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

Anrin Chakraborti
Stony Brook University
Stony Brook, New York, U.S.A
anchakrabort@cs.stonybrook.edu

Adam J. Aviv
United States Naval Academy
Annapolis, Maryland, U.S.A
aviv@usna.com

Seung Geol Choi
United States Naval Academy
Annapolis, Maryland, U.S.A
choi@usna.edu

Travis Mayberry
United States Naval Academy
Annapolis, Maryland, U.S.A
mayberry@usna.edu

Daniel S. Roche
United States Naval Academy
Annapolis, Maryland, U.S.A
roche@usna.edu

Radu Sion
Stony Brook University
Stony Brook, New York, U.S.A
sion@cs.stonybrook.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 shuffling the data into random positions such that the storage device is not sure where individual blocks are located, resulting in an access pattern on the device which is highly randomized. However, storage devices are usually optimized for *sequential* accesses, meaning that ORAMs can often induce a substantial overhead (in addition to their increased bandwidth) due to large numbers of disk seeks [13].

In this paper, we present an ORAM construction specifically suited for accessing ranges of sequential logical blocks while minimizing disk seeks. Our construction obtains better asymptotic efficiency than prior work with similar security guarantees [8], achieving $\mathbb{O}(r \cdot \log^2 N)$ communication cost (where r is the size of the range) and $\mathbb{O}(\log^2 N)$ seeks per access, regardless of r. This is an improvement of more than a $\mathbb{O}(\log N)$ factor in both metrics when compared to prior work. In evaluation, we find that our construction is 30-50x times faster than Path ORAM for similar workloads on local HDDs, almost 30x faster for local SSDs, and 10x faster for network block devices.

1 INTRODUCTION

ORAM. An attacker that is capable of viewing the communications or tracking accesses of a user over a data store can reveal a wealth of private information. This can happen even when the data contents are encrypted, because the access patterns (i.e., metadata) may reveal nearly as much as the data contents themselves. Oblivious RAM (ORAM) protocols [32, 52, 53] have been designed to solve this problem by provably making any two access patterns indistinguishable to an adversary observing reads/writes to an untrusted storage device.

The typical theoretical measure of ORAM performance is the communication overhead, or bandwidth, which describes the total number of additional data reads/writes needed to perform a single access. This was the metric used in the seminal work by Goldreich and Ovstrovsky [32]. Since then, other important metrics have been considered such as local computation complexity [38] and round complexity [48]. Numerous ORAM constructions [10, 14, 18, 34–36, 40, 47, 52, 53, 56, 57, 59, 62] have been proposed and studied which seek to optimize these performance measures.

Data locality and ORAM. One measure, which is known to be important but has been largely overlooked in the ORAM literature, is *data locality*. Specifically we refer to data locality as 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 has roughly 100x overhead compared to a single instruction, and the cost overhead of performing a disk seek can be up to 10,000x the cost of reading a sequential block from that disk[21]. This observation has led to the development of efficient data structures and algorithms which improve practical performance by optimizing data locality (see, e.g., [22, 29]).

Unfortunately, the very randomization used by ORAMs to ensure privacy seems to be in direct conflict with data locality. For even a single access, a typical ORAM requires many non-sequential accesses to underlying physical memory. Even worse, an optimized application which makes accesses in consecutive ranges gains no benefit using normal ORAMs, as the physical locations of memory blocks have no correlation with their logical addresses.

Range ORAM (rORAM). Asharov et al. [8] considered this issue and noted that providing data locality in an ORAM for range queries seems impossible on first inspection. By definition, an ORAM must not distinguish between a client requesting r random items or a contiguous region of size r, and an ORAM protocol that provides locality and protects against these two access sequences incurs significant overhead in bandwidth. However, if the security definition is relaxed, as suggested in [8], more efficient solutions are possible. In particular, Asharov et al. showed that range ORAMs can be constructed with $O(\log^3 N \cdot (\log\log N)^2)$ seeks per operation, independent of the length r of the range, provided the additional leakage of the size of each range that is accessed.

Our goal. In this paper, we continue this line of inquiry and significantly improve upon the previous result while asking the same question:

> *Can we construct a more efficient range ORAM scheme that preserves data locality while leaking only the lengths of contiguous regions accessed?*

We show that this question can be answered with a range ORAM construction with $\mathbb{O}(\log^2 N)$ seeks and $\mathbb{O}(r \cdot \log^2 N)$ communication overhead for accessing a range of r consecutive blocks.

	Seeks	Bandwidth	Server Space	Leakage
This work	$\mathbb{O}(\log^2 N)$	$\mathbb{O}(r \cdot \log^2 N)$	$\mathbb{O}(N \log N)$	$\lceil \log_2 r \rceil$
Asharov et al. [8]	$\mathbb{O}(\log^3 N \cdot (\log \log N)^2)$, amortized	$\mathbb{O}(r \cdot \log^3 N)$, amortized	$\mathbb{O}(N \log N)$	$\lceil \log_2 r \rceil$
Ordinary Path ORAM [53]	$\mathbb{O}(r \cdot \log^2 N)$	$\mathbb{O}(r \cdot \log^2 N)$	$\mathbb{O}(N)$	none
Demertzis et al. [23]	$\mathbb{O}(r)$	$\mathbb{O}(r \cdot N^{1/3} \cdot \log^2 N)$	$\mathbb{O}(N)$	none

Table 1: Performance comparisons when accessing a region of r contiguous blocks.

1.1 Highlights of our Solution

In this section, we briefly highlight our solution. See Section 3 for more detailed overview.

Multiple ORAMs each covering a subset of ranges. Our rORAM construction makes use of a $\mathbb{O}(\log N)$ of separate sub-ORAMs, each of which are based loosely on Path ORAM and designed to efficiently support queries of a certain range size, namely in powers of two. A client can access one of the R_i ORAMs to retrieve a contiguous range of 2^i blocks. This reveals the size of the range to an adversary that can observe which ORAM was accessed, but the additional leakage frees up the ORAM design to support a high degree of locality for logically sequential data.

Data locality: smart physical layout and batching. Within each of the ORAMs, data locality is achieved by labeling the paths (i.e., leaf nodes) in the ORAM tree in bit-reversed lexicographic ordering, and then *storing the physical buckets across each level of the tree in this same bit-reversed ordering*. This accomplishes two goals.

- In each of the ORAMs, each range of elements is labeled *adjacently according to the bit-reversed order* and will therefore be *physically adjacent* across levels of the tree, although they are *topologically disparate* from each other in the tree.
- Evictions in the ORAM may be scheduled deterministically in bit-reversed order [31] and thus *batched* together in the size of the range.

As a result, reading from a single ORAM tree requires $\mathbb{O}(\log N)$ seeks and performing $\mathbb{O}(r)$ batched evictions requires $\mathbb{O}(\log^2 N)$ seeks because each level of the ORAM trees are *accessed sequentially* independent of the requested range.

Combining position map and pointer-based technique. In the stateless setting, or when the client has limited local storage, the position maps associating logical to physical addresses in each of the ORAM trees must also be stored obliviously. This is a well-known problem with tree-based ORAMs like Path ORAM. A naïve solution should result in updating each of position maps on each access, resulting in $\mathbb{O}(\log^3 N)$ seeks and communication overhead.

We improve on this by combining the position maps with a pointer-based technique similar to that in Oblivious Data Structures [61]. Namely, alongside each block in each ORAM we store additional information that allows us to lookup the location of that same block in the other ORAMs "for free".

1.2 Our Contributions

Based on our rORAM construction presented herein, we make the following contributions:

(1) A novel oblivious range construction that optimizes data locality with more than an $\mathbb{O}(\log N)$ factor improvement over previous results [8]. See Table 1.

(2) The first construction to make use of bit-reversed lexicographic ordering for physical disk layout in order to achieve batch eviction and data locality in range queries.

(3) A novel construction for maintaining data positions in the rORAM that makes use of both position maps and pointer based techniques across multiple ORAMs.

(4) An open-source implementation of rORAM built on top of a widely available path ORAM implementation [10]. To the best of our knowledge, ours is the first implementation of a Range ORAM construction.

(5) Empirical performance measurements that indicate significant practical improvements compared to Path ORAM. For example, rORAM is 30-50x faster than Path ORAM for range queries of size $\geq 2^{10}$ blocks on local HDDs, almost 30x faster for local SSDs, and 10x faster for network block devices.

1.3 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 [33], many works have focused on improving efficiency and security of ORAM (e.g., [10, 14, 18, 34–36, 40, 47, 48, 52, 53, 56, 57, 59, 62]).

ORAM plays as an important tool to achieve secure cloud storage [44, 54, 55] and secure multi-party computation [26, 37, 42, 43, 59, 63] and secure processors [27, 41, 45]. 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 [11, 49, 58, 61]. The work of [39] considers obliviousness in the P2P content sharing system.

Locality in searchable encryption. Data locality has been a useful metric for evaluating searchable symmetric encryption [9, 16, 25]. 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. [8] 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 [33], 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

$\mathbb{O}(\log^3 N \cdot (\log\log N)^2)$. In this work, we adapt the same security setting but with a more efficient constructions.

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. [13] in the context of protecting hidden volumes, a randomized procedure was used to achieve obliviousness. Later, Roche et al. [50] 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 [27] 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 as is our goal here.

ORAMs have also been used to expand searchable encryption with locality. Work by Demetrizis et al. [23] proposed a hierarchical square-root ORAM [32] to support searchable encryption. This scheme makes use of locality-preserving version of Melbourne Shuffle [46] 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.

2 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

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

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

2.1 Range ORAM and Locality

In this work, we give an ORAM construction specifically suited for accessing sequential ranges of data. Thus we will use a slightly different security definition to capture the fact that our construction will 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:

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

it holds that

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

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

Informally, this means that a Range ORAM can leak the size of the ranges that are being accessed by each operation. This definition is equivalent to the one used by Asharov et al. [8]. It is interesting to note that in practice if one was to perform a query for a range of blocks using a traditional ORAM then this would also likely leak the size of the range in the timing channel, i.e. the client would pause after finishing the range and the server could note how many accesses had been done. They also show that any efficient Range ORAM which does not leak information about the ranges being queried must suffer poor locality [8, Theorem 8.4].

Our construction (and also that of prior works) actually achieves a slightly stronger form of security, namely we require only the weaker constraint:

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

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. Another way to view this is that $\mathbb{O}(\log \ell_i)$ bits are leaked per access, which is the *order of magnitude* of the range.

Locality. We define the locality of an algorithm as *the number of seeks* required on the storage medium during the execution of that algorithm. Specifically, 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 $\vec{z}$ we call the *locality* of the algorithm.

In practice, there may actually be multiple, separate physical storage devices on the server-side. See Section 7 for a discussion of how to extend our solution to take advantage of this setting.

2.2 Path ORAM

One of the most efficient ORAM constructions currently known is Path ORAM, presented in the seminal work of Stefanov et al. [53]. 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 whether or not the data was modified, as in any case the tag is reassigned 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 [31, 53]) 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 [53]. The Ring ORAM construction improved this further, demonstrating that $Z \geq 3$ is sufficient with a *deterministic* eviction strategy [48].

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 [50, 57]) to store the position map. In either solution, the total communication overhead for a single access with Path ORAM is $\mathbb{O}(\log^2 N)$.

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 $\mathbb{O}(r \cdot \log^2 N)$ seeks, where r is the number of blocks in the range. A locality-friendly ORAM, as we achieve here, should require a number of seeks *independent of the range size r*.

3 OVERVIEW OF rORAM CONSTRUCTION

In this section, we describe the basic, core construction details for rORAM. First, we build a construction based on multiple Path ORAM trees and show how to support efficient range queries with a local position map. Following, we make two optimizations: we describe how to increase data locality via smart layouts of the data blocks, based on bit-reversed ordering, and second, we show how to use pointer-based methods to make recursive storage of the position map possible while maintaining locality.

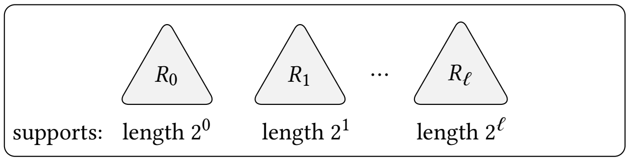


Figure 1: rORAM Organization. rORAM storing N blocks and supporting ranges up to 2^ℓ consists of $\ell + 1$ tree-based ORAMs $R_0, \ldots, R_\ell$. Each component ORAM R_i contains N blocks and supports ranges of size 2^i. All ORAMs have the same block size.

3.1 Core Construction

Multiple ORAMs each covering a subset of ranges. The first key construction detail, similar to that in prior work [8], is the use of multiple ORAMs to store ranges of a specific length.

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, the data is divided into ranges of size 2^i, as in $r^i_1, \ldots, r^i_{N/2^i}$. Each ORAM is specifically tailored so that contiguous ranges of length 2^i have a high degree of locality. The tradeoff is that these ranges can only be queried in their entirety, which is why we make use of $\ell + 1$ separate ORAMs: to support any size range with low overhead.

If the client requests a range that is exactly r^i_j, this could be fulfilled with a single access on the R_ith ORAM 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 $r^i_j \in r^{i'}_{j'}$, 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 $[a \cdot 2^i, (a + 1) \cdot 2^i)$ with $i < 6$ can cover $[31, 33)$.

Fortunately, there exists a simpler solution as noted in [8]. If a range overlaps a boundary, we can fulfill the request with *two* accesses of the same power-of-two size, which is proportional to the range size itself. 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 a piece of information about the range.

Operations for R_i. Each R_i is a tree-based, Path ORAM [48, 53, 59] storing all N logical blocks, duplicated across all R_i ORAMs. This allows rORAM to support range queries of any size efficiently. Crucially, the ORAMs have the *same physical block size* regardless of the served range size. This means that a single access in a given ORAM R_i occurs on 2^i blocks and is completed as a single *batched* operation. This is a significant difference compared to prior work [8] where each ORAM in the hierarchy used different physical block sizes to serve size ranges; a range is effectively just a larger sized block with a number of small blocks inside it. We will see that storing them with the same block size is the key to making our construction more efficient.

In rORAM, Each ORAM R_i supports the following operations:

- 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 $a = b \cdot 2^i$.
- BatchEvict(k, stash): Perform k evictions as a batch to write back multiple blocks to the ORAM from the stash for each of k evicted paths. Evictions occur in a deterministic order, and a global counter is used to maintain this order.

Remarks about BatchEvict. When a range is accessed from an ORAM R_j, in order to make sure that changes to these blocks are applied in all trees (in case a smaller or larger overlapping range is queried later), the new values must also be written to every other R_i for $i \neq j$. Therefore we cannot assume that the batch size k is equal to 2^i for ORAM R_i, because BatchEvict will occur across *all* ORAMs in the same magnitude as the queried range size. This means that a ReadRange operation on ORAM R_j will always be followed a BatchEvict(2^j, stash) to occur for all $\ell + 1$ ORAMs.

With different block sizes in the prior work [8], it is difficult to perform eviction of a small range in a larger-block ORAM with blocks designed for a large range. Prior work addresses this issue by amortizing the cost using a hierarchical ORAM [33]. In contrast, eviction in our scheme is much simpler and more efficient because the block sizes are the same in every constituent ORAM tree.

Operations on rORAM. The operations for the rORAM is composed of operations on each of the R_i path ORAMs. For rORAM, we have the following operation:

- Access(id, r): Given a range of size r beginning at logical identifier id, with $2^{i-1} < r \leq 2^i$, perform R_i.ReadRange(a_1) and R_i.ReadRange(a_2) with $a_1 = \lfloor \mathsf{id}/2^i \rfloor$ and $a_2 = (a_1 + 2^i) \bmod N$. The resulting $\mathbb{O}(r \log N)$ buckets in all 2^{i+1} paths are scanned linearly to access each of the ORAM tree levels. The updated data blocks are then appended to the stash of all $\ell + 1$ ORAMs. Then, for each R_j, call R_j.BatchEvict(2^{i+1}, 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.

3.2 Optimization 1: Smart 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 [31]. 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.

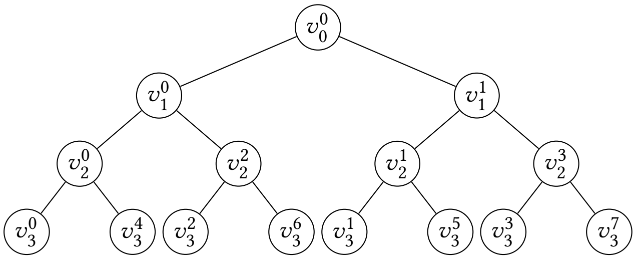


Figure 2: Labeling of ORAM tree buckets. A bucket label v_i^j signifies the jth bucket among those at level i in bit-reversed order.

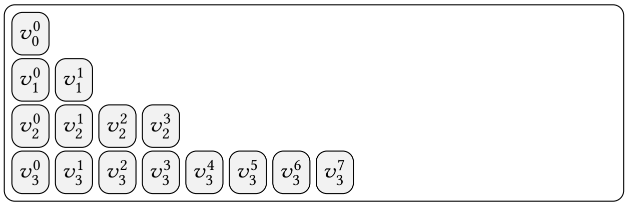

v_0^0
v_1^0 v_1^1
v_2^0 v_2^1 v_2^2 v_2^3
v_3^0 v_3^1 v_3^2 v_3^3 v_3^4 v_3^5 v_3^6 v_3^7

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

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 in the physical layout of the tree to achieve data locality for ranges.

Bit-reversed physical layout of ORAM R_i. An important observation is that the the path eviction schedule also implies deterministic ordering of *the evicted 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 at level 1, and 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 a high degree of locality. 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, this is *the first construction that considers the physical layout to improve efficiency of ORAM performance.*

Size independent $\mathbb{O}(\log N)$ seeks for BatchEvict. With a sequential layout of buckets on disk that match the bit-reversed order at each level x in the ORAM tree, BatchEvict(k) will visit $\min(k, 2^x)$ buckets at level x sequentially. 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 BatchEvict(k) is therefore

$\mathbb{O}(\log N)$. We stress that the number of seeks is independent of k, the number of eviction operations performed as a batch.

Optimizing seeks for ReadRange. The ReadRange on ORAM R_i reads *exactly* 2^i *blocks*. Naively storing all the blocks along random paths does not provide locality (as in the existing tree-based ORAMs), and will result in $\mathbb{O}(2^i \cdot \log N)$ seeks to read 2^i paths.

The number of seeks can be optimized here, similar to that of the batch eviction, by taking advantage of the physical disk layout with levels stored in bit-reversed order. In particular, we will have ORAM R_i *store logical blocks in the same range in paths with consecutive bit-reverse ordered leaves.* That is, according to the aforementioned labeling of buckets, addresses $[a, a + 2^i)$ are mapped to a 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 $j = 1, \ldots, 2^i - 1$, address $a + j$ is mapped to a path $P(v_h^{r+j \bmod N})$.

This mapping ensures that ReadRange also requires $\mathbb{O}(\log N)$ disk seeks, similar to the case of BatchEvict, because sequential paths are accessed in the bit-reversed order. At first glance it might seem bad for security that an entire range is stored sequentially. However since the size of queried ranges must be leaked anyway, and the start point of each range is chosen randomly and changed after each read, this does not actually reveal any further information to an adversary.

3.3 Optimization 2: Combining Position maps and Pointer-based Techniques

While the above optimization provides $\mathbb{O}(\log N)$ seeks for ReadRange and $\mathbb{O}(\log^2 N)$ for BatchEvict (one BatchEvict in each of $\ell \leq \log_2 N$ ORAMs), the system would still require $\mathbb{O}(\log^3 N)$ seeks if the position map could not be stored locally. However, we can use a pointer based method, similar to that of the pointer based techniques in Oblivious Data Structures [17, 49, 50, 61], to reduce the cost of the extra position maps queries and maintain $\mathbb{O}(\log^2 N)$ overall seek performance. In this section, we offer this optimization by first describing the challenges of maintaining the position map.

Challenges of naïve position map construction. Each R_i of the Path ORAMs is uniquely addressed for the blocks they store, even though each ORAM stores the same number of data blocks. As a result, there must be a separate position map for each of the ORAMs. If stored locally, this extra data is at no added cost, but typically the size of the position map may exceed local storage requirements and thus would also be stored recursively within separate ORAMs in the remote storage. Typically implementations require a recursive ORAM or an oblivious trie [50, 57] to store the position map obliviously, but both implementations require$\mathbb{O}(\log^2 N)$ seeks and network bandwidth[1].

Thus a ReadRange on a single ORAM costs $\mathbb{O}(\log N)$ seeks to access and evict, but a query on the position map costs $\mathbb{O}(\log^2 N)$ seeks. While this is an improvement over prior constructions, one must also consider the other $\mathbb{O}(\log N)$ ORAMs as data is duplicated. An access of data blocks on one ORAM requires updating *all* those data blocks on *all* $\mathbb{O}(\log N)$ other ORAMs via BatchEvict which each require position map accesses. The resulting number of seeks is $\mathbb{O}(\log^3 N)$, increasing the overall seeks for the rORAM construction. The goal is to reduce the number of seeks 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.

data	a	p_0	p_1	$\cdots$	p_ℓ

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

Reusing physical paths in unread Path ORAMs. A key observation is that since only one of the R_i Path ORAMs is actually read, the path locations of those blocks in the other ORAMs still remain oblivious, thus do not need to change. Recall that while data is duplicated across the ORAMs, the tags for which path data is assigned to is independent. Thus, only the ORAM for which the ReadRange occurred needs to make a position map update; the tags of the blocks in the other ORAMs can be reused, as proposed by Wan et al. [60] in a 2-server ORAM model for PIR.

Specifically, let a be the logical address for data block d_a which is currently found at path position p_i in ORAM R_i. When a ReadRange occurs that includes the logical address a in R_i, only tag p_i needs to be updated and randomly reassigned for a. The path tags in the other ORAMs, e.g., p_j in R_j for all $j \neq i$, can stay the same and be reused because *the physical path for the a was not revealed.*

Reusing the tags in the R_j other ORAMs is crucial because after a ReadRange the data blocks may be updated, and these updates must propagate to all the other ORAMs through a commensurate number of BatchEvict's. Now, during a batch eviction that includes a in ORAM R_j, the client can perform a position map look up to retrieve p_j and then replace the data for a with d'_a. Importantly, this is the same position for a as before in R_j, and the eviction procedure can remove duplicates by identifying fresher data higher in the tree (as described later). Unfortunately, this is not enough to reduce the position map lookups because while tags are only being updated in one of the range ORAMs, we require a position map look up in all the other ORAMs to learn the position of the data there, i.e., learning p_j for all $R_j, j \neq i$. As a result, the number of seeks and communication remain $\mathbb{O}(\log^3 N)$. But, we can further improve upon this by leveraging a pointer based technique.

Pointer-based techniques [17, 49, 50, 61]. Wang et al. [61] introduced pointer-based techniques to ORAM when they achieve an oblivious data structure (ODS), specifically an AVL tree on top of the non-recursive Path ORAM [57]. Briefly speaking, the pointer to a child node in the data structure *directly points to a physical location* instead of a logical location, and therefore it is no longer necessary to do position map lookups for every block. In essence, if you do a lookup for the root node you get all the remaining positions "for free" because they are contained within the links of the data structure itself.

[1] 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.

We extend this pointer-based technique to be applied over *multiple ORAMs* so that the above observation may come to fruition. In particular, alongside each physical block is stored *the physical locations 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 physical location 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

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

be the two physical blocks retrieved in this stage. Here p_j (resp., q_j) denotes the physical location 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 physical location of a should be p'_1.
(5) For $i = 0, \dots, \ell$:
Push the following two blocks in the stash for R_i:
$(d'_a, a, p_0, p'_1, p_2 \dots, p_\ell),\ (d'_b, b, q_0, q'_1, q_2 \dots, q_\ell)$.
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 physical location p_1, which needs $\mathbb{O}(\log^2 N)$ seeks. The physical locations 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. Generalizing this process, the rORAM only requires $\mathbb{O}(\log^2 N)$ seeks in total.

Handling duplicates. One thing to note here, as mentioned briefly earlier, 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 physical location 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.

4 FORMAL DESCRIPTION OF OUR SYSTEM

We now formally define the rORAM operations.

Position map and stash. The rORAM requires two supporting data structures, the position map and the stash, similar to Path ORAM. The position map for the range ORAM is 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), we map a *range ID* to a leaf label.
- The leaf label we store for a range corresponds to the leaf to which *the first block in the range* is mapped. This is enough since once we know the leaf label for the first block, we can easily determine the leaf labels for the remaining blocks due to our labeling mechanism.

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, our construction 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.

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 $r \leq L$ starting at any given addres $a \in [0, N - r)$. As explained in Section 3, 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 a two ReadRanges on the first/second half of the range, retrieve relevant 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 10 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).

5 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:

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

```
1:  Let $i \in [0, \ell)$ such that $2^{i-1} < r \leq 2^i$
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:     $\mathsf{stash}_j \leftarrow \mathsf{stash}_j \setminus \{B \in \mathsf{stash}_j : a_0 \leq B.a < a_0 + 2^{i+1}\}$
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 $D$
```

- 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.

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. In particular,

- 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: $\mathbb{O}(\log N)$.

Therefore, each 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 $L_i = 2^i \leq N$. 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 [8], 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 5.1. *Suppose ORAM R_i has bucket size $Z \geq 3$. Let $\mathrm{st}(R_i)$ be the number of blocks in the stash after a sequence of operations in ORAM R_i. Then, as long as $L_i \leq N/4$, we have*

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

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

Proof intuition. When looking into 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. So, by taking the union bound over these 2^i subtrees, we can prove the above theorem. The complete proof is found in Appendix A.

6 EMPIRICAL MEASUREMENT

6.1 Implementation Details

The rORAM is implemented on top of a publicly available Java library [4, 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[7] (currently redacted for submission).

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 a multiple 1GB files.

Platform. All benchmarks were evaluated on Linux installation with Intel Core i7-3520M processors running at 2.90GHz and 8GB+ of DDR3 DRAM. The storage 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[2] (cold storage HDD) mounted as an iSCSI device [6]. 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* [5].

6.2 Measurement Techniques

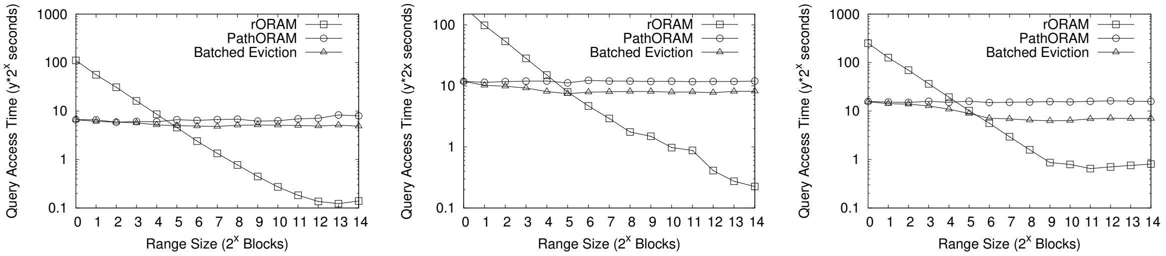
Benchmarks. To the best of our knowledge, this is the *first implementation and evaluation of a range ORAM construction.* Previous theoretical constructions [8, 24] are non-trivial to implement and have higher asymptotic costs, and so we use Path ORAM as a comparison point. While Path ORAM is not optimized for seek performance, it does provide a good baseline and the addition of our optimizations with a relaxed security definition does indeed result in significantly more efficient constructions for native Path ORAM without ranges.

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 [28]. The rORAM setup supports a maximum range size of $L = 2^{14}$ and the stashes are set for 128-bit security. 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 our 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.

[2] The choice of an EBS over Amazon S3 [2] is intentional – EBS provides a block storage device (the use-case for our construction) while S3 is an object store.

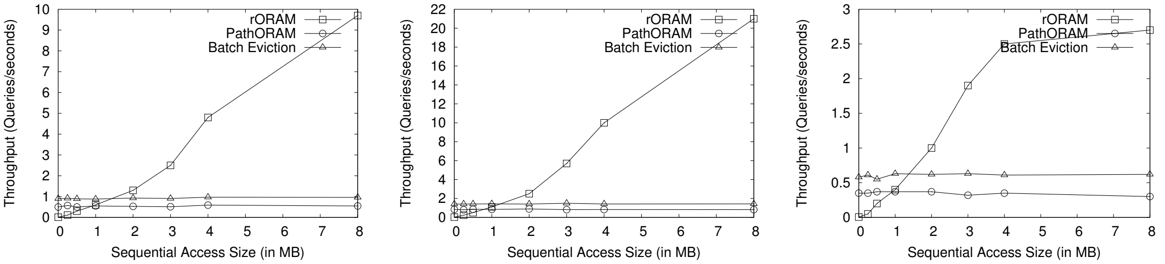


(a) Query access time on local SSD

(b) Query access time on local HDD

(c) Query access time on Network disk

Figure 5: Query access time (lower is better). Database size = 2^{22} 4kB blocks (16GB). For the local HDD, rORAM is 30-50x faster than Path ORAM for ranges sizes ≥ 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)

Figure 6: Sequential access throughput (higher is better). Database size = 2^{22} 4kB blocks (16GB). rORAM is 10-15x faster than Path ORAM for the local HDD and SSD. rORAM is almost 6x faster for network block devices.

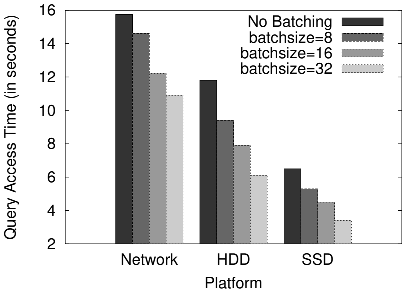


Figure 7: Average query access time (lower is better) for Path ORAM with batched evictions. For the local HDD, the combination of the seek-optimized physical layout and reduced number of I/Os due to batching results in a 2x improvement in average query response times. Similar improvements for SSDs and network devices is a result of reduce I/O requests (round-trips) due to batching.

Asynchronous versions of Path ORAM [51] can easily replace the synchronous versions used in our implementation.

6.3 Results

Batched evictions. The batch eviction procedure can improve the performance of Path ORAM, even without the addition of seek-optimized range functionality. To measure the effect of batch eviction, we evaluate the average query access time for Path ORAM with a deterministic eviction schedule (based on the bit-reverse ordering of leaves) performed in batches. The results are presented in Figure 7. As expected, batched evictions improve query access times by reducing the total number of seeks performed. Interestingly, batching evictions is helpful even for local SSDs and network devices since it reduces the number of I/O requests (round-trips).

Range queries. As a measurement of the range functionality, we measure the query access time query ranges of varying sizes for regular Path ORAM, Path ORAM with batched evictions (batch size = 32) and rORAM (Figure 5). 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 5 (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 due to lower asymptotic bandwidth costs. 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 ≥ 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 [50], 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 than Path ORAM for network block devices.

Sequential query throughput. 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. To measure this effect, we assess query throughput for largely sequential workloads. Specifically, similar to [10], we replay traces of the sequential read/write workloads used by FileBench version 1.4.9.1 to measure sequential access throughput. FileBench evaluates throughput by performing sequential accesses of user-specified sizes at random offsets followed by a small number of random accesses. To generate the trace, we first log all requests generated by FileBench while benchmarking a loopback device implemented in BUSE [19] and replay these requests to both Path ORAM and rORAM. Throughput results are presented in Figure 6 (a).

For the local HDD and SSD, the overall query throughput of rORAM increases with increases in sequential access size. For the local HDD (Figure 6 (a)) and the local SSD (Figure 6 (b)), rORAM can support up to 10 and 21 queries per second respectively, when performing sequential access of 8MB. 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.

7 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.

7.1 Parallel Seeks with Multiple Heads or Disks

Previously, the analytic assumption was that there existed a single read/write "head" on the server's storage device, but modern storage systems are not designed in this manner and 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 can be reduced to

$$\mathbb{O}\left(\log N \cdot \left(1 + \frac{\log N}{k}\right)\right).$$

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. construction itself.

7.2 Reducing position map costs

As described above, parallel seeks can improve the performance of the position map, but there are other optimizations for decrease 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 cat e $O(1)$ with parrallel seeks across levels, as described above.

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 which 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 were 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 map in the larger range trees 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 reduces 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 range trees, reasonably requiring that all accesses are on ranges is at least 8 blocks, reduces the global position map size to less than 1MB.

7.3 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 an 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 allows for read ranges, but only single block writes. 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 [15, 20, 30].

$\mathbb{O}(\log N)$ 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 the 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.

7.4 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*, as previously seen in multiple ORAM protocols [12, 53].

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.

8 CONCLUSION

In this paper, we introduce a new range ORAM construction that is locality optimized, only requiring $\mathbb{O}(\log^2 N)$ seeks, an $\mathbb{O}(\log N)$ improvement over prior work [8]. This is accomplished by making use of bit-reversed lexicographic ordering for the physical layout, batch eviction, and combining a pointer based technique with the position map. The implementation of the range ORAM is 30x-50x faster than using performing range queries on Path ORAM alone and a batch eviction Path ORAM. We also introduced a number of other tweaks and optimizations, including relaxing the security definition to reveal operation type and providing obliviousness in the malicious setting. Through this work, it should be clear that locality, the number of seeks, is an important factor in ORAM design. Even for ORAMs that do not naturally support range queries, we have shown that locality can have a large impact on performance and that seek optimization should be a design criteria for future ORAM technology.

9 ACKNOWLEDGEMENT

This work is supported by the National Science Foundation under awards 1526707, 1526102, 1319994, 1406177, 1618269 and by the Office of Naval Research.

REFERENCES

[1] April, 23 2018. Amazon Elastic Block Storage. (April, 23 2018). https://aws.amazon.com/ebs/.

[2] April, 23 2018. Amazon Simple Storage Service: GET Object. (April, 23 2018). https://docs.aws.amazon.com/AmazonS3/latest/API/RESTObjectGET.html.

[3] April, 23 2018. Ext4 Disk Layout. (April, 23 2018). https://ext4.wiki.kernel.org/index.php/Ext4_Disk_Layout.

[4] April, 23 2018. Home of the CURIOUS framework. (April, 23 2018). http://seclab.soic.indiana.edu/curious/.

[5] April, 23 2018. iPerf. (April, 23 2018). https://iperf.fr/.

[6] April, 23 2018. Linux-iSCSI Project. (April, 23 2018). http://linux-iscsi.sourceforge.net/.

[7] May 8, 2018. rORAM Implementation on Github. (May 8, 2018). https://github.com/anrinch/rORAM.

[8] Gilad Asharov, T-H. Hubert Chan, Kartik Nayak, Rafael Pass, Ling Ren, and Elaine Shi. 2017. Oblivious Computation with Data Locality. Cryptology ePrint Archive, Report 2017/772. (2017). http://eprint.iacr.org/2017/772.

[9] Gilad Asharov, Moni Naor, Gil Segev, and Ido Shahaf. 2016. Searchable symmetric encryption: optimal locality in linear space via two-dimensional balanced allocations. In *48th ACM STOC*, Daniel Wichs and Yishay Mansour (Eds.). ACM Press, Cambridge, MA, USA, 1101–1114.

[10] Vincent Bindschaedler, Muhammad Naveed, Xiaorui Pan, XiaoFeng Wang, and Yan Huang. 2015. Practicing Oblivious Access on Cloud Storage: the Gap, the Fallacy, and the New Way Forward. In *ACM CCS 15*, Indrajit Ray, Ninghui Li, and Christopher Kruegel: (Eds.). ACM Press, Denver, CO, USA, 837–849.

[11] Marina Blanton, Aaron Steele, and Mehrdad Aliasgari. 2013. Data-oblivious graph algorithms for secure computation and outsourcing. In *ASIACCS 13*, Kefei Chen, Qi Xie, Weidong Qiu, Ninghui Li, and Wen-Guey Tzeng (Eds.). ACM Press, Hangzhou, China, 207–218.

[12] Erik-Oliver Blass, Travis Mayberry, and Guevara Noubir. 2017. Multi-client Oblivious RAM Secure Against Malicious Servers. In *ACNS 17 (LNCS)*, Dieter Gollmann, Atsuko Miyaji, and Hiroaki Kikuchi (Eds.), Vol. 10355. Springer, Heidelberg, Germany, Kanazawa, Japan, 686–707.

[13] Erik-Oliver Blass, Travis Mayberry, Guevara Noubir, and Kaan Onarlioglu. 2014. Toward Robust Hidden Volumes Using Write-Only Oblivious RAM. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS '14)*. ACM, New York, NY, USA, 203–214. https://doi.org/10.1145/2660267.2660313

[14] Dan Boneh, David Mazieres, and Raluca Ada Popa. 2011. Remote oblivious storage: Making oblivious RAM practical. (2011). http://dspace.mit.edu/bitstream/handle/1721.1/62006/MIT-CSAIL-TR-2011-018.pdf.

[15] Christoph Bösch, Pieter Hartel, Willem Jonker, and Andreas Peter. 2014. A Survey of Provably Secure Searchable Encryption. *ACM Comput. Surv.* 47, 2 (Aug. 2014), 18:1–18:51. https://doi.org/10.1145/2636328

[16] David Cash and Stefano Tessaro. 2014. The Locality of Searchable Symmetric Encryption. In *EUROCRYPT 2014 (LNCS)*, Phong Q. Nguyen and Elisabeth Oswald (Eds.), Vol. 8441. Springer, Heidelberg, Germany, Copenhagen, Denmark, 351–368. https://doi.org/10.1007/978-3-642-55220-5_20

[17] Anrin Chakraborti, Chen Chen, and Radu Sion. 2017. DataLair: Efficient Block Storage with Plausible Deniability against Multi-Snapshot Adversaries. *Proceedings on Privacy Enhancing Technologies* 2017 (July 2017), 175–193. Issue 3.

[18] Kai-Min Chung, Zhenming Liu, and Rafael Pass. 2014. Statistically-secure ORAM with $\tilde{O}(\log^2 n)$ Overhead. In *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 62–81.

[19] Adam Cozzette. May 8, 2018. Block Device in User Space (BUSE). (May 8, 2018). "https://github.com/acozzette".

[20] Reza Curtmola, Juan A. Garay, Seny Kamara, and Rafail Ostrovsky. 2006. Searchable symmetric encryption: improved definitions and efficient constructions. In *ACM CCS 06*, Ari Juels, Rebecca N. Wright, and Sabrina De Capitani di Vimercati (Eds.). ACM Press, Alexandria, Virginia, USA, 79–88.

[21] Jeff Dean. 2018. Latency Numbers Every Programmer Should Know. Online. (2018). https://gist.github.com/jboner/2841832.

[22] Erik D. Demaine. 2002. Cache-Oblivious Algorithms and Data Structures. In *Lecture Notes from the EEF Summer School on Massive Data Sets*. http://erikdemaine.org/papers/BRICS2002/.

[23] Ioannis Demertzis, Dimitrios Papadopoulos, and Charalampos Papamanthou. 2017. Searchable Encryption with Optimal Locality: Achieving Sublogarithmic Read Efficiency. Cryptology ePrint Archive, Report 2017/749. (2017). http://eprint.iacr.org/2017/749.

[24] Ioannis Demertzis, Dimitrios Papadopoulos, and Charalampos Papamanthou. 2017. Searchable Encryption with Optimal Locality: Achieving Sublogarithmic Read Efficiency. Cryptology ePrint Archive, Report 2017/749. (2017). https://eprint.iacr.org/2017/749.

[25] Ioannis Demertzis and Charalampos Papamanthou. 2017. Fast Searchable Encryption With Tunable Locality. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD '17)*. ACM, New York, NY, USA, 1053–1067. https://doi.org/10.1145/3035918.3064057

[26] Jack Doerner and Abhi Shelat. 2017. Scaling ORAM for Secure Computation. In *ACM CCS 17*, Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu (Eds.). ACM Press, Dallas, TX, USA, 523–535.

[27] Christopher W. Fletcher, Ling Ren, Albert Kwon, Marten van Dijk, and Srinivas Devadas. 2015. Freecursive ORAM: [Nearly] Free Recursion and Integrity Verification for Position-based Oblivious RAM. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '15, Istanbul, Turkey, March 14-18, 2015.* 103–116.

[28] Christopher W. Fletcher, Ling Ren, Albert Kwon, Marten van Dijk, Emil Stefanov, Dimitrios Serpanos, and Srinivas Devadas. 2014. A Low-Latency, Low-Area Hardware Oblivious RAM Controller. Cryptology ePrint Archive, Report 2014/431. (2014). https://eprint.iacr.org/2014/431.

[29] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. 1999. Cache-Oblivious Algorithms. In *40th FOCS*. IEEE Computer Society Press, New York, NY, USA, 285–298.

[30] Benjamin Fuller, Mayank Varia, Arkady Yerukhimovich, Emily Shen, Ariel Hamlin, Vijay Gadepally, Richard Shay, John Darby Mitchell, and Robert K. Cunningham. 2017. SoK: Cryptographically Protected Database Search. In *2017 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, San Jose, CA, USA, 172–191.

[31] Craig Gentry, Kenny A. Goldman, Shai Halevi, Charanjit S. Jutla, Mariana Raykova, and Daniel Wichs. 2013. Optimizing ORAM and Using It Efficiently for Secure Computation. In *Privacy Enhancing Technologies - 13th International Symposium, PETS 2013, Bloomington, IN, USA, July 10-12, 2013. Proceedings.* 1–18.

[32] Oded Goldreich. 1987. Towards a Theory of Software Protection and Simulation by Oblivious RAMs. In *19th ACM STOC*, Alfred Aho (Ed.). ACM Press, New York City, NY, USA, 182–194.

[33] Oded Goldreich and Rafail Ostrovsky. 1996. Software Protection and Simulation on Oblivious RAMs. *J. ACM* 43, 3 (1996), 431–473. https://doi.org/10.1145/233551.233553

[34] Michael T. Goodrich, Michael Mitzenmacher, Olga Ohrimenko, and Roberto Tamassia. 2011. Oblivious RAM Simulation with Efficient Worst-case Access Overhead. In *Proceedings of the 3rd ACM Workshop on Cloud Computing Security Workshop (CCSW '11)*. ACM, New York, NY, USA, 95–100. https://doi.org/10.1145/2046660.2046680

[35] Michael T. Goodrich, Michael Mitzenmacher, Olga Ohrimenko, and Roberto Tamassia. 2012. Practical Oblivious Storage. In *Proceedings of the Second ACM Conference on Data and Application Security and Privacy (CODASPY '12)*. ACM, New York, NY, USA, 13–24. https://doi.org/10.1145/2133601.2133604

[36] Michael T. Goodrich, Michael Mitzenmacher, Olga Ohrimenko, and Roberto Tamassia. 2012. Privacy-preserving group data access via stateless oblivious RAM simulation. In *23rd SODA*, Yuval Rabani (Ed.). ACM-SIAM, Kyoto, Japan, 157–167.

[37] S. Dov Gordon, Jonathan Katz, Vladimir Kolesnikov, Fernando Krell, Tal Malkin, Mariana Raykova, and Yevgeniy Vahlis. 2012. Secure two-party computation in sublinear (amortized) time. In *ACM CCS 12*, Ting Yu, George Danezis, and Virgil D. Gligor (Eds.). ACM Press, Raleigh, NC, USA, 513–524.

[38] Thang Hoang, Ceyhun D. Ozkaptan, Attila A. Yavuz, Jorge Guajardo, and Tam Nguyen. 2017. S3ORAM: A Computation-Efficient and Constant Client Bandwidth Blowup ORAM with Shamir Secret Sharing. Cryptology ePrint Archive, Report 2017/819. (2017). http://eprint.iacr.org/2017/819.

[39] Yaoqi Jia, Tarik Moataz, Shruti Tople, and Prateek Saxena. 2016. OblivP2P: An Oblivious Peer-to-Peer Content Sharing System. In *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016.* 945–962.

[40] Eyal Kushilevitz, Steve Lu, and Rafail Ostrovsky. 2012. On the (in)security of hash-based oblivious RAM and a new balancing scheme. In *23rd SODA*, Yuval Rabani (Ed.). ACM-SIAM, Kyoto, Japan, 143–156.

[41] Chang Liu, Austin Harris, Martin Maas, Michael W. Hicks, Mohit Tiwari, and Elaine Shi. 2015. GhostRider: A Hardware-Software System for Memory Trace Oblivious Computation. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '15, Istanbul, Turkey, March 14-18, 2015.* 87–101.

[42] Chang Liu, Yan Huang, Elaine Shi, Jonathan Katz, and Michael W. Hicks. 2014. Automating Efficient RAM-Model Secure Computation. In *2014 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, Berkeley, CA, USA, 623–638. https://doi.org/10.1109/SP.2014.46

[43] Chang Liu, Xiao Shaun Wang, Kartik Nayak, Yan Huang, and Elaine Shi. 2015. ObliVM: A Programming Framework for Secure Computation. In *2015 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, San Jose, CA, USA, 359–376. https://doi.org/10.1109/SP.2015.29

[44] Travis Mayberry, Erik-Oliver Blass, and Agnes Hui Chan. 2014. Efficient Private File Retrieval by Combining ORAM and PIR. In *NDSS 2014*. The Internet Society, San Diego, CA, USA.

[45] Kartik Nayak, Christopher Fletcher, Ling Ren, Nishanth Chandran, Satya Lokam, Elaine Shi, and Vipul Goyal. 2017. Hop: Hardware makes obfuscation practical. In *24th Annual Network and Distributed System Security Symposium, NDSS*.

[46] Olga Ohrimenko, Michael T. Goodrich, Roberto Tamassia, and Eli Upfal. 2014. The Melbourne Shuffle: Improving Oblivious Storage in the Cloud. In *Automata, Languages, and Programming*, Javier Esparza, Pierre Fraigniaud, Thore Husfeldt, and Elias Koutsoupias (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 556–567.

[47] Rafail Ostrovsky. 1990. Efficient Computation on Oblivious RAMs. In *22nd ACM STOC*. ACM Press, Baltimore, MD, USA, 514–523.

[48] Ling Ren, Christopher W. Fletcher, Albert Kwon, Emil Stefanov, Elaine Shi, Marten van Dijk, and Srinivas Devadas. 2015. Constants Count: Practical Improvements to Oblivious RAM. In *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015.* 415–430.

[49] Daniel S. Roche, Adam J. Aviv, and Seung Geol Choi. 2016. A Practical Oblivious Map Data Structure with Secure Deletion and History Independence. In *2016 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, San Jose, CA, USA, 178–197. https://doi.org/10.1109/SP.2016.19

[50] Daniel S. Roche, Adam J. Aviv, Seung Geol Choi, and Travis Mayberry. 2017. Deterministic, Stash-Free Write-Only ORAM. In *ACM CCS 17*, Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu (Eds.). ACM Press, Dallas, TX, USA, 507–521.

[51] C. Sahin, V. Zakhary, A. E. Abbadi, H. Lin, and S. Tessaro. 2016. TaoStore: Overcoming Asynchronicity in Oblivious Data Storage. In *2016 IEEE Symposium on Security and Privacy (SP)*. 198–217. https://doi.org/10.1109/SP.2016.20

[52] Elaine Shi, T.-H. Hubert Chan, Emil Stefanov, and Mingfei Li. 2011. Oblivious RAM with $O((\log N)^3)$ Worst-Case Cost. In *ASIACRYPT 2011 (LNCS)*, Dong Hoon Lee and Xiaoyun Wang (Eds.), Vol. 7073. Springer, Heidelberg, Germany, Seoul, South Korea, 197–214.

[53] Emil Stefanov, Marten Van Dijk, Elaine Shi, T.-H. Hubert Chan, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. 2018. Path ORAM: An Extremely Simple Oblivious RAM Protocol. *J. ACM* 65, 4, Article 18 (April 2018), 26 pages. https://doi.org/10.1145/3177872

[54] Emil Stefanov and Elaine Shi. 2013. ObliviStore: High Performance Oblivious Cloud Storage. In *2013 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, Berkeley, CA, USA, 253–267.

[55] Emil Stefanov and Elaine Shi. 2013. ObliviStore: High Performance Oblivious Distributed Cloud Data Store. In *NDSS 2013*. The Internet Society, San Diego, CA, USA.

[56] Emil Stefanov, Elaine Shi, and Dawn Xiaodong Song. 2012. Towards Practical Oblivious RAM. In *NDSS 2012*. The Internet Society, San Diego, CA, USA.

[57] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher W. Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. 2013. Path ORAM: an extremely simple oblivious RAM protocol. In *ACM CCS 13*, Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung (Eds.). ACM Press, Berlin, Germany, 299–310.

[58] Tomas Toft. 2011. Brief Announcement: Secure data structures based on multi-party computation. In *30th ACM PODC*, Cyril Gavoille and Pierre Fraigniaud (Eds.). ACM, San Jose, CA, USA, 291–292.

[59] Xiao Wang, T.-H. Hubert Chan, and Elaine Shi. 2015. Circuit ORAM: On Tightness of the Goldreich-Ostrovsky Lower Bound. In *ACM CCS 15*, Indrajit Ray, Ninghui Li, and Christopher Kruegel: (Eds.). ACM Press, Denver, CO, USA, 850–861.

[60] Xiao Wang, Dov Gordon, and Jonathan Katz. 2018. Simple and Efficient Two-Server ORAM. Cryptology ePrint Archive, Report 2018/005. (2018). https://eprint.iacr.org/2018/005.

[61] Xiao Shaun Wang, Kartik Nayak, Chang Liu, T.-H. Hubert Chan, Elaine Shi, Emil Stefanov, and Yan Huang. 2014. Oblivious Data Structures. In *ACM CCS 14*, Gail-Joon Ahn, Moti Yung, and Ninghui Li (Eds.). ACM Press, Scottsdale, AZ, USA, 215–226.

[62] Peter Williams and Radu Sion. 2012. Single round access privacy on outsourced storage. In *ACM CCS 12*, Ting Yu, George Danezis, and Virgil D. Gligor (Eds.). ACM Press, Raleigh, NC, USA, 293–304.

[63] Samee Zahur, Xiao Shaun Wang, Mariana Raykova, Adria Gascón, Jack Doerner, David Evans, and Jonathan Katz. 2016. Revisiting Square-Root ORAM: Efficient Random Access in Multi-party Computation. In *2016 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, San Jose, CA, USA, 218–234. https://doi.org/10.1109/SP.2016.21

A PROOF OF THEOREM 5.1

As in the previous work [48, 53], we use ∞-ORAM, where each node in the tree has infinite capacity. It receives the same input request sequence as R_i.

A *rooted subtree* is a subtree which contains the parent of every node in the subtree; in particular every non-empty rooted subtree contains the root. For any rooted subtree T, let $X(T)$ be a random variable denoting the number of non-stale blocks stored in the nodes in T in ∞-ORAM. Let $n(T)$ denote the number of nodes in T. Then, by letting $p(\gamma) = \Pr[\exists T : X(T) > Z \cdot n(T) + \gamma]$, we have the following consequence of Lemma 2 in [53]:

$$\Pr[\mathrm{st}(R_i) > L_i \cdot (\lambda + 1)] = p(L_i \cdot (\lambda + 1)).$$

Partition of a rooted subtree. Consider the ORAM tree R_i, which we partition into $S_{-1}, S_0, \ldots, S_{L_i-1}$ as follows:

- Let S_{-1} be a subtree containing all the nodes at levels $0, 1, \ldots, i-1$.
- For $j = 0, 1, \ldots, L_i - 1$, let S_j be a subtree rooted with v_i^j (i.e., a node at level i) that has all the descendants of v_i^j as well.

So the entire tree consists of the first i levels in S_{-1} and then the 2^i disjoint subtrees $S_0, \ldots, S_{L_i-1}$ going from level i down to the leaves.

Then, for any rooted subtree T, we have

$$X(T) = X(T \cap S_{-1}) + X(T \cap S_0) + X(T \cap S_1) + \cdots + X(T \cap S_{L_i-1}).$$

Let $E(T, \gamma)$ denote an event that $X(T) > n(T) \cdot Z + \gamma$ and $E_j(T, \gamma)$ denote an event that $X(T \cap S_j) > n(T \cap S_j) \cdot Z + \gamma$. Then, by the pigeon-hole principle, we have:

$$E(T, L_i(\lambda + 1)) \Rightarrow E_{-1}(T, L_i) \vee \left(\bigvee_{j=0}^{L_i-1} E_j(T, \lambda) \right).$$

Let $p_j(\gamma) = \Pr[\exists T : E_j(T, \gamma)]$. The union bound gives:

$$\begin{aligned} p(L_i \cdot (\lambda + 1)) &= \Pr[\exists T : E(T, L_i(\lambda + 1)] \\ &\leq \Pr[\exists T : E_{-1}(T, L_i)] + \sum_{j=0}^{L_i-1} \Pr[\exists T : E_j(T, \lambda)] \\ &= p_{-1}(L_i) + \sum_{j=0}^{L_i-1} p_j(\lambda). \end{aligned}$$

Bounding $p_{-1}(L_i)$. We first argue that $p_{-1}(L_i) = 0$. This is because our deterministic eviction schedule ensures that after every $L_i = 2^i$ evictions, all blocks are pushed down at or below level i in ∞-ORAM. Therefore, only at most L_i blocks must remain in S_{-1}.

Chernoff-like bound. We would like to bound $p_j(\lambda)$. Fix some rooted subtree T. We start our analysis by relying on the paritioning effect that is explained in Section 5. That is, each S_j will have at most one non-stale block for each range, which implies that $T \cap S_j$ will have at most one block for each range. Let Range_i be the set of all logical ranges for R_i; each range has length 2^i, and its starting logical label is a multiple of 2^i. Let $Y(T)$ denote the number of different ranges to which the blocks in T belong. Then, due to the partitioning effect, we have

$$X(T \cap S_j) = Y(T \cap S_j).$$

That is, every block in $T \cap S_j$ belongs to a unique logical range of R_i.

In our construction, a logical range $[a \cdot 2^i, (a+1) \cdot 2^i)$ is assigned physical labels $[r, r + 2^i)$, where r is chosen at random from $[0, N)$. As observed in [8, Claim 3.2], this implies that the event that a range will be in S_j is independent of the event that other ranges will be in S_j. Therefore, we can apply the Chernoff-like bound given in [48, Section 4.3] where we denote $Y_j = Y(T \cap S_j)$:

$$E\left[e^{t \cdot Y_j}\right] \leq e^{(e^t - 1)E[Y_j]}.$$

Denote $X_j = X(T \cap S_j)$. Since we have $X_j = Y_j$, we have:

$$E\left[e^{t \cdot X_j}\right] \leq e^{(e^t - 1)E[X_j]}.$$

Bounding $E[X_j]$. The same analysis as in [48, Lemma 3] with $A = 1$ applies, and we get

$$E[X_j] \leq n(T \cup S_j).$$

This is because dependency (among the blocks in the same range) becomes irrelevant, when we consider the expectation due to linearity of expectation. Nevertheless, we give the proof for completeness.

For each node v of the ORAM tree, we define a random variable χ_v to be the number of blocks in v after the last eviction. Note that we have

$$E[X_j] = \sum_{v \in T \cap S_j} E[\chi_v].$$

So, it is sufficient to show that $E[\chi_v] \leq 1$ for any node v.

For our analysis, given a node v and a logical block with address x, we define an indicator random variable $\chi_{x,v} \in \{0,1\}$ for the event that a logical block with address x is located in node v at the end of the last eviction. Let $q_{x,v} = \Pr[\chi_{x,v} = 1]$.

If v is a leaf node, a fresh record corresponding to logical address x can be stored in that bucket if x is mapped to v and there was some evict operation that puts x in v. Since a block x is mapped to a random leaf, the probability that a block x is mapped to v is $1/N$, which implies $q_{x,v} \leq 1/N$ and thereby $E[\chi_{x,v}] \leq 1/N$. Since there are at most N logical blocks, by taking the linearity of expectation, we have

$$E[\chi_v] \leq E\left[\sum_{x \in [0,N)} \chi_{x,v}\right] = \sum_{x \in [0,N)} E[\chi_{x,v}] \leq \sum_{x \in [0,N)} (1/N) \leq 1.$$

Suppose v is a non-leaf node at level ℓ. Consider the last two evictions paths p_1 and p_2 that touch v, where p_2 takes place later. Say that the second path p_2 goes through v and some child c of v. Then, our deterministic scheduling makes sure that p_1 goes through v and the other child c'. Note:

- Blocks coming before the time of p_1 will never be in v, since two eviction paths p_1 and p_2 will push all blocks down at or below level $\ell + 1$.
- Blocks coming after time of p_2 will never be in v, since p_2 is the last eviction that touches v.

This means that the only blocks that entered between p_1 and p_2 can possibly remain in v. Let Between be the set of such blocks. Our deterministic scheduling ensures that the time span between p_1 and p_2 is exactly 2^ℓ, implying that $|\text{Between}| \leq 2^\ell$ (one can make a similar argument if there is only one eviction path that touches v throughout the entire access sequence).

Moreover, if a block in Between remains in v after p_2, it must be the case that the block must have been mapped to a physical leaf label from the descendant of c'; otherwise, p_2 will push it down to at or below c. Since the number descendant leaves of c' is $N/2^{\ell+1}$, for a block $x \in$ Between, the probability that x is randomly assigned to one of such leaves is $1/2^{\ell+1}$. Therefore, we have

$$\begin{aligned} E[\chi_v] &\leq E\left[\sum_{x \in \text{Between}} \chi_{x,v}\right] = \sum_{x \in \text{Between}} E[\chi_{x,v}] \\ &= \sum_{x \in \text{Between}} 1/2^{\ell+1} \leq 1/2. \end{aligned}$$

Let $n = n(T \cap S_j)$ for simplicity. As long as $L_i \leq N/4$ which ensures that $|S_j| \geq 7$, we have at most $\frac{4}{7} \cdot n$ leaves. Therefore, we have:

$$\begin{aligned} E_{[}X_j] &\leq \sum_{v \in \text{leaves}} E[\chi_v] + \sum_{v \in \text{internal}} E[\chi_v] \\ &\leq \frac{4n}{7} \cdot 1 + \frac{3n}{7} \cdot \frac{1}{2} \\ &\leq \frac{11}{14} \cdot n \end{aligned}$$

Putting them all together. Now, we are ready to bound $p_j(\lambda)$. Consider a subtree T and recall $n = n(T \cap S_j)$. Then we have

$$\begin{aligned} \Pr[E_j(T,\lambda)] &= \Pr[X_j > Zn + \lambda] \\ &= \Pr[e^{tX_j} > e^{-t(Zn+\lambda)}] \\ &\leq E\left[e^{tX_j}\right] \cdot e^{-t(Zn+\lambda)} \\ &\leq e^{(e^t-1)\frac{11n}{14}} \cdot e^{-t(Zn+\lambda)} \\ &\leq e^{-t\lambda} \cdot e^{(e^t-1-tZ)n} \\ &= Z^{-\lambda} \cdot e^{-n(Z\ln Z - \frac{11}{14}(Z-1))} \end{aligned}$$

We took t such that $e^t = Z$ in the last line in the above.

Let $q = Z \ln Z - \frac{11}{14}(Z-1) - \ln 4$. We assume $Z \geq 3$. Note, then we have $\frac{1}{1-e^{-q}} < 3.5$. Finally, we have

$$\begin{aligned} p_j(\lambda) = \Pr[\exists T : E_j(T,\lambda)] &\leq \sum_{n \geq 1} 4^n \cdot \max_{T : n(T \cap S_j) = n} \Pr[E_j(T,\lambda)] \\ &\leq \sum_{n \geq 1} 4^n \cdot Z^{-\lambda} \cdot e^{-n(Z\ln Z - \frac{11}{14}(Z-1))} \\ &= \sum_{n \geq 1} Z^{-\lambda} \cdot e^{-nq} \\ &< \frac{Z^{-\lambda}}{1 - e^{-q}} \\ &< 3.5 \cdot Z^{-\lambda} \end{aligned}$$

In summary, we have:

$$p(L_i \cdot (\lambda + 1)) = p_{-1}(L_i) + \sum_{j=0}^{L_i - 1} p_j(\lambda) \leq 3.5 \cdot L_i \cdot Z^{-\lambda}.$$