1$ separate ORAMs: to support any size range with low overhead.

If the client requests a range that is exactly r_j^i, this could be fulfilled with a single access on R_i by requesting range j. However, we must consider a client requesting an arbitrary range, which may not start on a power-of-two boundary. One strategy for fulfilling such requests in a single access would be to upgrade the query to the next, larger-range ORAM until $r_j^i \in r_{j'}^{i'}$, but there is an issue with this approach. In particular, even for a small range, as small as size 2, it is sometimes impossible to cover the range with a single access, unless the length of ranges of an ORAM is N. For example, suppose $N = 64$ and consider a range $[31, 33)$. No range of the form $[a \cdot 2^i, (a+1) \cdot 2^i)$ can cover $[31, 33)$.

Fortunately, there exists a solution [9, 20, 23]. If a range overlaps a boundary, we can fulfill the request with *two* accesses of the same power-of-two size. For example, access to the range $[15, 22)$ of length 7 would be covered by accessing ORAM R_3 (i.e., $\lceil \log_2 7 \rceil = 3$) with two ranges $[8, 16)$ and $[16, 24)$. We stress that so as *not to leak information about the range boundaries*, we should *always* perform *two* accesses even if the entire request fits within a single range; note that whether a range query is handled by a single access or two is indeed leaks information about the range.

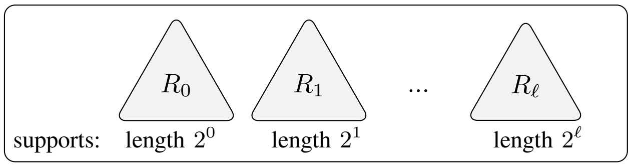


Fig. 1. rORAM Organization. rORAM storing N blocks and supporting ranges up to 2^ℓ consists of $\ell + 1$ tree-based ORAMs $R_0, \ldots, R_\ell$. Each component ORAM R_i contains N blocks and supports ranges of size 2^i. All ORAMs have the same block size.

Comparison with [9]. One crucial difference here compared to the prior work [9] that similarly uses sub-ORAMs to duplicate data and serve range queries of different sizes, is that the sub-ORAMs $R_0, \ldots, R_\ell$ in rORAM have the *same physical block size* regardless of the served range size. This means that a single access in a given sub-ORAM R_i occurs on 2^i blocks and is completed as a single *batched* operation.

In contrast, sub-ORAMs in [9] have different physical block sizes; a range is effectively stored and accessed as a large physical block (called a *superblock*), concatenating the content of the regular blocks in the range. We will see that retaining the same block size for all sub-ORAMs is the key to making rORAM more efficient. For this, we first introduce the operations supported by each sub-ORAM.

Operations for R_i. Recall R_i supports range queries of length 2^i. This requires two operations:

- $\mathsf{ReadRange}(a)$: Takes as input a logical address a and returns the 2^i blocks in the range $[a, a + 2^i)$ from the ORAM. Here a must be a multiple of 2^i, as in $a = b \cdot 2^i$.
- $\mathsf{BatchEvict}(k, \mathsf{stash})$: Perform k evictions as a batch to write back multiple blocks to the ORAM from the stash for each of the k evicted paths. Evictions occur in a deterministic order, and a global counter is used to maintain this order.

Remarks about BatchEvict. Now, recall that in rORAM (and also in [9]), all sub-ORAMs should consistently maintain the same data. By implication, updates in any ORAM R_j, must be followed by updates to every other R_i for $i \neq j$.

Specifically, a $\mathsf{ReadRange}$ operation on sub-ORAM R_j will be followed by a $\mathsf{BatchEvict}(2^j, \mathsf{stash})$ to all $\ell + 1$ sub-ORAMs. Therefore, we cannot assume that evictions to sub-ORAM R_k will always in be in batches of 2^k blocks.

With different physical block sizes (*superblocks*) in [9], it is difficult to perform eviction of a small range in a larger-block ORAM efficiently. Intuitively, in order to update a single block out of the several blocks that constitute a *superblock*, the entire *superblock* needs to be refreshed lest it leaks privacy. This becomes a significant overhead with larger superblocks.

To overcome this, [9] relies on amortizing the cost using a hierarchical ORAM [27]. In this amortized construction, eviction is significantly slower (by a factor of $\mathbb{O}(\log N)$) than standard tree-based ORAMs, *In contrast, by maintaining the same physical block size across all sub-ORAMs, rORAM can perform non-amortized evictions to each sub-ORAM with the same asymptotic complexity as the underlying tree-based ORAM.* This is critical for ensuring that singleton range queries in rORAM can be performed with the same asymptotic complexity as standard tree-based ORAM queries.

Operations on rORAM. rORAM operations are internally composed of operations on each sub-ORAM R_i. For rORAM, we have the following operation:

- $\mathsf{Access}(\mathsf{id}, r)$: Given a range of size r beginning at logical identifier id, with $\lceil \log_2 r \rceil = i$, run $R_i.\mathsf{ReadRange}(a_1)$ and $R_i.\mathsf{ReadRange}(a_2)$ with $a_1 = \lfloor \mathsf{id}/2^i \rfloor$ and $a_2 = (a_1 + 2^i) \bmod N$.
  The updated data blocks are then appended to the stash of all $\ell + 1$ sub-ORAMs. Then, for each R_j, call $R_j.\mathsf{BatchEvict}(2^{i+1}, \mathsf{stash})$.

As mentioned previously, an Access requires two ReadRange's to occur (to avoid leaking properties of the range) resulting in 2^{i+1} data blocks. For every Access, we need to perform the same magnitude of BatchEvict's for *all* $\ell + 1$ ORAMs, updating the data which is duplicated in each tree.

Remark about Choosing L. The choice of an appropriate max range size, L primarily depends on the application. However, the trade-off is that a larger L requires a larger client-side storage. This is because an L-size query necessitates the local storage of L blocks. This is reflected in the rORAM stash size bound (Table I). One thing to note here is that due to rORAM's $\mathbb{O}(\log^2 N)$ bandwidth overhead, for query sizes $\geq \frac{N}{\log^2 N}$, downloading the entire database is faster than accessing the range from an ORAM. Consequently, an appropriate upper bound is $L < \frac{N}{\log^2 N}$, thus ensuring that the rORAM stash size is sub-linear in N.

More importantly, applications rarely access very large ranges all at once, possibly to reduce the overall access latency. Instead, a typical application e.g., a file system breaks down a large access into multiple smaller sequential accesses, often not exceeding 1MB in size. In this case, it suffices to initialize rORAM with $L = 2^8$ blocks. In general, for almost all applications, a reasonable value of $L = \mathbb{O}(\sqrt{N})$ blocks with $\mathbb{O}(\sqrt{N})$ client-side storage. Larger range queries (if any) can be broken down into smaller range queries of appropriate size.

B. Insight 1: Locality-aware Physical Layout

A common extension to Path ORAM is to use a deterministic eviction strategy using bit-reversed ordering of the paths, as described by Gentry et. al [26]. In bit-reverse ordering, counting occurs with the least significant bit on the left, as compared to natural ordering, where the most significant bit is to the left and the least significant is the right. For example, counting in 3-bits, the number to follow 000 is not 001 but rather 100, leading to the sequence of 3-bit-reversed number ordering as 000 (0), 100 (4), 010 (2), 110 (6), 001 (1), 101 (5), 011 (3), 111 (7) — with the decimal value in parenthesis. Each bucket of the tree is now labeled with both its level in the tree and its bit-reversed ordering in that level, as in Figure 2. That is, a bucket labeled as v_i^j signifies the jth bucket among those at level i.

Evicting paths in this order ensures a good "spread" over the tree, making it less likely that any blocks get stuck, by chance, in the higher buckets of the tree and cause an overflow. But as we will show, the bit-reverse ordering can also be leveraged for the physical layout of the tree to achieve data locality.

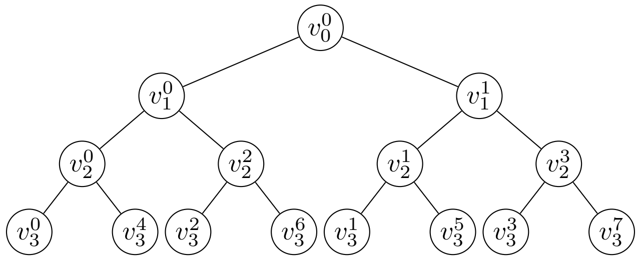


Fig. 2. Labeling of ORAM tree buckets. A bucket label v_i^j signifies the jth bucket among those at level i in bit-reversed order.

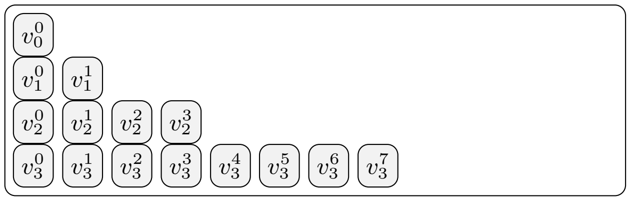

v_0^0
v_1^0 v_1^1
v_2^0 v_2^1 v_2^2 v_2^3
v_3^0 v_3^1 v_3^2 v_3^3 v_3^4 v_3^5 v_3^6 v_3^7

Fig. 3. Physical disk storage of ORAM R_i. Buckets at each level are stored sequentially according to the bit-reversed order.

Locality-aware physical layout of R_i. An important observation is that the the path eviction schedule also implies the deterministic ordering in which data is evicted to nodes within levels of the tree; in particular:

> *The nodes at the same level are ALSO evicted according to the bit-reversed order.*

Let $P(p)$ be a path from the root to a leaf with position p. For example, in the tree in Figure 2, the three consecutive eviction paths $P(v_3^2), P(v_3^3), P(v_3^4)$ visits buckets v_2^2, v_2^3, v_2^0 at level 2, v_1^0, v_1^1, v_1^0 at level 1, and v_0^0, v_0^0, v_0^0. At each level, the buckets are accessed according to the bit-reversed order (with wraparound).

If the ORAM stores each level sequentially on the storage device, according to the bit-reversed eviction ordering of the level (see Figure 3), evictions can be done with optimal number of seeks. Consecutive evictions, as is the case for BatchEvict, occur in bit-reverse order sequentially for each level in the tree. To the best of our knowledge, rORAM is *the first construction that considers the physical layout to improve efficiency of ORAM performance*.

$\mathbb{O}(\log N)$ seeks independent of range size. With a sequential layout of buckets on disk that matches the bit-reversed order at each level x in the R_j sub-ORAM tree, $R_j.\mathsf{BatchEvict}(k)$ will visit $\min(k, 2^x)$ buckets, stored physically adjacent to each other, at level x sequentially. Thus, reading and writing back to each level requires at most 2 seeks, with wraparounds, and the ORAM tree has $\log N + 1$ levels. The total number of seeks performed for $R_j.\mathsf{BatchEvict}(k)$ is therefore $\mathbb{O}(\log N)$. Note that the number of seeks is independent of k, the number of eviction operations performed as a batch. Updating $(\ell + 1)$ sub-ORAMs will require $\mathbb{O}(\ell + 1 \cdot \log N) = \mathbb{O}(\log^2 N)$ seeks.

C. Insight 2: Locality-sensitive Mapping

Tree-based ORAM schemes map logical addresses to paths in the ORAM tree, along which the block corresponding to the logical address is placed. For Path ORAM, a logical address is mapped to a new random path every time the corresponding block is accessed.

Recall that ReadRange on ORAM R_i reads *exactly* 2^i *blocks*. Using the traditional mapping mechanism (i.e., assigning a random path for each block) will not provide locality, resulting in $\mathbb{O}(2^i \cdot \log N)$ seeks to read 2^i random paths.

However, random block placement is critical for the security (and also correctness) of tree-based ORAMs. Designing a new mapping scheme requires careful analysis, in order to both achieve better locality and maintain acceptable security.

Our approach. rORAM uses a hybrid of random and deterministic placement policies for mapping blocks to paths:

- For blocks that do not belong to the same range, a purely random strategy can be applied since no locality guarantees are required.
- For blocks that belong to the same range, it is desirable to place these blocks along paths that are stored close to each other on disk.

The first requirement implies that blocks in different ranges can be mapped to random paths in the tree, independent of each other. While allowing better locality, the second requirement has implications on the security of the scheme. Specifically, blocks that belong to the same range will *not* be mapped independently to paths. Accessing a range of a particular size will be clearly observable based on the paths read. Since we allow the size of queried ranges to be leaked anyway, this does not actually reveal any further information.

Locality-sensitive mapping. *Keeping the locality-aware physical layout in mind, we map the blocks in the same range in R_i to paths that occur successively in the bit-reversed ordering of their leaf identifiers.* Then, the blocks in the range will appear physically adjacent to each other across levels, and we can reduce the number of seeks required to read the range.

In particular, consider logical addresses in $[j \cdot 2^i, (j+1) \cdot 2^i)$ in R_i. Letting $a = j \cdot 2^i$, the addresses are mapped to paths in the tree as follows:

- Address a: Address a is mapped to a random path, i.e., $P(v_h^r)$ where r is chosen at random and $h = \log N$.
- Adresss $a + j$: For $j = 1, \dots, 2^i - 1$, address $a + j$ is mapped to a path $P(v_h^{r+j \bmod N})$.

This ensures that R_i.ReadRange requires $\mathbb{O}(\log N)$ disk seeks: *sequentially scan 2^i buckets from each level of the tree (with wraparound), that are invariably adjacent to each other on storage*. Note that the number of seeks required is independent of the range size, 2^i.

D. Insight 3: Distributed Position Map

Although, the above techniques can achieve the desired $\mathbb{O}(\log N)$ seeks for ReadRange and $\mathbb{O}(\log^2 N)$ seeks for BatchEvict's, a critical challenge we are yet to address is optimizing the cost of multiple position map lookups while updating $\ell + 1$ sub-ORAMs. We detail our solution by first describing the challenges involved.

Challenges of naïve position map construction. In rORAM, each of the sub-ORAMs is uniquely addressed for the blocks they store, even though the same data blocks are duplicated across all ORAMs. As a result, a separate position map has

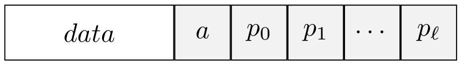


Fig. 4. The structure of a physical block. In addition to data, the block contains the logical address a and the path locations $p_0, \dots, p_\ell$ of the block in ORAMs $R_0 \dots, R_\ell$ respectively.

to be maintained for each sub-ORAM. If stored locally, this extra data is handled at no added cost, but typically the size of the position map would exceed local storage requirements, and would need to be stored securely in the remote storage. Typically this requires a recursive ORAM or an oblivious trie [36, 41] to store the position map obliviously – both solutions require $\mathbb{O}(\log N)$ disk seeks and bandwidth for a position map query [2].

Recall that an Access operation executes ReadRange once and BatchEvict $(\ell + 1)$ times. Each operation needs to look up the corresponding position-map in order to translate a logical address to a tree path, which increases both seeks and bandwidth for R_i.BatchEvict from $\mathbb{O}(\log N)$ to $\mathbb{O}(\log^2 N)$. Consequently, Access requires $\mathbb{O}(\ell \cdot \log^2 N) = \mathbb{O}(\log^3 N)$ seeks and bandwidth.

The goal is to reduce the overall number of seeks and bandwidth from $\mathbb{O}(\log^3 N)$ to $\mathbb{O}(\log^2 N)$ for Access in rORAM, and the key to achieving this is to reduce the number of expensive position-map look-ups to just one.

Reusing physical paths in unread Path ORAMs. A key observation is that since ReadRange reads paths corresponding to a range from only one of the sub-ORAMs, say R_i, the locations of these blocks in the other ORAMs, $R_j \neq R_i$ still remain hidden. Recall that while data is duplicated across the ORAMs, the paths (or sometimes called *tag* hereafter) along which the same logical data blocks are placed in the different sub-ORAMs are independently assigned. Thus, the location of blocks in the queried range in R_i, does not reveal any information about their location in the other sub-ORAMs.

After the ReadRange the data blocks may be updated, and these updates must propagate to all the other ORAMs through a commensurate number of BatchEvict's. *During a* BatchEvict *to sub-ORAM $R_j \neq R_i$, the queried blocks can be written back along the same paths to which they are already mapped, effectively reusing their tags.* This eliminates the expensive position map updates for these ORAMs. Reusing the tags in all the other sub-ORAMs is an important stepping stone for achieving better efficiency, because the $(\ell+1)$ BatchEvict's will now only need to update the position map for R_i. We note that a similar observation has been made in a different context of constructing a PIR-based 2-server ORAM [43].

Unfortunately, this is not enough to reduce the position map lookups because while tags are only being updated in one of the sub-ORAMs, a position map look up is required in all the other sub-ORAMs to determine the existing tags of the data being evicted as part of the BatchEvict. As a result, the number of seeks and communication remain $\mathbb{O}(\log^3 N)$.

[2]Note that the position map for each of the range ORAMs only needs to store the start position of the range as subsequent positions can be calculated by incrementing in bit-reversed order. As a result, larger range ORAMs have significantly smaller position maps that may not need to be stored recursively, but the position maps of small range ORAMs are the worst case in the analysis.

Pointer-based oblivious data structures. To solve the problem of updating multiple ORAMs where data may be duplicated, we propose a new distributed position map construction leveraging pointer-based oblivious data structure techniques, initially introduced by Wang et al. [44] and subsequently used by several ORAM solutions [15, 35, 36, 44]. In particular, alongside each physical block we store *the path tag of that block in all ORAMs*, as shown in Figure 4.

As an example, to query a a range of length 2 at logical addresses a and $b = a + 1$ the following procedure is used:

1) Refer to the position map of R_1 and obtain the path tag p_1 of a in R_1.
2) Read the two consecutive physical paths (according to the reversed-bit order) based on p_1 in R_1. Let

$$(d_a, a, p_0, p_1, \dots, p_\ell), \quad (d_b, b, q_0, q_1, \dots, q_\ell)$$

be the two physical blocks retrieved in this stage. Here p_j (resp., q_j) denotes the path tag for address a (resp., b) in ORAM R_j.

3) Choose p'_1 at random. Compute q'_1 to be next to p'_1 according to the reversed-bit order. Let d'_a, d'_b be the updated data.
4) Update the position map of R_1 so that the path tag of a should be p'_1.
5) For $i = 0, \dots, \ell$:
Push the following two blocks in the stash for R_i:
$(d'_a, a, p_0, p'_1, p_2 \dots, p_\ell), \quad (d'_b, b, q_0, q'_1, q_2 \dots, q_\ell)$.
Then, execute R_i.BatchEvict(2).

Note that the above procedure uses only a single position-map access (i.e., for R_1) in order to identify the path tag p_1, which needs $\mathbb{O}(\log^2 N)$ seeks. The path tags in other ORAMs were obtained from the retrieved physical blocks and then reused in BatchEvict, which requires $\mathbb{O}(\log^2 N)$ seeks as well. Consequently, rORAM only requires $\mathbb{O}(\log^2 N)$ seeks in total.

Handling duplicates. One thing to note here is that after the required range has been read from R_i, it is evicted back to all ORAMs $R_0, \dots, R_\ell$ and so there must be a process for handling duplicates. Since we do not read blocks in the range from R_j but add copies during the batched eviction, a block may have multiple copies in the tree that need to be removed during subsequent evictions.

This, however, is not a problem. Since the path tag will be reused in R_j, its old copy will also be along the same path that includes a newer copy and will be lower down in the tree. Thus, when the path is retrieved during an eviction the duplicate blocks in the lower level would be recognized as older and safely overwritten.

V. Formal Description

Position map and stash. rORAM requires two supporting data structures, the position map and the stash, similar to Path ORAM. Each sub-ORAM in rORAM has a position map similar to the position map for existing tree-based ORAMs with a couple of modifications:

- Instead of mapping a block ID (i.e., logical address) to a leaf identifier (i.e., physical location), a *range ID* is mapped to a leaf label.
- The leaf label stored for a range corresponds to the leaf to which *the first block in the range* is mapped. This is enough since once the leaf label of the first block is known, the leaf labels for the remaining blocks can be easily determined due to the locality-sensitive mapping.

Depending on the setting, each position map is stored either on the client-side or on the server side (in a recursive ORAM or in an oblivious trie). rORAM also stores a stash for each ORAM on the client-side to handle overflows from the tree.

Notations and parameters. Let N be the number of logical blocks that rORAM stores, and L be the maximum range size the rORAM needs to support. Let $\ell = \lceil \log_2 L \rceil$. Then, rORAM has $\ell + 1$ ORAMs $R_0, R_1, \dots, R_\ell$.

Let $h = \lceil \log_2 N \rceil$ denote the height of each ORAM tree R_i. A bucket label v_i^r signifies the rth bucket among those at level i in bit-reversed order. In an ORAM tree T, let $P_T(v_h^r)$ be a path from the root to a leaf v_h^r; we will often omit subscript T if obvious from the context. Note that the following property holds in an ORAM tree:

$$P(v_h^r) = \{v_j^{r \bmod 2^j} : j = 0, \dots, h\}.$$

In the algorithm descriptions, we use V_j to refer to the set of nodes on level j among the currently-considered paths.

Let PM_i and stash_i denote the position map and stash for ORAM R_i. Let cnt be a global integer variable, initially 0, which is used to track the deterministic eviction schedule according to the bit-reversed order.

A physical bucket $(d, a, p_0, \dots, p_\ell)$ is valid if every p_j falls in the valid range $[0, N)$. Let Z be the number of physical blocks that a bucket v_i^j contains. Dummy (invalid) blocks are used to pad buckets to approriate size in case the bucket contains less than Z real data blocks.

ReadRange. The ReadRange operation for ORAM R_i is described in Algorithm 1, and it returns the *result* set of blocks with position meta-data as well as a new path position, p' for the start address a. The operations performs three tasks:

1) Query the position map to determine the leaf label to which the first block in the range is mapped (Step 3).
2) Update the position map with a new leaf label for the first block in the range (Steps 4-5).
3) Retrieve the buckets along the paths to which the blocks of the range are mapped, *level by level* while scanning for the required blocks (Steps 6–9). Note that the if-statement on Step 9 handles the duplicates by ignoring older blocks on lower levels.

Algorithm 1 R_i.ReadRange(a)

```
1: Let $U := [a, a + 2^i)$.
2: $result \leftarrow$ Scan $\mathsf{stash}_i$ for blocks in range $U$.
3: $p \leftarrow \mathsf{PM}_i.\mathsf{query}(a)$    // Get the leaf label $p$ for address $a$
4: $p' \leftarrow [0, N)$    // random leaf label $p'$
5: $\mathsf{PM}_i.\mathsf{update}(a, p')$ // Update the position map for address $a$
6: for $j = 0, \dots, h$ do
7:     Read the ORAM buckets $V = \{v_j^{t \bmod 2^j} : t \in [p, p + 2^i)\}$.
8:     for each valid block $B = (d, a, p_0, \dots, p_\ell)$ in $V$ do
9:         if $B.a \in U$ and $B \notin result$ then $result \leftarrow result \cup \{B\}$
10: return $(result, p')$
```

BatchEvict. The BatchEvict operation is described in Algorithm 2. The operation performs three tasks:

1) Read the buckets from the server along the next k eviction paths *level by level* (Steps 1-5).
2) Evict blocks locally to the eviction paths (Steps 6-11).
3) Write back the updated buckets read to the tree in the *level-by-level* manner (Steps 12-13).

Algorithm 2 R_i.BatchEvict(k)

```
    // cnt: a global integer variable tracking the eviction schedule
    // $h = \log N$: the height of the ORAM tree.
    // Fetch buckets from server
 1: for $j = 0, \ldots, h$ do
 2:     Read ORAM buckets $V_j = \{v_j^{t \bmod 2^j} : t \in [\mathsf{cnt}, \mathsf{cnt}+k)\}$.
 3:     for each valid block $B = (d, a, p_0, \ldots, p_\ell)$ in $V$ do
 4:         if $\mathsf{stash}_i$ has no block with address $B.a$ then
 5:             $\mathsf{stash}_i \leftarrow \mathsf{stash}_i \cup \{B\}$
    // Evict paths and write buckets back to server
 6: for $j = h, \ldots, 0$ do // Evicting paths: bottom-up, level-by-level
 7:     for $r \in \{t \bmod 2^j : t \in [\mathsf{cnt}, k+\mathsf{cnt})\}$ do // For each path
 8:         $S' \leftarrow \{(d, a, p_0, \ldots, p_\ell) \in \mathsf{stash}_i : p_i \equiv r \pmod{2^j}\}$
 9:         $S' \leftarrow$ Select $\min(|S'|, Z)$ blocks from $S'$
10:         $\mathsf{stash}_i \leftarrow \mathsf{stash}_i \ / \ S'$
11:         $v_j^{r \bmod 2^j} \leftarrow S'$.
    // Write back buckets to server
12: for $j = 0, \ldots, h$ do
13:     Write the ORAM buckets $\{v_j^{t \bmod 2^j} : t \in [\mathsf{cnt}, \mathsf{cnt}+k)\}$.
```

Access protocol in rORAM. We are ready to give the formal description of the Access protocol of rORAM. The protocol supports any range of size $r \le L$ starting at any given addres $a \in [0, N-r)$. As explained in Section IV, this will be partitioned into two ranges of size $\lceil \log_2 r \rceil$.

The Access protocol, described in Algorithm 3, takes the following input: a the start address of the range; r is the size of the range; *op* is the operation, either *read* or *write*; and D^* the new data, optionally, to be updated during a write for data in the range. The operation is performed in two main tasks, each performed twice to cover arbitrary ranges obliviously:

1) Perform two ReadRanges on the first/second half of the range, retrieve data, and update positions (Steps 4–7).
2) Perform a BatchEvict by updating the each ORAM's stash with the new data (Steps 10-13). Note that Step 11 is necessary to first remove any old "stale" data from the stash with the same address as one in the range.

On a write, the data is updated between these steps (Steps 8–9). On a read, the values fetched within the requested range are returned at the end (Step 15).

VI. Analysis

Correctness and obliviousness. Correctness of our protocol follows by inspection. Obliviousness, with leakage of the length of the given range, holds from the following facts:

- All data items exchanged over the network are encrypted with IND-CPA secure encryption.
- ReadRange: We choose ORAM R_i based only on the length of the range. In ORAM R_i, the paths selected for reading do not reveal any information to the adversary other than the fact that two ReadRange operations occurred on R_i.
- BatchEvict has a deterministic schedule.

Algorithm 3 Access(a, r, op, D^*)

```
 1: Let $i \in [0, \ell)$ such that $2^{i-1} < r \le 2^i$
 2: Let $a_0 = \lfloor a/2^i \rfloor \cdot 2^i$
 3: $D \leftarrow \{\}$
    // Perform two ReadRanges to cover the range $[a, a+r)$
 4: for $a' \in \{a_0, a_0 + 2^i\}$ do
 5:     $(B_{a'}, \ldots, B_{a'+2^i-1}, p') \leftarrow R_i.\mathsf{ReadRange}(a')$
 6:     for $j \in [0, 2^i)$ do
 7:         $B_{a'+j}.p_i \leftarrow p' + j$ // update positions for all blocks
    // Update data if writing
 8: if $op$ = "write" then
 9:     for $j \in [a, a+r)$ do $B_j.d \leftarrow D_j^*$
    // Update stashes and evict in each tree
10: for $j = 0, \ldots, \ell$ do
11:     $\mathsf{stash}_j \leftarrow \mathsf{stash}_j \setminus \{B \in \mathsf{stash}_j : a_0 \le B.a < a_0 + 2^{i+1}\}$
12:     $\mathsf{stash}_j \leftarrow \mathsf{stash}_j \cup \{B_{a_0}, \ldots, B_{a_0+2^{i+1}-1}\}$
13:     $R_j.\mathsf{BatchEvict}(2^{i+1})$
14: $\mathsf{cnt} \leftarrow \mathsf{cnt} + 2^{i+1}$
15: if $op$ = "read" then return $D$
```

Only the second item, on ReadRange, warrants some additional explanation. Recall that every read in ORAM R_i will be a block of 2^i consecutive positions in the bit-reversed order. An adversary therefore learns from each ReadRange on R_i the first position of the range. But this first position is chosen at random, then invalidated and re-assigned randomly after each time it is revealed. Therefore the adversary learns nothing from this observation.

Bandwidth and locality. Note each Access(a, r, op, D^*) performs ReadRange twice and BatchEvict $(\ell + 1)$ times.

- ReadRange: The position map access (Steps 3-5) needs $\mathbb{O}(\log^2 N)$ seeks and bandwidth. As to reading the paths (Steps 6-9), we need $\mathbb{O}(r \log N)$ bandwidth, since $\mathbb{O}(r)$ paths are retrieved with each path having $\mathbb{O}(\log N)$ buckets. For locality, thanks to the bit-reversed disk layout, reading buckets in a given level (Step 7) takes at most 2 seeks, which implies that $\mathbb{O}(\log N)$ seeks are necessary in total. Overall, we have
  - Bandwidth: $\mathbb{O}(\log^2 N + r \log N)$
  - Locality: $\mathbb{O}(\log^2 N + \log N) = \mathbb{O}(\log^2 N)$.
- BatchEvict: It performs reading and writing $\mathbb{O}(r)$ paths. By applying the argument right above, we have:
  - Bandwidth: $\mathbb{O}(r \log N)$
  - Locality: $\mathbb{O}(\log N)$.

Access has bandwidth $\mathbb{O}(r \log^2 N)$ and locality $\mathbb{O}(\log^2 N)$.

Stash analysis. Consider ORAM R_i in our construction and let L_i be the length of a range in R_i; that is, we have $L_i = 2^i \le N$. The following theorem shows that the size of stash is stabilized around $L_i \cdot \lambda$, where λ is the security parameter. We note that this bound is essentially the same as that in the previous work [9], where ORAM R_i is a usual tree-based ORAM whose block size is large enough for a block to contain a range of size L_i entirely; therefore, the size of the stash for R_i therein has the same bound.

Theorem 1. *Suppose ORAM R_i has bucket size $Z \ge 3$. Let $\mathsf{st}(R_i)$ be the number of blocks in the stash after a sequence of operations in ORAM R_i. Then, for $L_i \le N/4$,*

$$\Pr[\mathsf{st}(R_i) > L_i \cdot (\lambda + 1)] < 3.5 \cdot L_i \cdot Z^{-\lambda}.$$

Since $L_i \leq N/4$ is independent of λ, the probability above decreases exponentially in λ.

Proof intuition. Observing Figure 2, we can identify an interesting property:

> *All labels with an even (resp., odd) number belong to the left (resp., right) half.*

Going further, let T_0, T_1, T_2, T_3 be subtrees in Figure 2, each containing two leaf nodes such that T_k is rooted with node v_2^k. Observe that each subtree T_k contains all leaf nodes v_3^j such that $j \equiv k \pmod 4$.

This property provides the ORAM with an interesting partitioning power. In particular, consider a length-4 range $(a, a+1, a+2, a+3)$; this range will be assigned some leaf labels $(r, r+1, r+2, r+3)$, where r is chosen randomly. Then, blocks $a, a+1, a+2, a+3$ will belong to $T_{r \bmod 4}, T_{(r+1) \bmod 4}, T_{(r+2) \bmod 4}, T_{(r+3) \bmod 4}$ respectively. In other words, each T_k will have *exactly one block*, no more or no less, from the range.

Therefore, in general, in ORAM R_i, we will have 2^i subtrees, each of which behaves as a single block ORAM. A union bound over these 2^i subtrees proves the above theorem. The complete proof is found in Appendix **??**.

VII. Empirical Measurement

A. Implementation Details

rORAM is implemented on top of a publicly available Java library [2, 10] that provides optimized implementations of several well-known ORAM schemes, including Path ORAM. The implementation requires about 2000 LOC and is publicly available on github [4] (currently redacted for submission).

First range ORAM implementation. To the best of our knowledge, this is the *first implementation and evaluation of a range ORAM construction* since previous theoretical constructions [9, 21] are non-trivial to implement and have not been empirically verified. *More importantly, these constructions are asymptotically less efficient (by a factor of $\mathbb{O}(\log N)$) for both seeks and bandwidth compared to rORAM (Table I).*

In fact, notwithstanding the complexity of implementation, due to the high bandwidth overhead (which is often times more expensive than seeks), it is not immediately clear whether these construction provide any real speedup over bandwidth-optimized standard ORAMs on hardware.

Instead, we use Path ORAM as a comparison point. While Path ORAM is not optimized for seek performance, it does provide a good baseline and the addition of the locality-aware physical layout with a relaxed security definition does indeed result in a more efficient native Path ORAM construction.

Data layout. The layout of data on disk requires careful consideration in the measurements. Prior work primarily focused on performance metrics related to communication and computation, and as such, may have used layouts that failed to expose the costs of disk seeks. For example, a standard data layout for evaluation is to store tree-based ORAM's in a series of individual, smaller files, e.g., one bucket per file. While this layout eliminates the costs of seeks within files—each file/bucket is read in a single seek access—the measurement of a query time will then mostly capture the costs of computation and accesses of local memory but not the costs of seeks. Additionally, this storage technique is prohibitively expensive or impossible for larger databases. A 4 TB ORAM (including the position map) would require more than 2^{32} files, exceeding the number of that can be allocated in a `ext4` file system with 32-bit inode labeling [3]. To better capture the impact of seeks using a more realistic setup, we store the entire rORAM in multiple 1GB files.

Platform. All benchmarks were performed on Linux installation with Intel Core i7-3520M processors running at 2.90GHz and 8GB+ of DDR3 DRAM. The devices of choice were:

1) **Local Hard Drive**: 1TB IBM 43W7622 SATA HDD running at 7200 RPM. The average seek time and rotational latency of the disk is 9ms and 4.17ms respectively. The data transfer rate is 300MB/s.
2) **Local Solid State Drive**: 1 TB Samsung-850 Evo SSD.
3) **Network Block Device**: 1TB Amazon EBS [1] volume (cold storage HDD) mounted as an iSCSI device [5]. The network bandwidth between the local client and the t2.large Amazon instance hosting the EBS volume is measured to be around 40 - 60MBps using *iperf* [6].

B. Measurement Techniques

Setup. A 16GB database size is used for evaluation (2^{22} blocks of 4KB each). We instantiate Path ORAM with a recursively stored position map and a locally stored stash of size set for 128-bits of security according to [25]. The rORAM setup supports a maximum range size of $L = 2^{14}$. Each test comprises of 5 trials with a new random permutation to initialize the ORAMs. Results are collected with a 95% confidence interval.

Metrics. The main metrics for evaluation are *query access time* and *overall query throughput*. As noted in [10], high query access times for logically related queries (such as a range) is the major bottleneck for synchronous ORAMs. This forces applications to wait indefinitely while multiple logically related blocks are fetched individually, one at a time. *rORAM solves this problem by allowing range queries for multiple logically related data blocks.*

Traditionally, ORAMs are evaluated based on the average time required to complete a query (query access/response time). The query access time is measured as the time elapsed between the time when a query (for any range size) was initiated and the time when the query was finally completed. For tests, we generate random queries (of different sizes) at a steady rate and measure the clock-time required to complete these queries. The next query is issued only when the previous one has completed. Note by design Path ORAM supports only synchronous query processing. Asynchronous versions of Path ORAM [37] can easily replace the synchronous versions used in our implementation.

C. Query Access Time

Locality-aware disk layout and batched evictions. Batching evictions for standard Path ORAM equipped with the locality-aware disk layout, improves performance even without the addition of range functionality. We evaluate the average query

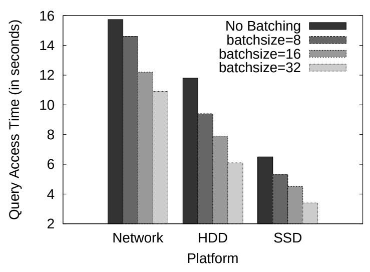


Fig. 5. Average query access time per block (lower is better) for Path ORAM with batched evictions based on the locality-aware disk layout. The no batching Path ORAM is the standard algorithm whereby only one eviction occurs per access. The batched Path ORAM is where b evictions (the batch size) are all done together and deterministically following b queries. We obtain 2x improvement in average query response times for HDDs. Similar improvements are observed for SSDs and network devices.

access time for Path ORAM with a deterministic eviction schedule (based on the locality-aware disk layout) performed in batches. The results are presented in Figure 5. As expected, optimizing total number of seeks reduces query access times.

For the local HDD, the locality-aware physical layout results in a 2x improvement in average query response times. Interestingly, batching evictions is helpful even for local SSDs and network devices since it optimizes I/O requests/round-trips.

Range queries. As a measurement of the range functionality, we measure the query access time to query ranges of varying sizes for regular Path ORAM, Path ORAM with batched evictions (batch size = 32) and rORAM (Figure 6). Note that the values on the X-axis are range size exponents. The query access time for a range of size 2^x can be determined by multiplying the Y-axis value corresponding to x with 2^x. E.g., the total query time for a range of size 2^6 for Path ORAM on the local HDD (Figure 6 (a)) is around 10×2^6 seconds. For all cases, the query access time for Path ORAM and Path ORAM with batched evictions increases linearly with the range size. Batching evictions generally improves performance by reducing the overall number of seeks.

For smaller ranges, Path ORAM performs better than rORAM, since the cost of rORAM dealing with multiple sub-ORAMs is dominant in this regime. For all platforms, rORAM performs better than Path ORAM for ranges of size 2^5 and more. For the local HDD, rORAM is almost 30x faster than Path ORAM for range sizes $\geq 2^{10}$ (which corresponds to 4MB of logically sequential data) and 50x faster than Path ORAM for even larger ranges of size $> 2^{12}$ (16MB of sequential data). This is the result of optimizing seeks and reducing overall I/O since rORAM accesses larger chunks of data with a single request compared to a large number of requests generated in case of Path ORAM.

In fact, the reduction in I/O requests makes rORAM faster than Path ORAM even for SSDs (around 30x) and network block devices (around 10x). As noted previously [36], ensuring locality of accesses improves performance on SSDs while the reduced number of round-trips required to fetch all blocks in a range makes rORAM faster for network block devices.

D. Query Throughput & Application Benchmarks

Although rORAM is primarily designed for range query applications, the construction can also be used to speedup applications that have largely sequential access patterns. Logically sequential blocks can be fetched as a range. Specifically, the application, e.g., a file system generates requests specifying the total number of bytes/blocks it wants to consecutively access from the memory. This is translated into a range query of appropriate size rounding up to a power of 2.

To measure this effect, we assess query throughput for several real world workloads. Specifically, similar to [10], we replay access traces of several applications – sequential reads, file server and video server workloads used by FileBench version 1.4.9.1 – and measure the corresponding query throughput. To generate the trace, we first log all requests generated by FileBench and replay these requests to both Path ORAM and rORAM. Since file systems typically break down an access to a large sequential chunk accesses into smaller sequential chunks not exceeding 1MB in size it suffices to initialize rORAM with $L = 2^8$ blocks.

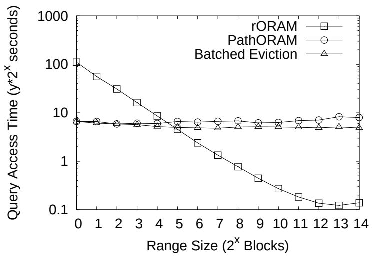
Sequential reads. The sequential read workload generates requests for sequential reads of size 8MB over a large (10GB) file, interleaving a small number of random reads/writes in between[3]. For the local HDD (Figure 7 (a)) and the local SSD (Figure 7 (b)), rORAM can support up to 10 and 21 queries per second respectively. This is almost 20x improvement over the query throughput of Path ORAM. Note that the overall query throughput of Path ORAM and Path ORAM with batched evictions remains largely unaffected by sequential accesses since each query is treated as a query for a random block, regardless of sequentiality. For the network block device, the overall query throughput increases but plateaus as larger ranges throttle the available bandwidth.

File server. In order to evaluate rORAM for real world applications, we used the file server workload of FileBench. The workload generates accesses similar to a regular file system and closely resembles the SPECFsfs benchmark suite [7]. In particular, read and write requests are generated for files of variables sizes, while also updating corresponding metadata. rORAM handles accesses of variable sizes well and shows a 5x increase in overall throughput compared to Path ORAM for the local HDD (Figure 7 (b)). Similar trends are observable for both the SSD and network device scenarios (Figure 7 (b,c)).

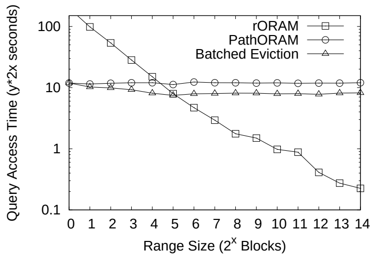
Video server. A more appropriate benchmark for range ORAM applications is a video server that deploys multiple threads to fetch large sequential chunks of streaming data. In this case, the large sequential requests can be performed as range queries. Additionally, the application performs writes of variable size to metadata and inactive video files. Note that a naive solution of storing and accessing files in large sequential chunks, to ensure small number of seeks, will waste significant I/O while updating the metadata (often small in size) well.

Since, rORAM allows range queries of arbitrary sizes, variable-sized sequential accesses are handled well. As as result, rORAM features a 11x increase in query throughput

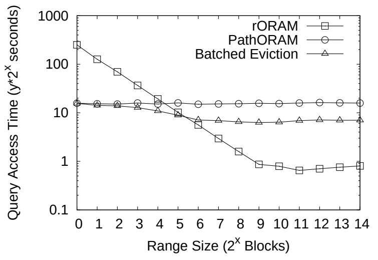
[3]Since reads and writes are equivalent for ORAMs, we expect similar results for sequential writes as well



(a) Query access time on local SSD

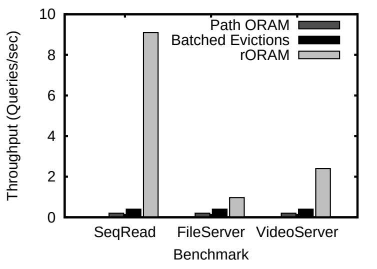


(b) Query access time on local HDD

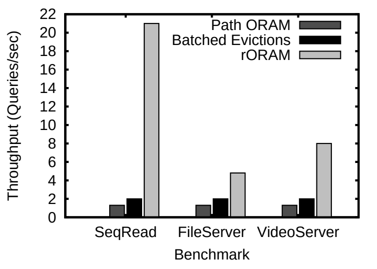


(c) Query access time on Network disk

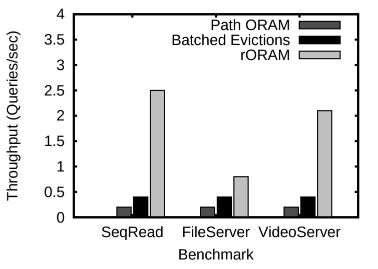
Fig. 6. Average query access time per block (lower is better). Database size = 2^{22} 4kB blocks (16GB). "Batched Evictions" refers to a Path ORAM variant equipped with the locality-aware layout for efficiently batching evictions. For the local HDD, rORAM is 30-50x faster than Path ORAM for ranges sizes $\geq 2^{10}$. rORAM is faster by almost 20-30x for local SSDs and 10x faster for network block devices.



(a) Query Throughput (HDD)



(b) Query Throughput (SSD)



(c) Query Throughput (Network)

Fig. 7. Query throughput (higher is better). Database size = 2^{22} 4kB blocks (16GB). "Batched Evictions" refers to a Path ORAM variant equipped with the locality-aware layout for efficiently batching evictions. For purely sequential workloads, rORAM is 10-15x faster than Path ORAM for the local HDD and SSD. rORAM is almost 6x faster for network block devices. rORAM is up to 5x faster running a file server and up to 11x faster running a video server compared to Path ORAM.

over Path ORAM for the local HDD (Figure 7 (a)), an 8x increase in throughput for the SSD (Figure 7(b)), and 4x increase in throughput for the network device (Figure 7(c)).

Client-side storage. For the storage configurations used in our experiments, rORAM will require (in the worst-case) approximately 8GB of client-side stash when $L = 2^{14}$ blocks (combined stash size for all 15 sub-ORAMs) and 128MB when $L = 2^8$ blocks (Section VI). Note that the stash size only depends on L and is independent of the total outsourced storage size. Empirical observations show that in the average-case, a smaller stash size may suffice in practice. E.g., in our experiments, the maximum observed stash occupancy was around 2^{14} blocks (64MB) and 2^{10} blocks (4MB) for $L = 2^{14}$ and $L = 2^8$ blocks respectively.

VIII. System tweaks and optimizations

The described Range ORAM construction is designed with the main goal of minimizing the number of disk seeks per operation in the general setting of client/server ORAMs with limited client storage. In practice, there are a number of other parameters or settings which the client may alter to allow further improvements. In this section, we briefly outline a few of these alternations and tweaks.

A. Parallel Seeks with Multiple Heads or Disks

Modern storage systems may have multiple read/write heads (a high-capacity HDD disk has up to 8) or use arrays of high capacitive disks that may be striped (e.g., using a RAID). Such configurations, where seeks can occur in parallel, can lead to significant performance gains, and rORAM can leverage these situations with limited modification.

Assume that if the server's storage is partitioned into k equal-sized parts (disk platters or cluster nodes), and that each part can be read or written separately in parallel, it can be shown that the number of parallel seeks per access is $\mathbb{O}\left(\log N \cdot \left(1 + \frac{\log N}{k}\right)\right)$. That is, perfect parallel speedup in the number of seeks is possible for $k \leq \log N$. This improvement is achieved by observing that an access for a range of length r consists of essentially three stages:

1) 2 position map accesses in the target ORAM tree
2) 2 range-r read operations in the target ORAM tree
3) r batch evictions in each of the $\mathbb{O}(\log N)$ ORAM trees.

Step 2 already incurs only $\mathbb{O}(\log N)$ seeks which fits the bound stated above. For Step 3, roughly $(\log N)/k$ ORAM trees are stored on each of the k disks which allows for batch evictions to occur in parallel, meeting the stating bound.

The position map access (Step 1) is more challenging due to the recursion which must occur *sequentially* because the path to access in the next smaller recursive ORAM is only revealed once the path in the larger ORAM has been accessed, leading to $\mathbb{O}(\log^2 N)$ seeks. However, retrieving each of the buckets along the path is deterministic and can be completed using parallel seeks by distributing the levels of the recursive ORAMs across k disks. Fetching a single path at a single recursive level incurs $\mathbb{O}((\log N)/k)$ parallel seeks, and repeating this sequentially for each of the recursive levels gives the cost stated above.

B. Reducing position map costs

Parallel seeks can improve the performance of the position map, but there are other optimizations for decreasing the cost of the position map and increasing the overall performance of the rORAM if we consider larger client storage scenarios.

$\mathbb{O}(\log N)$ **seeks with larger block sizes.** First observe that larger block sizes improve the performance of a position map because the number of seeks for a single position map access is only $\mathbb{O}(\log^2 N/\log B)$, where B is the size of a block. For example, with 4KB blocks and 1GB total storage, the number of recursive levels in any position map is just 2. More generally, if the block size B is large enough to store N^α pointers for some constant $0 < \alpha < 1$, then the number of seeks per position map operation is only $\mathbb{O}(\log N)$. In this setting, there are $O(1)$ levels of recursion for the position map and the total cost is $O(1)$ with parallel seeks across levels.

Locally-stored position map optimizations. If the position map can be stored locally in persistent storage, it does afford a number of optimizations. The most obvious of these is that a single global position map suffices, rather than one for each tree. The second optimization is that locally-stored position maps can be reduced if smaller ranges are not supported.

A position map for all the $\mathbb{O}(\log N)$ ORAM trees could require $\mathbb{O}(N)$ local storage for the position map, but many of those stored positions are a result of tracking locations in the smaller range trees. The position maps in the larger sub-ORAMs are significantly smaller since only the position of the first block in the range is required to reveal the other blocks due to the bit-reversed position ordering within a range. By eliminating a small portion of the *smaller range* trees, the position map size is dramatically reduced without greatly effecting the functionality of the system. For example, in the situation with 1GB total data split into 4KB blocks, the total storage for a local position map is roughly 11MB. Removing the bottom 3 sub-ORAMs, reasonably requiring that all accesses are on ranges of at least 8 blocks, reduces the global position map size to less than 1MB.

C. Revealing operation type

The security definition for range ORAM requires that any two access patterns with the same range sizes are indistinguishable, hiding the contents, addresses, and operation type of each access. Only the size of the range is leaked. An interesting security/performance tradeoff to consider is relaxing the definition to reveal the operation *type* (read or write) to an observer in addition to the range. Roughly speaking, such a security definition allows for leakage of the *direction* of information flow which may be acceptable in some situations.

If operation type is leaked, we claim that the number of seeks per operation can be reduced to just $\mathbb{O}(\log N)$ without affecting the bandwidth under the following conditions.

1) The position map seek cost is $\mathbb{O}(\log N)$ using some ideas from the previous subsection.
2) The operation type (read or write) is revealed.
3) Each write operation is for a single block at a time.

In particular, such a construction still reads ranges, but only writes single blocks. We argue this scenario is actually quite common and useful; for example, revealing the operation type and limiting updates to one block at a time are quite common in searchable symmetric encryption (SSE) scenarios [13, 16].

$\mathbb{O}(\log N)$ **seeks per read.** For reading a range of size $r = 2^i$, two accesses occur on the ORAM tree R_i and r batch evictions in every tree, but for a read operation the data is not actually modified. If the operation type is revealed, batch evictions on the other R_j where $j \neq i$ other ORAM trees does not need to occur because there is no update to the data blocks, reducing the seek cost to $O(\log N)$.

$\mathbb{O}(\log N)$ **seeks per write.** Consider first that while writing a single item, the R_0 ORAM tree needs to be updated, at a cost of $\mathbb{O}(\log N)$ seeks, and the modified item must also be updated in all the other ORAM trees. If those evictions are performed immediately, the cost would be $\mathbb{O}(\log^2 N)$ seeks. However, because the write was only to a single block, we can delay those evictions by simply appending the updated block to each stash and only performing a *single* batch eviction on one other tree, deterministically. With single item writes, i.e., no range writes, we can achieve $\mathbb{O}(\log N)$ seeks.

Specifically, say the construction contains $\ell \in \mathbb{O}(\log N)$ ORAM trees. Then each single block write always updates the R_0 tree, appends the updated block to all $\ell - 1$ other stashes, and then performs a batch eviction of size $(\ell-1)$ for tree index $(i \bmod (\ell-1))+1$. All three steps — updating R_0, appending to $\ell - 1$ other stashes, and performing a single batch eviction — require $\mathbb{O}(\log N)$ seeks. Furthermore, because each stash is cleared out after $\mathbb{O}(\log N)$ updates, the size of stash for each ORAM tree no more than doubles.

D. Malicious security

The rORAM construction, as described, is secure against an honest-but-curious adversary who always follows the protocols correctly, but may observe and remember all communication and past states of the remote storage. Achieving a higher level of security against a malicious adversary who may actually change the contents of remote storage or otherwise disobey the protocol requires relatively straightforward techniques for ensuring *integrity* [12, 41].

As in prior works, a *Merkle tree* can be embedded within each individual ORAM trees to ensure integrity. However, there is one important difference which is critical for minimizing the number of disk seeks. In a typical Merkle tree, each node stores a combined hash of its two children. However, doing this would require doubling the number of seeks because updating a single tree path requires reading all sibling nodes in the path as well.

Instead, each ORAM tree node stores a *separate* hash of each child node so that updating a path in any of the ORAM trees only requires reading and re-writing the nodes in that path. The extra hashes introduce a (small) constant factor increase in the bandwidth and remote storage size but does not change the number of seeks. The hashes are stored contiguously with the data.

Finally, the individual hashes of all $\mathbb{O}(\log N)$ ORAM trees are collected into a single "root block" of hashes, which is stored contiguously with the root node of any one of the

ORAM trees. Reading the root block on every access does not introduce any extra seeks, and the client only needs to store the hash of this root block locally in persistent storage.

IX. Conclusion

rORAM is an ORAM specifically suited for accessing ranges of *sequential logical blocks* while *minimizing the number of random physical disk seeks*. rORAM is significantly more efficient than prior designs [9], reducing a $\mathbb{O}(\log N)$ multiplicative factor *both* in the number of seeks and in communication complexity.

A rORAM implementation is 30-50x times faster than Path ORAM for similar range-query workloads on local HDDs, 30x faster for local SSDs, and 10x faster for network block devices. rORAM's novel disk layout can also speed up standard ORAM constructions, e.g., resulting in a 2x faster Path ORAM variant. rORAM's novel disk layout can also speed up standard ORAM constructions, e.g., resulting in a 2x faster Path ORAM variant. Importantly, experiments demonstrate suitability for real world applications – rORAM is up to 5x faster running a file server and up to 11x faster running a range-query intensive video server workloads compared to standard Path ORAM.

rORAM raises the significant practical issue of data locality as an important factor in ORAM design. Even for ORAMs that do not naturally support range queries, locality can have a large impact on performance and seek optimization should be a design criteria for future ORAM technology.

X. Acknowledgments

This work is supported by the National Science Foundation under awards 1526707, 1526102, 1319994, 1406177, 1618269 and by the Office of Naval Research. We thank our shepherd, Dimitrios Papadopoulos and the anonymous reviewers for their valuable suggestions and comments.

References

[1] "Amazon elastic block storage," April, 23 2018, https://aws.amazon.com/ebs/.

[2] "Home of the curious framework," April, 23 2018, http://seclab.soic.indiana.edu/curious/.

[3] "Ext4 disk layout," April, 23 2018, https://ext4.wiki.kernel.org/index.php/Ext4_Disk_Layout.

[4] "roram implementation on github," https://github.com/anrinch/rORAM.

[5] "Linux-iscsi project," April, 23 2018, http://linux-iscsi.sourceforge.net/.

[6] "iperf," April, 23 2018, https://iperf.fr/.

[7] "Specsfs benchmark suite," https://www.spec.org/sfs2014/.

[8] G. Asharov, M. Naor, G. Segev, and I. Shahaf, "Searchable symmetric encryption: optimal locality in linear space via two-dimensional balanced allocations," in *48th ACM STOC*, D. Wichs and Y. Mansour, Eds. Cambridge, MA, USA: ACM Press, Jun. 18–21, 2016, pp. 1101–1114.

[9] G. Asharov, T.-H. H. Chan, K. Nayak, R. Pass, L. Ren, and E. Shi, "Oblivious computation with data locality," Cryptology ePrint Archive, Report 2017/772, 2017, http://eprint.iacr.org/2017/772.

[10] V. Bindschaedler, M. Naveed, X. Pan, X. Wang, and Y. Huang, "Practicing oblivious access on cloud storage: the gap, the fallacy, and the new way forward," in *ACM CCS 15*. ACM Press, Oct. 12–16, 2015, pp. 837–849.

[11] E.-O. Blass, T. Mayberry, G. Noubir, and K. Onarlioglu, "Toward robust hidden volumes using write-only oblivious ram," in *CCS*, 2014, pp. 203–214.

[12] E.-O. Blass, T. Mayberry, and G. Noubir, "Multi-client oblivious RAM secure against malicious servers," in *ACNS 17*, ser. LNCS, vol. 10355. Springer, Heidelberg, Germany, 2017, pp. 686–707.

[13] C. Bösch, P. Hartel, W. Jonker, and A. Peter, "A survey of provably secure searchable encryption," *ACM Comput. Surv.*, vol. 47, no. 2, pp. 18:1–18:51, Aug. 2014.

[14] D. Cash and S. Tessaro, "The locality of searchable symmetric encryption," in *EUROCRYPT 2014*, ser. LNCS, vol. 8441. Copenhagen, Denmark: Springer, Heidelberg, Germany, May 11–15, 2014, pp. 351–368.

[15] A. Chakraborti, C. Chen, and R. Sion, "DataLair: Efficient block storage with plausible deniability against multi-snapshot adversaries," *Proceedings on Privacy Enhancing Technologies*, vol. 2017, pp. 175–193, Jul. 2017.

[16] R. Curtmola, J. A. Garay, S. Kamara, and R. Ostrovsky, "Searchable symmetric encryption: improved definitions and efficient constructions," in *ACM CCS 06*, A. Juels, R. N. Wright, and S. Vimercati, Eds. Alexandria, Virginia, USA: ACM Press, Oct. 30 – Nov. 3, 2006, pp. 79–88.

[17] J. Dean, "Latency numbers every programmer should know," Online, 2018, https://gist.github.com/jboner/2841832.

[18] E. D. Demaine, "Cache-oblivious algorithms and data structures," in *Lecture Notes from the EEF Summer School on Massive Data Sets*, 2002, http://erikdemaine.org/papers/BRICS2002/.

[19] I. Demertzis and C. Papamanthou, "Fast searchable encryption with tunable locality," in *SIGMOD*, 2017, pp. 1053–1067.

[20] I. Demertzis, S. Papadopoulos, O. Papapetrou, A. Deligiannakis, and M. Garofalakis, "Practical private range search revisited," in *SIGMOD '16*, 2016, pp. 185–198.

[21] I. Demertzis, D. Papadopoulos, and C. Papamanthou, "Searchable encryption with optimal locality: Achieving sublogarithmic read efficiency," in *Advances in Cryptology – CRYPTO 2018*, 2018, pp. 371–406.

[22] J. Doerner and A. Shelat, "Scaling ORAM for secure computation," in *ACM CCS 17*, B. M. Thuraisingham, D. Evans, T. Malkin, and D. Xu, Eds. Dallas, TX, USA: ACM Press, Oct. 31 – Nov. 2, 2017, pp. 523–535.

[23] S. Faber, S. Jarecki, H. Krawczyk, Q. Nguyen, M.-C. Rosu, and M. Steiner, "Rich queries on encrypted data: Beyond exact matches," *IACR Cryptology ePrint Archive*, vol. 2015, p. 927, 2015.

[24] 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 *ASPLOS*, 2015, pp. 103–116.

[25] C. W. Fletcher, L. Ren, A. Kwon, M. van Dijk, E. Stefanov, D. N. Serpanos, and S. Devadas, "A low-latency, low-area hardware oblivious ram controller," *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*, pp. 215–222, 2015.

[26] C. Gentry, K. A. Goldman, S. Halevi, C. S. Jutla, M. Raykova, and D. Wichs, "Optimizing ORAM and using it efficiently for secure computation," in *PETS*, 2013, pp. 1–18.

[27] O. Goldreich, "Towards a theory of software protection and simulation by oblivious RAMs," in *19th ACM STOC*, A. Aho, Ed. New York City, NY, USA: ACM Press, May 25–27, 1987, pp. 182–194.

[28] M. S. Islam, M. Kuzu, and M. Kantarcioglu, "Access pattern disclosure on searchable encryption: Ramification, attack and mitigation," in *in Network and Distributed System Security Symposium (NDSS)*, 2012.

[29] C. Liu, A. Harris, M. Maas, M. W. Hicks, M. Tiwari, and E. Shi, "Ghostrider: A hardware-software system for memory trace oblivious computation," in *ASPLOS*, 2015, pp. 87–101.

[30] C. Liu, X. S. Wang, K. Nayak, Y. Huang, and E. Shi, "ObliVM: A programming framework for secure computation," in *2015 IEEE Symposium on Security and Privacy*. San Jose, CA, USA: IEEE Computer Society Press, May 17–21, 2015, pp. 359–376.

[31] T. Mayberry, E.-O. Blass, and A. H. Chan, "Efficient private file retrieval by combining ORAM and PIR," in *NDSS 2014*. San Diego, CA, USA: The Internet Society, Feb. 23–26, 2014.

[32] I. Miers and P. Mohassel, "IO-DSSE: scaling dynamic searchable encryption to millions of indexes by improving locality," in *NDSS*, 2017.

[33] O. Ohrimenko, M. T. Goodrich, R. Tamassia, and E. Upfal, "The melbourne shuffle: Improving oblivious storage in the cloud," in *ICALP*, 2014, pp. 556–567.

[34] L. Ren, C. W. Fletcher, A. Kwon, E. Stefanov, E. Shi, M. van Dijk, and S. Devadas, "Constants count: Practical improvements to oblivious RAM," in *USENIX Security 15*, 2015, pp. 415–430.

[35] D. S. Roche, A. J. Aviv, and S. G. Choi, "A practical oblivious map data structure with secure deletion and history independence," in *2016 IEEE Symposium on Security and Privacy*. San Jose, CA, USA: IEEE Computer Society Press, May 22–26, 2016, pp. 178–197.

[36] D. S. Roche, A. J. Aviv, S. G. Choi, and T. Mayberry, "Deterministic, stash-free write-only ORAM," in *ACM CCS 17*, B. M. Thuraisingham, D. Evans, T. Malkin, and D. Xu, Eds. Dallas, TX, USA: ACM Press, Oct. 31 – Nov. 2, 2017, pp. 507–521.

[37] C. Sahin, V. Zakhary, A. E. Abbadi, H. Lin, and S. Tessaro, "Taostore: Overcoming asynchronicity in oblivious data storage," in *2016 IEEE Symposium on Security and Privacy (SP)*, May 2016, pp. 198–217.

[38] E. Shi, T.-H. H. Chan, E. Stefanov, and M. Li, "Oblivious RAM with $O((\log N)^3)$ worst-case cost," in *ASIACRYPT 2011*, ser. LNCS, vol. 7073. Springer, Heidelberg, Germany, Dec. 4–8, 2011, pp. 197–214.

[39] E. Stefanov and E. Shi, "ObliviStore: High performance oblivious distributed cloud data store," in *NDSS 2013*. San Diego, CA, USA: The Internet Society, Feb. 24–27, 2013.

[40] E. Stefanov, E. Shi, and D. X. Song, "Towards practical oblivious RAM," in *NDSS 2012*. San Diego, CA, USA: The Internet Society, Feb. 5–8, 2012.

[41] 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 *ACM CCS 13*. Berlin, Germany: ACM Press, Nov. 4–8, 2013, pp. 299–310.

[42] X. Wang, T.-H. H. Chan, and E. Shi, "Circuit ORAM: On tightness of the Goldreich-Ostrovsky lower bound," in *ACM CCS 15*. Denver, CO, USA: ACM Press, Oct. 12–16, 2015, pp. 850–861.

[43] X. Wang, D. Gordon, and J. Katz, "Simple and efficient two-server oram," in *Asiacrypt*, 2018, pp. 141–157.

[44] X. S. Wang, K. Nayak, C. Liu, T.-H. H. Chan, E. Shi, E. Stefanov, and Y. Huang, "Oblivious data structures," in *ACM CCS 14*. Scottsdale, AZ, USA: ACM Press, Nov. 3–7, 2014, pp. 215–226.

[45] P. Williams and R. Sion, "Single round access privacy on outsourced storage," in *ACM CCS 12*, T. Yu, G. Danezis, and V. D. Gligor, Eds. Raleigh, NC, USA: ACM Press, Oct. 16–18, 2012, pp. 293–304.