Recursive ORAMs with Practical Constructions

Sarvar Patel[*] Giuseppe Persiano[†] Kevin Yeo[‡]

September 30, 2017

Abstract

We present Recursive Square Root ORAM (R-SQRT), a simple and flexible ORAM that can be instantiated for different client storage requirements. R-SQRT requires significantly less bandwidth than Ring and Partition ORAM, the previous two best practical constructions in their respective classes of ORAM according to client storage requirements. Specifically, R-SQRT is a 4x improvement in amortized bandwidth over Ring ORAM for similar server storage. R-SQRT is also a 1.33-1.5x improvement over Partition ORAM under the same memory restrictions. R-SQRT-AHE, a variant of R-SQRT, is a 1.67-1.75x improvement over the reported Partition ORAM results in the same settings. All the while, R-SQRT maintains a single data roundtrip per query. We emphasize the simplicity of R-SQRT which uses straightforward security and performance proofs.

Additionally, we present Twice-Recursive Square Root ORAM (TR-SQRT) with smaller client storage requirements. Due to its flexibility, we construct several instantiations under different memory requirements. TR-SQRT is asymptotically competitive with previous results, yet remarkably simple.

[*]Google, Inc., sarvar@google.com
[†]Google, Inc. and Università di Salerno, giuper@gmail.com
[‡]Google, Inc., kwlyeo@google.com

Contents

1 Introduction

The concept of an *Oblivious RAM* (or, simply, *ORAM*) was introduced by Goldreich [9] and consists of a protocol between a *client* and a *server*. The client has N data blocks each of size B remotely stored on the server. The client wants to be able to access the data blocks in an *oblivious* manner. That is, while hiding the access pattern. This can be easily achieved if every time the client wishes to access one block, all N blocks are downloaded (in a streaming fashion) from the server, thus incurring in a linear bandwidth overhead and requiring constant memory on the client side. The construction proposed by [10, 19], Square Root ORAM (SQRT ORAM), introduced the first construction with bandwidth $O(\sqrt{N} \log N)$ blocks per ORAM access. Since then, several other constructions have been presented trying to reduce the bandwidth overhead as it is the main measure of efficiency along with client storage.

In this paper, we present constructions that recursively apply the ideas of SQRT ORAM with a focus on simple and practical constructions. Our constructions should be flexible to handle different constraints in real-world scenarios.

The idea of recursively applying ORAM schemes was presented as soon as the first non-trivial ORAM construction was shown [10] and various other hierarchical constructions have been proposed [11, 13, 20] since (see Section 2.1 for a discussion). Roughly speaking, the server storage of a recursive ORAM is partitioned into l levels, for some integer l, and each level must be periodically shuffled into a higher level in an oblivious manner. The need for oblivious shuffling was the main impediment to an efficient recursive ORAM with low bandwidth overhead. This motivated the study of tree-based ORAMs, most notably the very elegant Path ORAM [27] that dispenses with the need for an oblivious shuffle by instead continuously reshuffling carefully chosen small portions of the server storage by entirely downloading to client memory. Tree-based constructions though tend to be more complex to analyse and require sophisticated probabilistic arguments to bound client memory usage (see Section 2.2).

In this work, we stay in the recursive field and present recursive ORAMs that are more efficient than previous constructions. Early constructions employed sorting networks for oblivious shuffling (with total bandwidth $\Omega(N \log N)$) and the first linear oblivious shuffling was given in [18]. We observe though that known oblivious shuffling algorithms (including the ones based on sorting networks) protect from adversaries that are much more powerful than the ones we have to deal with in ORAMs. Indeed, typically in a recursive ORAM, the adversarial server learns the current position of a subset of the blocks (the ones that were involved in client accesses performed since the previous reshuffle) and has no information on the remaining *untouched* blocks. As a consequence, the untouched blocks need not to be "completely" re-shuffled and this allows for a substantial saving in bandwidth, as shown in [21]. Further optimization can be made by observing that a fraction of the blocks in each level are dummies. Dummy blocks do not contain any data and are used only to hide which block the client really meant to download. Upload bandwidth optimization for shuffling with dummies are also shown in [21].

Tree-based and recursive constructions share the need of downloading several blocks for each client access. Indeed, recursive ORAMs download a block from each level of the server storage where at most one block is real. Similarly, tree-based constructions download an entire path. The XOR technique [6] reduces the online bandwidth in tree-based constructions. Unfortunately, this does not interact well with the optimization regarding dummy blocks. Instead, we employ an oblivious selection technique using additive homomorphic encryption to reduce the number of blocks transferred for each client access (see Section 7.3).

Our Contributions. Next, we outline our constructions and compare them to the current, state-of-the-art constructions in various classes. All our constructions are conceptually simple and their analysis does not require complex probabilistic arguments.

In Section 5, we present *Recursive SQRT ORAM* (R-SQRT) a construction that is parametrized by two integers c and l that, roughly speaking, determine the degree and the depth of the recursion, respectively. The client of R-SQRT stores a *position map* of size $O(N/B)$ blocks which keeps track of the location of the current version of each block. We give a detailed analysis of the hidden constants of R-SQRT in Section 7 showing that R-SQRT can be instantiated to use less bandwidth than previous state-of-the-art ORAMs with position maps. Specifically, we consider *online bandwidth* that is the number of blocks that are transferred for each block access and *amortized bandwidth* that takes into account the blocks transferred to periodically re-adjust the server storage (that is, to obliviously shuffle the blocks). In Section 6, we present a version of R-SQRT, denoted R-D-SQRT, where the worst case and amortized bandwidth are equal while maintaining small online bandwidth.

Our second construction, *Twice Recursive SQRT ORAM* (TR-SQRT), is shown in Section 8. TR-SQRT derives from recursive storing the position map of R-SQRT and thus it removes the requirement of a client-stored position map. By using different shuffling algorithms, we construct several variations of TR-SQRT with different client memory sizes. The asymptotic results of TR-SQRT are competitive with previous works. We also construct TR-D-SQRT that, using techniques similar to the ones in Section 6, reduces the worst case bandwidth requirement of each client access.

Comparisons with Previous Constructions. In evaluating ORAM constructions, we look at bandwidth, client memory and server memory. Throughout the rest of the paper, we refer to bandwidth as the number of blocks transmitted between the client and the server (in either direction). For example, a bandwidth cost of 5 means 5 blocks of data are being transmitted for each client block access. Bandwidth is the most expensive resource and in this work we try to minimize it by keeping the server storage to be at most $c \cdot N$, for some small constant c. To compare our result, we consider five classes of ORAMs according to different client memory requirements. Thanks to the simplicity and flexibility of our constructions, we present competitive ORAM constructions in each of the five classes.

First, we discuss two classes with focus on practical real-world constructions. The two classes require storage of a position map of $O(N/B)$ blocks and $\omega(\log N)$ or $O(\sqrt{N})$ additional blocks, respectively. A position map keeps track of the position of the current version of each block and, while asymptotically inefficient, they are small enough to be practical in most real-world situations and they have been employed in several practical ORAMs (most notably, Partition and Ring ORAM). For example, the position map of a 4 GB of memory consisting of $N = 2^{20}$ blocks of size $B = 4$ KB can be stored using only 256 blocks.

The first class of constructions requires $\omega(\log N)$ blocks and a position map on the client. The best construction in this class is Ring ORAM [22]. When instantiated to use $8N$ blocks of server memory, Ring ORAM requires $6 \log_2 N$ blocks of amortized bandwidth. Pushing server memory to an impractical $100N$ blocks reduces Ring ORAM's amortized bandwidth to $2.5 \log_2 N$. When instantiated with depth $l = \log_2 N - \log_2 \log_2 N - 1$ and $c = 2$, R-SQRT achieves $1.5(\log_2 N - \log_2 \log_2 N)$ blocks of amortized bandwidth using $4N$ blocks of server memory. So, R-SQRT is a 4x improvement for practical server memory sizes. In the table below we also mention R-D-SQRT whose worst case bandwidth is equal to the amortized bandwidth (whereas R-SQRT has a much higher worst case).

Table 1: ORAMs with $O(N/B) + \omega(\log N)$ Client Storage

	Online Bandwidth	Amortized Bandwidth	Server Storage
Path [27]	$4 \log_2 N$	$8 \log_2 N$	$8N$
Ring [22]	1	$6 \log_2 N$	$8N$
Ring [22]	1	$2.5 \log_2 N$	$> 100N$
R-SQRT	1	$1.5 \log_2 \frac{N}{\log_2 N}$	$4N$
R-D-SQRT	1	$1.5 \log_2 \frac{N}{\log_2 N}$	$4N$

Constructions in the second class require $O(\sqrt{N})$ blocks in addition to the position map in client memory. Partition ORAM [26] is the best construction in this class and uses a little over $\log_2 N$ blocks of amortized bandwidth with $4N$ blocks of server storage. When instantiated with depth $l = 0.5 \log_2 N - 1$ and $c = 2$, R-SQRT achieves $0.75 \log_2 N$ blocks of amortized bandwidth with an identical $4N$ blocks of server storage thus yielding a 1.33-1.5x improvement over Partition ORAM. By using dummy block optimization, we obtain a variant of R-SQRT, R-SQRT-AHE, which is a 1.67-1.75x improvement over reported Partition ORAM performance in the same settings.

Table 2: ORAMs with $O(N/B) + O(\sqrt{N})$ Client Storage

	Online Bandwidth	Amortized Bandwidth	Server Storage
Partition [26]	$\log_2 N$	$> \log_2 N$	$4N$
Path [27]	$2\log_2 N$	$4\log_2 N$	$4N$
Ring [22]	1	$3\log_2 N$	$4N$
Ring [22]	1	$1.25\log_2 N$	$> 50N$
R-SQRT	1	$0.75\log_2 N$	$4N$
R-D-SQRT	1	$0.75\log_2 N$	$4N$

The ORAMs in the remaining three classes do not store a position map in the client. They are more theoretical with a focus on asymptotics. The three classes are parametrized by available client storage of $O(N^\epsilon)$, $\omega(\log N)$ and $O(1)$ respectively. For each class, we construct a variant of TR-SQRT with different oblivious shuffling algorithms. We remark that for the case of $O(1)$ client memory, we use the AKS sorting sorting network as the shuffling algorithm which is only of interest for large values of N. Nonetheless, the asymptotic performance of TR-SQRT is competitive with previous best results in each class. We also remark that TR-SQRT is the first hierarchical ORAM in the $\omega(\log N)$ class with better performance from among those that only need $O(1)$ client storage. For comparison, we use TR-D-SQRT, the version of TR-SQRT that reduces the worst case bandwidth, and we refer to the construction from Theorems 9, 10 and 11.

Table 3: ORAMs with $O(N^\epsilon)$ Client Storage

	Amortized Bandwidth	Worst Case Bandwidth	Client Storage
GMOT [12]	$O(\log N)$	$O(\log N)$	$O(N^\epsilon)$
Partition [26]	$O\left(\frac{\log^2 N}{\log B}\right)$	$O(\sqrt{N})$	$O(\sqrt{N})$
TR-D-SQRT	$O\left(\frac{\log^2 N}{\log B}\right)$	$O\left(\frac{\log^2 N}{\log B}\right)$	$O(N^\epsilon)$

Table 4: ORAMs with $\omega(\log N)$ Client Storage

	Amortized Bandwidth	Worst Case Bandwidth
Path [27]	$O\left(\frac{\log^2 N}{\log B}\right)$	$O\left(\frac{\log^2 N}{\log B}\right)$
Ring [22]	$O\left(\frac{\log^2 N}{\log B}\right)$	$O\left(\frac{\log^2 N}{\log B}\right)$
TR-D-SQRT	$O\left(\frac{\log^3 N}{\log\log N \log B}\right)$	$O\left(\frac{\log^3 N}{\log\log N \log B}\right)$

Table 5: ORAMs with $O(1)$ Client Storage

	Amortized Bandwidth	Worst Case Bandwidth	Server Storage
Square Root [10]	$O(\sqrt{N}\log N)$	$O(N\log N)$	$O(N)$
Hierarchical [10]	$O(\log^3 N)$	$O(N\log N)$	$O(N\log N)$
OS [20]	$O(\log^3 N)$	$O(\log^3 N)$	$O(N\log N)$
GMOT [11]	$O(\log^2 N)$	$O(\log^2 N)$	$O(N)$
KLO [15]	$O(\frac{\log^2 N}{\log\log N})$	$O(\frac{\log^2 N}{\log\log N})$	$O(N)$
TR-D-SQRT	$O\left(\frac{\log^3 N}{\log B}\right)$	$O\left(\frac{\log^3 N}{\log B}\right)$	$O(N)$

2 Related Work

ORAMs have been previously applied to several scenarios. Wang *et al.* [30] used ORAMs to construct oblivious versions of common data structures such as maps, sets, stacks and queues. In addition, oblivious versions of common algorithms such as shortest-path are presented. Zahur *et al.* [31] investigated secure computation in the RAM model with the original SQRT ORAM of Goldreich and Ostrovsky. In this model, ORAM is a primary building block as it provides a memory abstraction that can obliviously read and write to arbitrary memory locations.

Previous ORAM protocols can be roughly divided into two classes: hierarchical and tree constructions. We outline the best results in both classes.

2.1 Hierarchical Constructions

One class of ORAM constructions are called *Hierarchical ORAMs.* In general, Hierarchical constructions are divided into several levels of memory, decreasing in size. After a query occurs, the retrieved data block is stored in the level with the smallest memory. As levels fill, data blocks are oblivious moved to larger memory levels.

The first Hierarchical construction was described in [10] by Goldreich and Ostrovsky, which required $O(\log^3 N)$ amortized query bandwidth. However, the proposal required the use of expensive sorting networks (either Batcher's Sort [2] or AKS [1]) and $O(N \log N)$ server storage preventing it from being practical. Further work done by Ostrovsky and Shoup [20] lowered the worst case to $O(\log^3 N)$.

More recently, work by Stefanov *et al.* [26] removed the use of expensive sorting networks by introducing a clever partition framework. Each partition consisted of ORAMs with $O(\sqrt{N})$ blocks of capacity. All the data blocks within a partition fit into a client memory. The oblivious sorting algorithms could be replaced with a simple download and upload of partitions. Their construction required $O(\log N)$ amortized query bandwidth and $O(N/B)$ client memory. A second recursive construction reduced client memory to $O(\sqrt{N})$ but $O(\log^2 N/ \log B)$ amortized query bandwidth. Oblivistore [25] is implemented using partioning.

The best asymptotic results for ORAM have been shown by Goodrich *et al.* [11]. Using Cuckoo Hashing, they present two constructions. The first construction required only $O(\log N)$ amortized query bandwidth while using only $O(N^\epsilon)$ client memory. Further work in [12] reduced the worst case to $O(\log N)$ for the first construction. In their other construction, they reduced the client memory to $O(1)$ but increased amortized query bandwidth to $O(\log^2 N)$. Work done in [13] allows multiple stateless users to access the second construction. Kushilevitz *et al.* [15], improved the bandwidth to $O(\log^2 N/ \log \log N)$ in the $O(1)$ client storage model. Unfortunately, these constructions have large hidden constants. While this class provides the best asymptotic results, the constants prevent their use on a practical number of data blocks.

2.2 Tree Constructions

Shi *et. al* introduced tree-based ORAM constructions in [24]. In the elegant work of Stefanov *et. al* in [27], the simple Path ORAM was proposed. Path ORAM has one of the best asymptotic and practical performances obtained amongst tree constructions. Path ORAM required $O(\log N)$ worst case query bandwidth using $O(N/B)$ client memory. By recursively applying their ideas, a construction with $O(\log^2 N/ \log B)$ worst case query bandwidth, but only $\omega(\log N)$ client memory is required.

Work done by Ren *et. al* introduced Ring ORAM [22], which improved the practicality of Path ORAM. Online query bandwidth was decreased to $O(1)$ using an XOR technique, first described in [6]. The overall amortized bandwidth was decreased by 2.5x - 4x.

Two recent constructions have appeared using *Garbled Circuits.* In general, circuits are built to handle the eviction process in the tree-based constructions. Wang *et al.* show that there exists a small circuit to perform eviction in Path ORAM [28]. Garg *et al.* show that garbled circuits can reduce the number of rounds in ORAM to exactly two [8]. SCORAM [29] was designed for small circuit complexity, for use in secure computation. Unfortunately, garbled circuits still remain expensive for practical use.

Another ORAM construction using *Homomorphic Encryption* was introduced by Devadas *et al.* [7]. Using either the Damgard-Jurik cryptosystem or BGV somewhat homomorphic encryption without bootstrapping [4], Onion ORAM only requires $O(1)$ query bandwidth in the worst case with polylogarithmic server computation. However, Onion ORAM requires block sizes of $\Omega(\log^5 N)$ identifiers. For example, when $N = 2^{20}$, blocks must be at least 8 MB.

3 Problem Definition

We formally define the *Oblivious RAM* problem. The storage model and its assumptions are explained in detail as well as the proper security definitions of obliviousness. We recall that RAM protocols store N data blocks and allow clients to request any single block in a query.

3.1 Memory Model

Throughout this work, we assume that both the client and server have their own separate memory. Any operations that are performed in the client's memory is private and completely hidden from the server. The server observes information data transmission between the client and server and all operations performed on server memory.

The data is divided into N data blocks, each exactly size of B words. In addition to storing the B words, each data block may store metadata. The metadata can include timestamps, indexes, positions, etc. When we refer to performing an operation on the data block, we also perform the operation on the metadata. For example, encrypting the data block means also encrypting the metadata.

3.2 Security

From a high level, ORAM protocols should guarantee two properties:

1. The server does not learn information about the contents of the N data blocks.

2. The server cannot distinguish any two access patterns to data blocks of the same length.

The first property is guaranteed by ensuring all data blocks are IND-CPA encrypted before being uploaded to the server. Furthermore, all data blocks are re-encrypted when downloaded and re-uploaded. With fresh randomness, the server is unable to link information between two different encryptions of the same block. Therefore, the goal of ORAM protocols focus on the second property.

4 Tools

In this section, we discuss the main building blocks to our constructions. We specify the shuffling algorithms which rearrange data blocks on the server in an oblivious manner. Next, we describe Square Root ORAM [10] in detail, which is the basis of our constructions.

4.1 Oblivious Shuffling

The *Oblivious Shuffle* problem was first introduced in [18]. Suppose that an untrusted adversarial server is currently storing N data blocks in the *source array* A. The N blocks are stored according to some permutation π, that is block i is stored at $A[\pi(i)]$. The goal of shuffling is to move all N data blocks in A to a destination array D according to some new permutation σ. Formally, after the shuffle algorithm terminates, block i should be located at $D[\sigma(i)]$. For a shuffling algorithm to be oblivious, the server should not learn about σ or data block contents.

Ohrimenko *et al.* [18] proposed the first $O(N)$ oblivious shuffling using only $O(N^\epsilon)$ client memory, named the *MelbourneShuffle*. Previously, oblivious shuffling had been performed using $\Omega(N \log N)$ oblivious sorting algorithms like Batcher's Sort or AKS.

Patel *et al.* [21] introduced the notion of a K*-Oblivious Shuffle*, a variant of the Oblivious Shuffle algorithm. In Oblivious Shuffling it is implicitly assumed that permutation π is entirely revealed to the adversarial server. This is rarely the case when Oblivious Shuffle is used as a component in a larger ORAM construction. Indeed, in our constructions everytime (a level of) the server memory is to be reshuffled, it is well known the number K of the values of π known to the server and the actual values (as they correspond to client accesses that have taken place between two consecutive reshuffles). In the same work, the first

K-Oblivious Shuffle algorithm, *CacheShuffle*, is presented. With $O(K)$ blocks of client storage, CacheShuffle requires exactly $2N$ blocks of bandwidth to shuffle. Note that nothing is gained asymptotically but, as we are interested in concrete construction, the small constant 2 allows us to construct a more efficient ORAM. Additional optimizations are shown for handling dummy blocks using polynomial interpolation.

4.2 Revisiting Square Root ORAM

We revisit the first ORAM by Goldreich and Ostrovsky [10], that we refer to as the *SQRT ORAM.* We present a variation, which we call GO. GO consists of a *shelter* Sh stored in client memory and of a server storage denoted by M. GO is parametrized by S, the number of data blocks stored in the shelter Sh. We note GO uses more client memory than the original SQRT ORAM but it is a more intuitive building block for the rest of our work.

For N data blocks, $\mathsf{B} = B_1, \ldots, B_N$, the server storage M is of size $N + S$; N of the data blocks are *real* and S are *dummy* data blocks. We use the notation $\mathrm{GO}(\mathsf{B}, S)$ to denote an instantiation on the N blocks of B using a shelter, Sh, with S blocks. We will assume that $\mathsf{M}[q]$ and $\mathsf{Sh}[q]$ refers to access of array location q of M and Sh respectively. In the original Square Root ORAM, the *shelter* is stored on the server and downloaded before each query. In GO, instead, Sh always stays on the client.

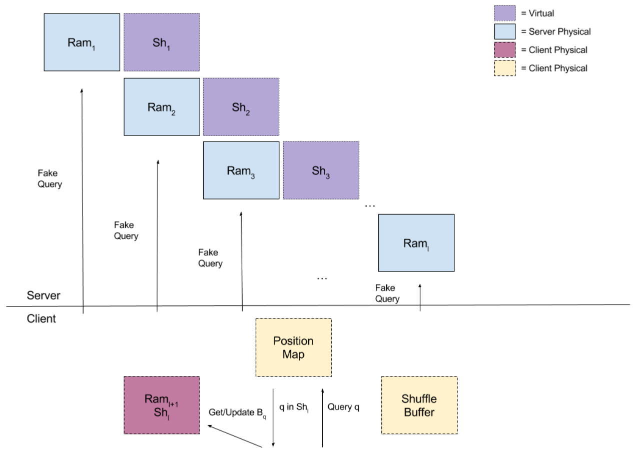
Additionally, the client stores a *position map*, PM. PM keeps essential information to help query data blocks. Specifically, for all block B_i, $\mathsf{PM}[i] = (\mathtt{lev}_i, \mathtt{pos}_i)$ stores a *level* and a *position.* The level $\mathtt{lev}_i \in \{\mathsf{Sh}, \mathsf{M}\}$, denotes whether B_i is currently in the shelter or main. The position, $\mathtt{pos}_i$, specifies the location of B_i within the structure. Note, the position map is not explicit in the original construction, but is useful for our future protocols.

The system is initialized by the client by randomly selecting a permutation π of $[N + S]$ and by storing data block i, for $i = 1, \ldots, N$, in encrypted form at location $\pi(i)$ in M on the server. We note that i is the *virtual location* of B_i and $\pi(i)$ is the *physical location* of B_i. The dummy data blocks, found at locations $\pi(N + 1), \ldots, \pi(N + S)$ in M, contain encryptions of arbitrary plaintexts of size B. For all blocks B_i where $i \in \{1, \ldots, N\}$, PM is initialized such that $\mathsf{PM}[i] = (\mathsf{M}, \pi(i))$. The protocol proceeds in two alternating phases: the *query* phase and the *oblivious shuffle* phase.

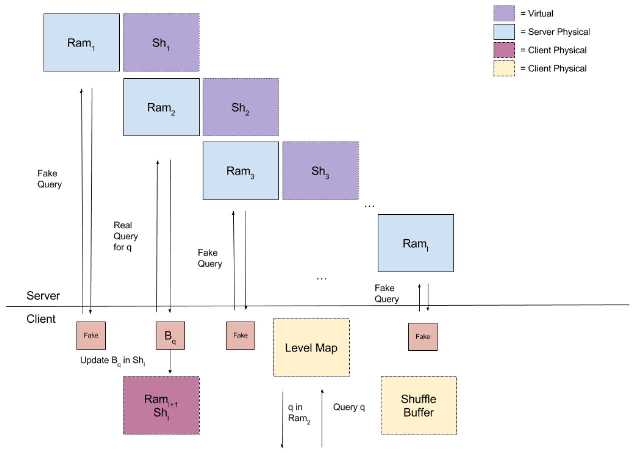
A query phase consists of S queries and the client keeps counters $\mathtt{dCnt}$ and $\mathtt{nSh}$ initialized to $N + 1$ and 1, respectively. These counters keep track of the next unqueried dummy block and of the next empty location in Sh. A read query for data block $q \in [N]$ is executed in the following way. First, the client retrieves $\mathsf{PM}[q] = (\mathtt{lev}_q, \mathtt{pos}_q)$. If $\mathtt{lev}_q = \mathsf{M}$ then B_q is stored as $\mathsf{M}[\mathtt{pos}_q]$. The client then queries the server for data block in position $\mathtt{pos}_q$ of M; the block received is decrypted and returned; and a copy is stored in $\mathsf{Sh}[\mathtt{nSh}]$. The position map is updated by setting $\mathsf{PM}[q] = (\mathsf{Sh}, \mathtt{nSh})$ and $\mathtt{nSh}$ is incremented. This implies that during the execution of the algorithm, there could be two copies of the same block, one on M and one in Sh. The block in M is considered *obsolete* and does not affect subsequent reads. Contrarily, the block in Sh is considered *fresh.* If instead, $\mathtt{lev}_q = \mathsf{Sh}$, then B_q is stored as $\mathsf{Sh}[\mathtt{pos}_q]$ and the data block in $\mathsf{Sh}[\mathtt{pos}_q]$ is returned. The client also queries the server for data block in physical location $\pi(\mathtt{dCnt})$ of M which is immediately discarded. Finally, $\mathtt{dCnt}$ is incremented. The dummy data block is retrieved to hide that the requested data block exists in Sh. The counter $\mathtt{dCnt}$ is incremented to guarantee that no dummy data block is retrieved twice.

A write query for data block q is executed exactly like a read query. However, instead of writing the original block B_q into Sh, the new block is written instead. The algorithm never writes any data block to M on the server and any updates to data blocks will be reflected in Sh on client memory. Also, no block of M is accessed more than once and exactly S different blocks are accessed.

After S queries, the protocol shifts to the oblivious shuffle phase. In the original protocol of [10], oblivious sorting algorithms are used. To modernize the construction, we use a S-Oblivious Shuffle algorithm on the $N + S$ data blocks of M according to a new pseudorandomly generated permutation σ. We assume the S accessed blocks of M are the S revealed indices of π. All obsolete data blocks on M will be replaced by their fresh version in Sh during the shuffle. After shuffling, Sh is emptied, as all of its data blocks have been moved back to M. Both $\mathtt{dCnt}$ and $\mathtt{nSh}$ are initialized to $N + 1$ and 1 respectively. Finally, π is replaced by σ and the protocol returns to the query phase.



(a) Querying for a data block which is in Sh_l.



(b) Querying for a data block which is on the server.

Figure 1: Diagrams of Query Algorithm for R-SQRT.

5 The R-SQRT ORAM

We present *Recursive SQRT ORAM* (*R-SQRT*) obtained by recursively applying SQRT ORAM on the shelter data blocks. At a high level, we are increasing the server storage cost to lower the amortized query cost compared to GO arising from shuffles.

5.1 Warm Up: Two Levels

We first describe a two-level version of our first construction. This will be extended to an arbitrary number of levels later.

Suppose that we are given N data blocks, $\mathsf{B} = B_1, \dots, B_N$. Fix a constant $c > 1$. We consider a first instance $\mathsf{Ram}_1 := \mathrm{GO}(\mathsf{B}, N/c)$, which we will denote as the *first level* of the construction. Recall, Ram_1 provides an ORAM for the $N_1 := N$ data blocks of B using $S_1 := N/c$ data blocks of client memory. Ram_1 stores $N_1 + S_1$ data blocks on the server in M_1, where N_1 of the data blocks are real (from B) and the remaining S_1 are dummy. Ram_1 stores S_1 data blocks on the client in Sh_1.

We recursively apply the GO construction on Sh_1. Specifically, we construct $\mathsf{Ram}_2 := \mathrm{GO}(\mathsf{Sh}_1, S_1/c)$, which is the *second level.* Ram_2 provides oblivious queries to the $N_2 := S_1$ data blocks of Sh_1 using $S_2 := S_1/c$ data blocks on the client. Again, Ram_2 consists of server storage, M_2, of $N_2 + S_2$ data blocks and client storage, Sh_2, of S_2 data blocks. We stop the recursion at Ram_2. As we shall see in the next section, we can proceed for $O(\log N)$ steps to achieve better performance.

Since Sh_2 is completely stored on the client storage, we may obliviously access blocks of Sh_2. For convenience, we can imagine that $\mathsf{Ram}_3 := \mathsf{Sh}_2$. We note that Sh_2 is not constructed using the GO protocol, but is just stored in the client. Similarly, it turns out that $\mathtt{nSh}_1$ is not needed in this construction since Sh_1 is stored using Ram_2. For simplicity, we will define $\mathtt{nSh} := \mathtt{nSh}_2$.

Taking a look at the entire memory organization, the server currently stores M_1 and M_2, using a total of

$$N_1 + S_1 + N_2 + S_2 = N(1 + 1/c + 1/c + 1/c^2).$$

For simplicity, it is useful to imagine $\mathsf{Sh}_0 := \mathsf{B}$. We note that Sh_0 and Sh_1 are virtual. To access Sh_0 and Sh_1, we use Ram_1 and Ram_2 respectively. Recall that each Ram_i uses a permutation π_i, which maps virtual locations to physical locations. Therefore, virtual location p of Ram_i currently stored physically in $\pi_i(p)$. For the client storage, we observe that, since we stopped at Ram_2, the N/c^2 data blocks of Sh_2 (= Ram_3) are physically stored on the client. In addition, the client will also store a position map, PM, which maps each to the level where that block is available for querying. Formally, for every block B_i, $\mathsf{PM}[i] = (\mathtt{lev}_i, \mathtt{pos}_i)$, which means that B_i is currently stored at $\mathsf{Ram}_{\mathtt{lev}_i}$ at virtual location $\mathtt{pos}_i$. Before any queries, the position map is initialized such that $\mathsf{PM}[i] = (1, \pi_1(i))$ for all blocks $i \in [N]$. We stress that there is only one PM for the entire construction.

Recall, the query algorithm for GO consists of searching in Sh and M. We slightly modify the query algorithm for our new construction. For Ram_1 (Ram_2), searching in Sh_1 (Sh_2) will now be handled by a query to Ram_2 (Ram_3). Therefore, a query to Ram_i will now only search M_i. Formally, a query to Ram_i for virtual location p will consist of retrieving $\mathsf{M}_i[\mathtt{pos}_p]$, where the physical location $\mathtt{pos}_p$ is stored at $\mathsf{PM}[p]$. A query to the entire protocol will be broken down to exactly one search to each level. In this case, there will be one query to each of $\mathsf{Ram}_1, \mathsf{Ram}_2$ and Ram_3.

Before the query phase, all N items of Sh_0 are stored in Ram_1, while Ram_2 is empty (since Sh_1 is also empty). To query for block B_q, first look into the position map, $\mathsf{PM}[q]$, to retrieve $(\mathtt{lev}_q, \mathtt{pos}_q)$. We now break up the algorithm into the different possible values of $\mathtt{lev}_q$.

If $\mathtt{lev}_q = 3$, we send a query for physical location $\mathtt{pos}_q$ to Ram_3. We will send queries to dummy blocks to each of Ram_1 and Ram_2. Formally, we query for virtual locations $\mathtt{dCnt}_1$ and $\mathtt{dCnt}_2$ (which translates to physical locations $\pi_1(\mathtt{dCnt}_1)$ and $\pi_2(\mathtt{dCnt}_2)$) to Ram_1 and Ram_2 respectively. Finally, the counters $\mathtt{dCnt}_1$ and $\mathtt{dCnt}_2$ are incremented by 1 so that we do not query the same dummy block twice. For write queries, the new block is uploaded $\mathsf{Ram}_3[\mathtt{nSh}]$ and PM is updated such that $\mathsf{PM}[q] = (3, \mathtt{nSh})$. Finally, $\mathtt{nSh}$ is incremented for both read and write queries.

If $\mathtt{lev}_q = 2$, then the desired block is stored in Ram_2 at physical location $\mathtt{pos}_q$. Note, since Ram_3 is stored on the client, we do not need to send a dummy query. We send a query to $\mathtt{pos}_q$ for Ram_2 and a dummy query to $\pi_1(\mathtt{dCnt}_1)$ to Ram_1. We then increment $\mathtt{dCnt}_1$ to ensure we do not query the same dummy block twice. Now, we move B_q to physical location $\mathtt{nSh}$ of Ram_3 and update $\mathsf{PM}[q] = (3, \mathtt{nSh})$. For write queries, we write the new block instead of B_q to $\mathsf{Ram}_3[\mathtt{nSh}]$. The update to the position map reflects that B_q is only available from Ram_3 now instead of Ram_2. Next, we increment $\mathtt{nSh}$ to represent the next empty location in Ram_3. The algorithm is identical for $\mathtt{lev}_q = 1$, by sending a dummy query to Ram_2 and a real query to Ram_1 instead.

During the query algorithm, we note that blocks only travel down to Ram_3. After S_2 queries, Ram_3 will become full. At this point, we perform an oblivious shuffle to move all S_2 data blocks of Ram_3 into Ram_2. After shuffling, we must update PM so that queries know the correct location of all blocks. When a block B_i is moved into physical location p of Ram_2, we set $\mathsf{PM}[i] = (2, p)$. Additionally, $\mathtt{dCnt}_2$ and $\mathtt{nSh}$ are reset to $N + 1$ and 1 respectively. Note, after another S_2 queries, Ram_3 will become full again. We can repeatedly shuffle the S_2 data blocks into Ram_2. After c shuffles, Ram_2 becomes full. At this point, we shuffle all S_1 data blocks of Ram_2 into Ram_1. Again, the position map entries for all S_1 data blocks of Ram_2 must be updated accordingly and $\mathtt{dCnt}_1$ is reset. In general, Ram_1 performs a shuffle every S_1 queries and Ram_2 performs a shuffle every S_2 queries.

5.1.1 Modified Oblivious Shuffle

In this section, we discuss the details of the K-Oblivious Shuffle algorithm. Recall, that shuffles moves N data blocks from the source array A to a destination array D. The N data blocks are arranged according to π in A prior to the shuffle, where K entries of π are revealed. After the shuffle, the N data blocks are arranged according to σ in D.

In the two level description, the shuffling algorithm is expected to update the position map PM as well as replace obsolete data blocks. To ensure obliviousness, any algorithm must download all N data blocks at least once. Note that if $\mathsf{PM}[i] = (\mathtt{lev}_i, \mathtt{pos}_i)$, then the fresh version of block B_i is found at physical location $\mathtt{pos}_i$ in $\mathsf{Ram}_{\mathtt{lev}_i}$. Upon uploading a block B_i to physical location p in Ram_j, we will append encrypted metadata including the block's identity i, its virtual location of insertion p and its level of insertion j. When block B is downloaded for a shuffle, the client checks using PM if B is fresh. If B is obsolete, B will be replaced with a dummy block. Otherwise, if B is fresh, we ensure to update PM to reflect the new level and physical location of B after the shuffle.

5.1.2 Reducing Online Query Bandwidth

In the two-level ORAM we have described, a query requires 2 blocks of bandwidth, one block from each of GO_1 and GO_2. We can reduce the query bandwidth to exactly 1 data block using the *XOR technique*, first introduced by Dautrich *et al.* [6] to reduce the bandwidth requirement of their proposed Burst ORAM construction to 1 block per request. The technique was also used in the construction of Ring ORAM [22]. We will describe the general technique as well as its application to our constructions.

Suppose that the server holds N ciphertexts computed using a deterministic encryption scheme. The client knows the keys used to compute the ciphertexts and the plaintexts of $N-1$ ciphertexts. What is the required bandwidth for the client to learn the one missing plaintext without leaking which ciphertext he is learning? In the simplest protocol, the server could send all N ciphertexts to the client. The client would decrypt the single ciphertext for which the plaintext is unknown. The XOR technique gives a more efficient protocol in which the server sends the XOR of all N ciphertexts. The client recreates the ciphertexts of the known $N-1$ encryptions and XORs them out. The final result is the ciphertext of the only unknown plaintext, which can be decrypted. The total bandwidth has been thus decreased to exactly one block.

To apply the XOR technique to in our two-level ORAM, we observe that at most one data block returned from GO_1 and GO_2 is real. We thus modify our construction so that all all dummy blocks are $\mathbf{0}$ and use a different key for each data block based on their level and physical location. This is needed to avoid that all dummy blocks being equal and being deterministically encrypted using the same key would result in the same ciphertext. Specifically, we will store two keys K_1 and K_2, for Ram_1 and Ram_2, respectively. The key for encrypting data block at physical location p in Ram_i will be $F(K_i, p)$, where F is a pseudorandom function. After shuffling, the keys K_1 and K_2 will be selected again uniformly at random. We stress that in our construction, each physical position of Ram_i holds at most one encrypted data block between two consecutive oblivious shuffles. Since keys are refreshed during shuffles, each key of the deterministic encryption scheme is used to encrypt at most one block. For a concrete instantiation, we can use AES encryption under Galois Counter Mode (GCM).

5.2 The Full Construction

We now describe the Recursive Square Root ORAM, R-SQRT for short. Diagrams of R-SQRT can be seen in Figure 1. The construction is parameterized by a constant $c > 1$ and integer $l \geq 1$. The parameter l represents the number of levels of recursion applied. Given input $\mathsf{B} = B_1, \ldots, B_N$ of N data blocks each of size B, R-SQRT(B, c, l) provides oblivious access to B. The construction from the previous section is obtained from R-SQRT$(\mathsf{B}, c, 2)$, by setting $l = 2$. As done previously, we will define $\mathsf{Sh}_0 := \mathsf{B}$ for simplicity.

Identical to before, $\mathsf{Ram}_1 := \text{GO}(\mathsf{Sh}_0, N/c)$, which consists of M_1 on the server and Sh_1 on the client, is the first level. We apply this recursion l times. So, $\mathsf{Ram}_2 := \text{GO}(\mathsf{Sh}_1, N/c^2), \ldots, \mathsf{Ram}_l := \text{GO}(\mathsf{Sh}_{l-1}, N/c^l)$ are levels $2, \ldots, l$ respectively. Finally, Sh_l is stored completely on the client allowing for oblivious access to Sh_l. Therefore, for simplicity, we denote $\mathsf{Ram}_{l+1} := \mathsf{Sh}_l$ as level $l+1$. Similar to previous section, we denote $N_i := N/c^{(i-1)}$ to be the number of data blocks that are stored in Ram_i. Also, $S_i := N/c^i$ is the size of the shelter of Ram_i (except Ram_{l+1}, which has no shelter).

As previous, R-SQRT(B, c, l) will store a position map, PM, on the client. For each block B_q, $\mathsf{PM}[i] = (\texttt{lev}_q, \texttt{pos}_q)$ represents that block B_q can be retrieved from $\mathsf{Ram}_{\texttt{lev}_q}$ at physical location $\texttt{pos}_q$.

In the exact manner as the previous section, we modify the queries of each instance of Ram_i. Each query of GO performs a search in the shelter Sh and M. We let queries to Ram_{i+1} handle the search to Sh_i. Therefore, each query to Ram_i for virtual position p consists of retrieving $\mathsf{M}_i[\texttt{pos}_p]$, where $\texttt{pos}_p$ is stored at $\mathsf{PM}[p]$. A query to R-SQRT(B, c, l) consists of exactly one query to each of $\mathsf{Ram}_1, \ldots, \mathsf{Ram}_{l+1}$.

Suppose that a query to B_q is requested. First, a lookup to the position map occurs to retrieve $\mathsf{PM}[q] = (\texttt{lev}_q, \texttt{pos}_q)$. For each level $i \neq \texttt{lev}_q$, a query to dummy block at physical location $\pi_i(\texttt{dCnt}_i)$ is sent and $\texttt{dCnt}_i$ is incremented. Note, no dummy queries need to be sent to Ram_{l+1}. One real query is performed to $\mathsf{Ram}_{\texttt{lev}_q}$ for physical location $\texttt{pos}_q$. Since, at most one real data block is queried, the XOR technique still applies. That is, the server retrieved blocks for $\mathsf{Ram}_1, \ldots, \mathsf{Ram}_l$ can be XOR'd together before being returned. We ensure that each Ram_i will have have their own key K_i like in the previous construction.

Using the same logic from the previous section, we see that a shuffle occurs for Ram_1 every $O(S_1)$ queries. In general, every Ram_i must shuffle every $O(S_i)$, except for Ram_{l+1} that never shuffles. During each shuffle, we ensure to re-initialized $\texttt{dCnt}_i$, $\texttt{nSh}$ to $N_i + 1$ and 1 respectively as well as refresh K_i. Pseudocode is provided in Appendix C.

5.2.1 Reducing the Number of Oblivious Shuffles

To further improve practical performance, we generalize the oblivious shuffle algorithm beyond a single input array. Consider an l-level construction after exactly S_i queries have been performed. In the description, we shuffle S_l blocks from Ram_{l+1} into Ram_l. Then, we move S_{l-1} blocks from Ram_l into Ram_{l-1}. In general, we sequentially shuffle S_j blocks from Ram_{j+1} into Ram_j for all $j = l, l-1, \dots, i+1, i$, requiring $l - i + 1$ oblivious shuffles.

We show that this movement can be achieved with exactly one shuffle. Note that all the blocks currently stored in $\mathsf{Ram}_{l+1}, \dots, \mathsf{Ram}_{i+1}$ will be moved into Ram_i. Conceptually, we can concatenate the current blocks of $\mathsf{Ram}_{l+1}, \dots, \mathsf{Ram}_i$ to be a single input array and construct a super permutation over the combined input array using $\pi_{l+1}, \dots, \pi_i$. Note, the input array will be larger than the output array as a result of the accumulated dummy blocks from the higher levels. We can ensure to remove all dummies and only introduce exactly S_i to Ram_i.

The decrease in the number of total shuffles does not affect the asymptotic behavior. Note, Ram_i still shuffles every $O(S_i)$ queries. However, the performance improvements in practice are significant as will be shown later.

5.2.2 Analysis of R-SQRT

Theorem 1. *R-SQRT is an ORAM.*

Proof. For each query q, exactly one query is issued to each of $\mathsf{Ram}_1, \dots, \mathsf{Ram}_l$. Also, for each Ram_i, exactly S_i queries are performed to S_i distinct array locations between oblivious shuffles. Furthemore, the S_i chosen locations are generated completely at random and independent from the data and access sequence. Therefore, R-SQRT is an ORAM protocol. □

For the rest of this section, we will assume that R-SQRT uses the CacheShuffle [21] version requiring $O(N^\epsilon)$ blocks of client memory, $O(N)$ blocks of server memory and $O(N)$ total bandwidth. The MelbourneShuffle [18] could also be used.

Theorem 2. *R-SQRT with l levels of recursion has amortized query bandwidth of $O(l)$, online query bandwidth of $O(1)$ using $O\left(\frac{N}{B} + \frac{N}{c^l} + N^\epsilon\right)$ data blocks of client storage and $O(N)$ blocks of server storage.*

Proof. Note, every Ram_i stores $N_i + S_i = \frac{N}{c^{(i-1)}} + \frac{N}{c^i}$ blocks on the server in M_i. Additionally, the CacheShuffle requires $O(N)$ server memory. Therefore, the total server storage is $\sum_{i=1}^{l} \left(\frac{N}{c^{(i-1)}} + \frac{N}{c^i}\right) + O(N) = O(N)$. The PM stores $O(1)$ number of words for each block. So, PM uses $O(N)$ words or $O(N/B)$ blocks of client storage. Ram_{l+1} requires $O(S_l) = O(N/c^l)$ blocks of client storage. Note, $\mathsf{Ram}_1, \dots, \mathsf{Ram}_l$ do not require any client storage. Also, the CacheShuffle requires $O(N^\epsilon)$ blocks of client memory. So, the total client storage is $O(N/B + N/c^l + N^\epsilon)$ data blocks.

Due to the XOR technique, our construction only requires $O(1)$ data blocks of bandwidth for online queries. The remaining bandwidth come from shuffling. It suffices to compute the amortized bandwidth in the period between shuffles of Ram_1. This period consists of $S_1 = N/c$ queries. Note, Ram_i shuffles every $S_i = N/c^i$ queries. Thus, Ram_i requires $c^{(i-1)}$ shuffles in this period. Recall that the CacheShuffle requires $O(N)$ bandwidth. Therefore, the total shuffling bandwidth of this period is $\sum_{i=1}^{l} c^{(i-1)} \cdot O\left(\frac{N}{c^i}\right) = O(l \cdot N)$. So, the amortized bandwidth is $O(l)$, since c is constant. □

Theorem 3. *There exists an ORAM construction that has $O(1)$ online query bandwidth and $O(\log N)$ amortized query bandwidth using $O(N/B)$ blocks of client storage and $O(N)$ blocks of server storage.*

Proof. By Theorem 2 for $c = \Theta(1)$ and $l = \Theta(\log N)$. □

6 Lowering Worst Case Bandwidth

We note that, in the worst case, R-SQRT requires $\Theta(N)$ bandwidth to satisfy a client access and this happens when Ram_1 is being shuffled. We show that a slight modification to the previous construction gives a new construction, that we call R-D-SQRT, with an $O(l)$ worst case bandwidth overhead. To achieve this, we perform the oblivious shuffle slowly over time. Specifically, at each level, we will perform some constant number of steps of the shuffle algorithm with each query. With this approach, we design the algorithm to ensure that the oblivious shuffling will be complete before the shuffled data is required.

6.1 Query Distributed Oblivious Shuffling

In this section, we describe how to distribute the operations of shuffling algorithms over many queries. We explain the protocol for a single level, Ram_i but all levels are performing steps of the oblivious shuffle with each query.

The algorithm consists of two phases: a Collection and Work phase. The Collection phase collects data blocks that require shuffling. Once all data blocks have been collected, the Work phase performs the necessary steps of the oblivious shuffle algorithm. Both the Collection and Work phase for Ram_i occur over exactly $S_i/2$ queries.

The basic intuition of our algorithm is that the shuffle of Ram_i is essentially moving the previously S_i queried data blocks into Ram_i. We double the number of shuffles occurring by forcing a shuffle every $S_i/2$ queries. The goal of the Collection phase is simply collect the last $S_i/2$ queries, which need to be shuffled into Ram_i. The client will upload an encryption of the queried data block, which will be stored in a queue C_i of $S_i/2$ size. The blocks placed in C_i are not involved in querying (or XOR technique), so they can be probabilistically encrypted using a single key.

After $S_i/2$ queries, the Work phase begins. The Work phase will perform the oblivious shuffle on input C_i concatenated with the $N_i + S_i$ data blocks of Ram_i. As mentioned previously, intermediate encryptions can be done using any probabilistic scheme and a single key. The final encryption before a block is placed into its correct location must be done using the properly derived key outlined in the XOR technique. We divide the necessary operations over $S_i/2$ queries to ensure well-distributed performance.

Note, during the Work phase, queries are still occurring that need to be collected for the next oblivious shuffle. Therefore, a Collection phase must be simultaneously occurring. In general, at any point, exactly one Work and Collection phase will be executing. Thus, exactly two oblivious shuffles are always simultaneously running.

We can further optimize the Collection phase by taking a look at the collective behaviour of all levels. In general, all levels are collecting the same data blocks. Therefore, we could instead keep a single queue C, which is global to all levels, of size S_1. C will store the last S_1 queried data blocks. The Work phase of any level may perform the oblivious shuffle over the correct subarray.

Theorem 4. *If the underlying shuffle algorithm is oblivious, the query distributed shuffle algortihm is also oblivious.*

Proof. The only difference between the underlying shuffle algorithm and the query distributed shuffle algorithm is the number of steps performed at each query. Since, the number of steps performed at each query is independent of the input, the query distributed shuffle algorithm is also oblivious. □

6.2 The R-D-SQRT ORAM

We now combined the above query distributed oblivious shuffling algorithms with an l-level construction to get a worst case $O(l)$ query cost. We will denote this construction as the Recursive Distributed Square Root (R-D-SQRT, for short). Unfortunately, we have to slighlty alter R-SQRT to use the query distribution shuffle.

Consider the shuffle algorithm after exactly S_i queries. For simplicity, we will refer queries indexed in their order. For example, q_1 will be the first query issued by the client. Only the first $S_i/2$ data blocks of queries, $q_1, \dots, q_{S_i/2}$ have been shuffled and are ready for use in Ram_i. The last $S_i/2$ queried blocks are not available yet in Ram_i. However, the blocks of queries $q_{S_i/2+1}, \dots, q_{S_i/2+S_{i+1}/2}$ are available in Ram_{i+1}, which is $S_i/4$ of the last queried blocks. Similarly, $S_i/8$ of the last queried blocks are available in Ram_{i+2}, $S_i/16$ in Ram_{i+3} and so forth. By our design, each of the last S_i queries are available in $\mathsf{Ram}_i, \dots, \mathsf{Ram}_{l+1}$, where the most recent queries appear in higher levels. Note, this is slightly different from the previous construction, which guaranteed that all previous S_i queries would be available at Ram_i.

Theorem 5. *R-D-SQRT has worst case query bandwidth $O(l)$ using $O\left(\frac{N}{B} + \frac{N}{c^l}\right)$ data blocks of client storage and $O(N)$ blocks of server storage.*

Proof. Note, we only increased the number of shuffles by a factor of two. Therefore, the total query bandwidth remains the same as R-SQRT. On the other hand, the worst case query bandwidth becomes the same as the amortized query bandwidth of R-SQRT, which is $O(l)$.

The only increases in memory are incurred by the Collection and Work phases. For the Collection phase, we only require an extra $O(N)$ data block of storage. Similarly, for each Ram_i, the Work phase requires $O(N_i) = O(N/c^{(i-1)})$ extra blocks of storage. Therefore, the total increase in server storage is $\sum_{i=1}^{l} O(N/c^{(i-1)}) = O(N)$. Finally, note the client storage does not change. □

6.2.1 Number of Server-Client Online Rounds

The number of data transmission rounds between the server and client is important for practical performance. Due to the possible physical distance between the server and client, a large number of rounds potentially increases overhead. Therefore, we wish to keep the number of rounds to be exactly one. We show this is possible for R-D-SQRT.

During the Collection phase, queried data blocks need to be encrypted and uploaded to the server. The most naive way to upload the queried data block would be to first query for the block and upload it. However, that requires two rounds of data exchange. Instead, the queried data block may be uploaded with the next query request.

Similarly, the steps of a Work phase can be split into download and upload requests. Note, these can also be done with just a single round of data transmission. Data blocks to be uploaded can be sent with the original query request. Download requests can also be sent with the original query request and the data blocks can be sent back with the original query response. Note, simultaneously running Work and Collection phase are independent so they may be combined with the original query request.

6.2.2 Flexibility of Bandwidth Distribution

In this section, we discuss the flexibility of the query distributed shuffling algorithm. The description of the query distributed algorithm describes a protocol that ensures all queries uses almost identical amounts of bandwidth. However, there is no reason that bandwidth costs could be distributed in other manners.

As an example, suppose that clients distinguishes queries as high and low priority. High priority queries should happen as quickly as possible, while low priority queries can incur larger overheads. Then, high priority queries could skip performing the steps of oblivious shuffling. Low priority queues could become slightly slower by executing all the steps that were skipped by the previous high priority requests. In fact, there is no reason that shuffling steps have to be even performed with a query. At any point in time, the client could perform the oblivious shuffle operations that were skipped by previous queries.

In general, all types of bandwidth cost distribution can be constructed. For example, queries can partitioned into a larger number of priority classes, each incurring the different bandwidth costs. Some clients might wish for extremely fast queries and perform all the shuffling costs during downtime (at night). The original R-SQRT construction suffices for this case. We could also distribute costs like Ring ORAM [22], where almost every query is extremely fast. However, every Q-th query incurs a larger overhead, where Q is small (for example, $Q = 8$). We use this cost distribution when comparing with Ring ORAM in a later section.

7 Practical Instantiations of R-SQRT

In this section, we provide a detailed analysis of the constants for R-SQRT and describe some techniques and tuning that yield practical instantiations.

7.1 CacheShuffle for R-SQRT

Throughout this section, our constructions will use CacheShuffle [21], specifically $\mathsf{CacheShuffle}_\mathsf{K}^\mathsf{K}$ ($\mathsf{CS}_\mathsf{K}^\mathsf{K}$). $\mathsf{CS}_\mathsf{K}^\mathsf{K}$ assumes that K indices of the input permutation π are revealed and $O(K)$ blocks of client memory. Suppose $\mathsf{CS}_\mathsf{K}^\mathsf{K}$ is shuffling N blocks and outputting M blocks. Note, $M \geq N$ due to dummy blocks. $\mathsf{CS}_\mathsf{K}^\mathsf{K}$ requires N and M blocks of download and upload. When applying to R-SQRT, it turns out the K revealed blocks are in client memory before $\mathsf{CS}_\mathsf{K}^\mathsf{K}$ starts. So, only $N - K$ blocks of download are required.

Let us consider the R-SQRT construction with $c = 2$ without a set value for l yet. Then, R-SQRT consists of $\mathsf{Ram}_1, \ldots, \mathsf{Ram}_{l+1}$. In the table below, we outline the number of real and dummy blocks available at each level. Additionally, for each Ram_i, we determine the times and frequency of shuffles that move blocks *into* Ram_i. Therefore, the entries of Ram_{l+1} for shuffling are not relevant since no blocks are ever moved into Ram_{l+1} by a shuffle.

	Real Blocks	Dummy Blocks	Shuffle Times	Shuffle Frequency
Ram_1	N	N	$N, 2N, 3N, \ldots$	N
Ram_2	$\frac{N}{2}$	$\frac{N}{2}$	$\frac{N}{2}, \frac{3N}{2}, \frac{5N}{2}, \ldots$	N
Ram_3	$\frac{N}{4}$	$\frac{N}{4}$	$\frac{N}{4}, \frac{3N}{4}, \frac{5N}{4}, \ldots$	$\frac{N}{2}$
$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$
Ram_{l-1}	$\frac{N}{2^{(l-2)}}$	$\frac{N}{2^{(l-2)}}$	$\frac{N}{2^{(l-2)}}, \frac{3N}{2^{(l-2)}}, \ldots$	$\frac{N}{2^{(l-3)}}$
Ram_l	$\frac{N}{2^{(l-1)}}$	$\frac{N}{2^{(l-1)}}$	$\frac{N}{2^{(l-1)}}, \frac{3N}{2^{(l-1)}}, \ldots$	$\frac{N}{2^{(l-2)}}$
Ram_{l+1}	$\frac{N}{2^{(l-1)}}$	0	N/A	N/A

Before any queries have arrived, Ram_1 is filled with N real blocks and N dummy block. All of $\mathsf{Ram}_2, \ldots, \mathsf{Ram}_{l+1}$ are empty. As blocks are queried, resulting data blocks are stored in Ram_{l+1}. Note, that Ram_{l+1} stores $N/2^{(l-1)}$ data blocks on the client. For convenience, suppose that $K := N/2^{(l-1)}$. These K blocks will be the revealed indices of π and exist in client memory before $\mathsf{CS}_\mathsf{K}^\mathsf{K}$ is executed. After K queries, $\mathsf{CS}_\mathsf{K}^\mathsf{K}$ moves all blocks from Ram_{l+1} into Ram_l. Note, that Ram_{l+1} is emptied immediately afterwards. In general, when shuffling into Ram_i, the levels $\mathsf{Ram}_{l+1}, \mathsf{Ram}_l, \ldots, \mathsf{Ram}_{i+1}$, are emptied.

We note that when $\mathsf{CS}_\mathsf{K}^\mathsf{K}$ moves blocks into Ram_i, Ram_i will always be empty (with the only exception being Ram_1). Suppose that Ram_i is not empty, which means the since the last shuffle into Ram_i, only higher levels have been shuffled into. If a level lower than Ram_i was shuffled, all the blocks of Ram_i would have been moved and Ram_i would be emptied. However, blocks will be shuffled into Ram_{i+1} before the next shuffle moves blocks into Ram_i. Thus, when $\mathsf{CS}_\mathsf{K}^\mathsf{K}$ shuffles into Ram_i, Ram_i will be empty.

Suppose that $\mathsf{CS}_\mathsf{K}^\mathsf{K}$ is moving blocks into Ram_i from $\mathsf{Ram}_{l+1}, \ldots, \mathsf{Ram}_{i+1}$ where $i \neq 1$. Ram_{l+1} is currently filled with the K last queried data blocks, all of which are stored on the client. Furthermore, Ram_l stores K real and dummy blocks respectively and K of these blocks were touched during queries. Our key observation is that touched blocks do not need to partake in shuffles. If the touched block was a dummy, then $\mathsf{CS}^{\mathsf{K}}_{\mathsf{K}}$ can ignore as we can always create new dummy blocks easily. For all touched real blocks, their fresh decrypted version exist in Ram_{l+1}. So all touched blocks can be ignored. Generalizing, it turns out half of the blocks of each of $\mathsf{Ram}_l, \ldots, \mathsf{Ram}_{i+1}$ have been touched. The source array of $\mathsf{CS}^{\mathsf{K}}_{\mathsf{K}}$ will be the untouched data blocks of each of $\mathsf{Ram}_l, \ldots, \mathsf{Ram}_{i+1}$. All queried data blocks are in Ram_{l+1} and already available on the client. So, we may use $\mathsf{CS}^{\mathsf{K}}_{\mathsf{K}}$ without downloading the K touched blocks first.

When $\mathsf{CS}^{\mathsf{K}}_{\mathsf{K}}$ shuffles into Ram_i, $\mathsf{CS}^{\mathsf{K}}_{\mathsf{K}}$ will get K queried data blocks already on the client and $N/2^{(l-1)} + N/2^{(l-2)} + \ldots + N/2^i = N/2^{(i-1)} - N/2^{(l-1)}$ untouched data blocks from $\mathsf{Ram}_l, \ldots, \mathsf{Ram}_{i+1}$. $\mathsf{CS}^{\mathsf{K}}_{\mathsf{K}}$ must output $N/2^{(i-2)}$ data blocks. Acute readers might notice that the number of output data blocks is larger than the number of total input blocks. However, half the output data blocks will be dummy blocks. To send a dummy block, the client will simply encrypt $\mathbf{0}$ and upload it to the server. So, the cost of $\mathsf{CS}^{\mathsf{K}}_{\mathsf{K}}$ is simply downloading the untouched data blocks and uploading the total number of output blocks since all queried data blocks are already on the client, which is $N/2^{(i-1)} - N/2^{(l-1)} + N/2^{(i-2)} = 3N/2^{(i-1)} - N/2^{(l-1)}$. We now look at the total cost of shuffling into Ram_i over N queries. Over N queries, blocks are moved into Ram_i at most $\frac{N}{(N/2^{(i-2)})}$ times. Therefore, the total cost is

$$\left(\frac{3N}{2^{(i-1)}} - \frac{N}{2^{(l-1)}}\right)(2^{(i-2)}) = N\left(\frac{3}{2} - 2^{(i-l-1)}\right).$$

The total cost of shuffling into $\mathsf{Ram}_2, \ldots, \mathsf{Ram}_l$ is

$$\begin{aligned}\sum_{i=2}^{l} N\left(\frac{3}{2} - 2^{(i-l-1)}\right) &= \frac{3N(l-1)}{2} - \frac{N}{2^l}\sum_{i=2}^{l} 2^{(i-1)} \\ &= \frac{3Nl}{2} - 3.5N + \frac{N}{2^{(l-1)}}.\end{aligned}$$

Note, Ram_1 is slightly different since it is not empty when being shuffled. There are N untouched blocks in Ram_1 in addition to the $N/2 + \ldots + N/2^{(l-1)} \leq N$ untouched blocks of $\mathsf{Ram}_2, \ldots, \mathsf{Ram}_l$. Finally, $2N$ data blocks must be placed into Ram_1 giving a total cost of $4N$. Noting that the online bandwidth of any single query is exactly one data block because of the XOR technique, we get the total amortized cost becomes $1.5l - 3.5 + \frac{1}{2^{(l-1)}} + 5 \approx 1.5(l+1)$.

7.2 CacheShuffle for R-D-SQRT

If we focus on using $\mathsf{CS}^{\mathsf{K}}_{\mathsf{K}}$, it turns out query distributing the shuffle is easier. In the previous section, we generalized the paradigm for all oblivious shuffling algorithms. Both the Collection and Work phases will occur over $K/2$ queries. Collection stores the previous $K/2$ queries into Ram_{l+1}. The Work phase downloads all untouched blocks in lower levels and uploads all needed blocks to the correct level.

It turns out the total amortized cost remains the same for R-SQRT and R-D-SQRT. R-D-SQRT requires twice as many shuffles in the same period for each level. But, each shuffle uses half as many untouched blocks as input and outputs half as many blocks.

To ensure uniform costs in R-D-SQRT, we either have to sacrifice one roundtrip per query or exactly one block of online bandwidth per query. However, we take the paradigm of Ring ORAM [22]. In Ring ORAM, most queries require exactly one block of bandwidth. Every Q (such as $Q = 4$) queries, an extra amount of bandwidth is required. So, we will ensure exactly one block of bandwidth for $Q - 1$ out of Q queries. The remaining query will perform the shuffling operations required for all Q queries. We note this trick is not necessary and we could always ensure a uniform cost of computation for each query.

7.3 Saving on Dummy Blocks: R-SQRT-AHE

Dummy blocks are a potential source of bandwidth saving in shuffling. Optimization for dummies have been used in Partition ORAM [26]. For a simple example, consider the case in which we have three blocks, numbered 1, 2 and 3. Suppose one of the three blocks is dummy and the other two are real. It is possible to upload all three blocks by using bandwidth equal to the size of only two blocks and still hide which block is dummy in the following way. The blocks (or actually their encryptions) can be seen as elements in a field $\mathbb{F}$. We compute the degree-one univariate polynomial P with coefficients in $\mathbb{F}$ that, when evaluated at the points in $\{1, 2, 3\}$ corresponding to the two real blocks, gives the correct value for the two real blocks. Then it is enough to upload the two coefficients of P and ask the server to evaluate P at $\{1, 2, 3\}$ to obtain the three blocks. We note that half of the blocks in each level of server storage in R-SQRT are dummy and thus there is a considerable gain in not having to upload all of them. This approach unfortunately does not interact well with the XOR technique since the value of the dummy blocks depend on the real blocks whereas the XOR technique relies on the value of the dummy blocks to be easily computable by the client.

We resort to a technique presented in [21] for dummy optimization in combination with Additively Homomorphic Encryption. Thus, we call the resulting construction R-SQRT-AHE. Specifically, we consider algorithm $\mathsf{DCS}^{\mathsf{K}}_{\mathsf{K}}$ from [21] that performs oblivious shuffle when a constant fraction of the output blocks are dummy. For $0 < \rho < 1$, if we are uploading M blocks with $R := \rho M$ real blocks, then $\mathsf{DCS}^{\mathsf{K}}_{\mathsf{K}}$ requires $(1+\epsilon)N$ blocks of upload bandwidth where ϵ decreases with N. So, $\mathsf{DCS}^{\mathsf{K}}_{\mathsf{K}}$ has smaller constants for larger N. Recall that $\mathsf{CS}^{\mathsf{K}}_{\mathsf{K}}$ required M blocks of upload bandwidth, meaning $\mathsf{DCS}^{\mathsf{K}}_{\mathsf{K}}$ is $\approx 1/2$ the cost of $\mathsf{CS}^{\mathsf{K}}_{\mathsf{K}}$ since $\rho = 1/2$ for R-SQRT.

Our AHE instantiation will be the Damgård-Jurik cryptosystem [5] using two primes of 1024 bits and an additive 1/10 ciphertext expansion rate on homomorphic operations. This results in ciphertexts of 2048 bits $* 10 \approx 2.56$ KB. Instead of the XOR trick, we use AHE to select the one block the client is interested in from the l blocks that are touched for each client access. More precisely, we consider blocks as consisting of C parts where each part is encrypted individually and has size about 2.56 KB. We send l ciphertexts, one for each block, at most one of which is the encryption of a 1 and the other are encryptions of 0. All C block parts are scalar multiplied with the ciphertext associated with the block. All resulting ciphertexts for each part are then added and the resulting C ciphertexts are returned to the client. Therefore, the total online bandwidth becomes $2.56l$ KB $+ 1.1B_s$ where B_s is the block size. Using the same analysis from above, we get the amortized bandwidth cost for each block is $\approx ((1 + \epsilon/2)l + 3.1 + \epsilon/2)B_s + 2.56l$ KB and therefore the overhead decreases with larger block sizes.

8 The TR-SQRT ORAM

In this section, we present the Twice-Recursive Square Root ORAM (or TR-SQRT, for short), which uses a second recursion to store the position map of size $O(N/B)$. Specifically, consider an instance of the Recursive Square Root ORAM protocol R-SQRT(B, c, l) on N input blocks $\mathsf{B} = B_1, \dots, B_N$. The client stores $O(N/B)$ blocks for PM. Instead, the client could instead store PM in another R-SQRT protocol leading to our next construction.

8.1 Modifying the Position Map

Before we can place the position map PM into R-SQRT, we must modify the R-SQRT construction slightly. In the original R-SQRT construction, the new physical locations of all moved blocks must be recorded in PM. Since PM was stored on the client, there was no bandwidth cost to reading or updating an entry in PM. If PM is stored in another R-SQRT construction, the cost becomes larger. We devise a new scheme that avoids updating entries in PM during shuffles, which is the basis of our next recursive solution.

Suppose that we index all incoming queries. The first queried block will have index 1, the second queried block will have index 2 and so on. We use counter $\mathtt{qCnt}$ to keep track of the total number of queries. Denote $\mathtt{prev}_i$ as the previous or latest query index for block B_i. It turns out that $\mathtt{prev}_i$ is extremely powerful. Note, B_i was not been queried since $\mathtt{prev}_i$ (or $\mathtt{prev}_i$ would be updated). Using $\mathtt{prev}_i$ and the total number of queries $\mathtt{qCnt}$, we can determine the level where the fresh version of B_i currently exists, denoted $\mathtt{lev}(\mathtt{prev}_i)$. We can also determine the number of data blocks that were queried before B_i and moved into $\mathsf{Ram}_{\mathtt{lev}(\mathtt{prev}_i)}$ during the same shuffle, which we denote $\mathtt{pos}(\mathtt{prev}_i)$. Therefore, it suffices for the position map to only store the latest query index for B_i, that is $\mathsf{PM}[i] = \texttt{prev}_i$. We modify the invariances of R-SQRT so block B_i is now stored at physical location $\mathsf{M}_{\texttt{lev(prev}_i\texttt{)}}[\pi_{\texttt{lev(prev}_i\texttt{)}}(\texttt{pos(prev}_i\texttt{)})]$. For convenience, we denote this the virtual location of $\texttt{pos(prev}_i\texttt{)}$ for $\mathsf{Ram}_{\texttt{lev(prev}_i\texttt{)}}$. Note that explicitly storing π_j for all levels would result in $O(N/B)$ client storage again. Instead, we can use the pseudorandom permutations described in Appendix A, which can succinctly store π_j using little memory.

This modification ensures that PM does not require updating during an oblivious shuffle. However, the shuffle needs to determine $\texttt{prev}_i$ of all data blocks B_i that are being shuffled. To avoid reading $\mathsf{PM}[i]$, we instead append an encryption of $\texttt{prev}_i$ to the metadata of each block B_i. Now, oblivious shuffling algorithms can read the index directly from the block without PM.

For concreteness, we will now fully define the modified R-SQRT protocol and how queries would be performed. We initialize the position map such that $\mathsf{PM}[i] = i$ for all data blocks B_i. Equivalently, we may assume that the blocks $B_1, \ldots, B_N$ are inserted sequentially. For convenience, we initialize $\texttt{qCnt}$ to $N + 1$. The counters $\texttt{dCnt}_i$ are still initialized to $N_i + 1$. Similarly, $\texttt{nSh}$ is initialized to 1. Block B_i is initially stored at $\mathsf{M}_1[\pi_1(i)]$.

Suppose a client queries for virtual location i, that is B_i. As usual, we retrieve $\mathsf{PM}[i] = \texttt{prev}_i$. We compute $\texttt{lev(prev}_i\texttt{)}$ and $\texttt{pos(prev}_i\texttt{)}$. For all levels $j \neq \texttt{lev(prev}_i\texttt{)}$, a dummy query to physical location $\pi_j(\texttt{dCnt}_j)$ in M_j is given (except Ram_{l+1}) and $\texttt{dCnt}_j$ is incremented by 1. Finally, a real query to the virtual location $\texttt{pos(prev}_i\texttt{)}$ of $\mathsf{M}_{\texttt{lev(prev}_i\texttt{,qCnt)}}$ is performed. The position map is updated so $\mathsf{PM}[i] = \texttt{qCnt}$ and $\texttt{qCnt}$ is incremented by 1. Throughout the rest of this section, we will assume R-SQRT uses these modifications. We note the XOR technique may still be used.

8.2 Warm Up: Two Levels

Like the previous section, we will briefly describe a two level construction. For convenience, denote $\text{R-SQRT}_1 := \text{R-SQRT}(\mathsf{B}, c, l)$. We note that R-SQRT_1 requires storing PM_1 on the client using $O(N/B)$ data blocks. Instead, we define $\text{R-SQRT}_2 := \text{R-SQRT}(\mathsf{PM}_1, c, l)$, which requires a position map of $O(N/B^2)$ data blocks on the client.

A query to R-SQRT_1 for B_i involves reading the value at $\mathsf{PM}_1[i]$ followed by updating $\mathsf{PM}_1[i]$ to the current query index, $\texttt{qCnt}$. We slightly modify the write operation to handle this in a single query. Previously, we expected the new block as an argument B'_q along with its virtual location, q. Instead, the client may only send q. The current block B_q will be returned to the client, which the server can use to construct B'_q. Therefore, the client can read the old value while overwriting in a single query. We first read virtual location $\lceil i/B \rceil$ of R-SQRT_2 for $\texttt{prev}_i$. We update the block such that $\mathsf{PM}_1[i] = \texttt{qCnt}$ and write the updated block back to virtual location $\lceil i/B \rceil$ in R-SQRT_2. Note, since PM_2 is stored on the client, queries to R-SQRT_2 remain the same. Using $\texttt{prev}_i$, queries to R-SQRT_1 continue identically as before without any interaction with PM_1.

8.3 The Full Construction

We extend the previous construction to l' levels. Formally, we describe TR-SQRT protocol, which is parameterized by a constant $c > 1$ and integers $l, l' \geq 1$. The parameter c and l becomes the parameters of the basis R-SQRT protocol. The value l' represents the number of recursion levels performed on the position map.

On input $\mathsf{B} = B_1, \ldots, B_N$, **TR-SQRT**$(\mathsf{B}, c, l, l')$ constructs $\text{R-SQRT}_1 := \text{R-SQRT}(\mathsf{B}, c, l)$. R-SQRT_1 produces PM_1. For $i \in \{2, \ldots, l'\}$, we construct $\text{R-SQRT}_i := \text{R-SQRT}(\mathsf{PM}_{i-1}, c, l)$. Therefore, $\mathsf{PM}_{l'}$, which consists of $O(N/B^{l'})$ data blocks will be stored on the client.

Queries to **TR-SQRT**(B, c, l, l') are surprisingly simple. Suppose we query for virtual location q. We first start running a query to R-SQRT_1 until we require reading from $\mathsf{PM}_1[q]$. In a single query, we read $\mathsf{PM}_1[q]$ and update the position map such that $\mathsf{PM}_1[q] = \texttt{qCnt}$ using a query to virtual location $\lceil q/B \rceil$ to R-SQRT_2. In general, there will be a single query to each R-SQRT_i to virtual location $\lceil q/B^{i-1} \rceil$ to read and update PM_{i+1}. The highest level results in a simple read to $\mathsf{PM}_{l'}$ which is on the client.

As previously shown, shuffles of R-SQRT$_i$ no longer require interaction with PM_i. Therefore, there is no extra cost of bandwidth outside the shuffling algorithm itself. Furthermore, the query distributed shuffling techniques may still be applied in the TR-SQRT construction.

Theorem 6. *TR-SQRT is an ORAM protocol.*

Proof. Note that each query to R-SQRT$_i$ is oblivious by Theorem 1 since none of the changes to the PM affect obliviousness. Furthermore, each query to **TR-SQRT**(B, c, l, l') performs exactly one query to each R-SQRT$_1, \ldots,$ R-SQRT$_{l'}$. Therefore, the TR-SQRT construction is also an ORAM. $\square$

In a somewhat different approach from R-SQRT, we will suppose the use of a general oblivious shuffling algorithm, S. Suppose that on N data blocks, S requires $S_t \cdot N$ data blocks of bandwidth using S_c blocks of client memory and S_s blocks of server memory.

Theorem 7. *TR-SQRT has amortized query bandwidth $O(S_t \cdot l \cdot l')$, online query bandwidth of $O(l')$ using $O(N/B^{l'} + S_c)$ blocks of client storage and $O(N + S_s)$ blocks of server storage.*

Proof. Each R-SQRT$_i$ requires $O(S_t \cdot l)$ amortized query bandwidth and $O(1)$ online query bandwidth. Therefore, TR-SQRT requires $O(S_t \cdot l \cdot l')$ and $O(l')$ amortized and online query bandwidth respectively. The only client memory is stored using $\mathsf{PM}_{l'}$ and S. So, the total client memory is $O(N/B^{l'} + S_c)$. Finally, each R-SQRT$_i$ requires $O(N/B^{(i-1)})$ blocks of server storage. Additionally, S requires $O(S_s)$ server blocks. Altogether, the total server storage is $\sum_{i=1}^{l'} \left(O(N/B^{(i-1)})\right) + O(S_s) = O(N + S_s)$. $\square$

We will denote protocol using distributed shuffling algorithms as TR-D-SQRT. Using the techniques of Section 6, we arrive at the following theorem.

Theorem 8. *TR-D-SQRT has worst case query bandwidth $O(S_t \cdot l \cdot l')$ using $O(N/B^{l'} + S_c)$ blocks of client storage and $O(N + S_s)$ blocks of server storage.*

8.4 Instantiations

To show the versatility of our ORAM construction, we provide a concrete example for the three classes of constructions that do not require the client to store a position map. Each construction is simply derived from TR-SQRT with the parameters $c = \Theta(1), l = \Theta(\log N)$ and $l' = \Theta(\log N/\log B)$ and by employing CacheShuffle with $O(N^\epsilon)$ and $\omega(\log N)$ client storage and the AKS sorting network, respectively. We note that in each case, the oblivious shuffling algorithm can also be query distributed.

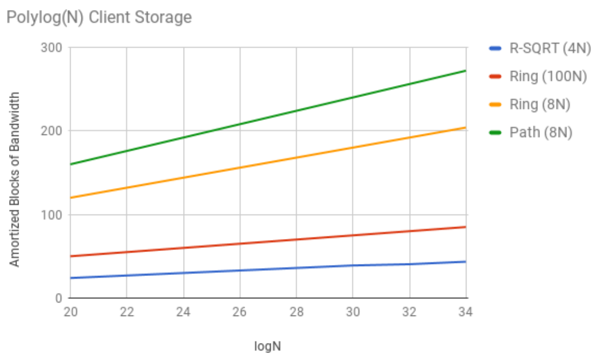
Theorem 9. *TR-D-SQRT with CacheShuffle using $O(N^\epsilon)$ blocks of client storage requires $O\left(\frac{\log^2 N}{\log B}\right)$ worst case bandwidth using $O(N^\epsilon)$ blocks of client storage and $O(N)$ blocks of server storage.*

Theorem 10. *TR-D-SQRT with CacheShuffle using $\omega(\log N)$ blocks of client storage requires $O\left(\frac{\log^3 N}{\log\log N \log B}\right)$ worst case bandwidth using $\omega(\log N)$ blocks of client storage and $O(N)$ blocks of server storage.*

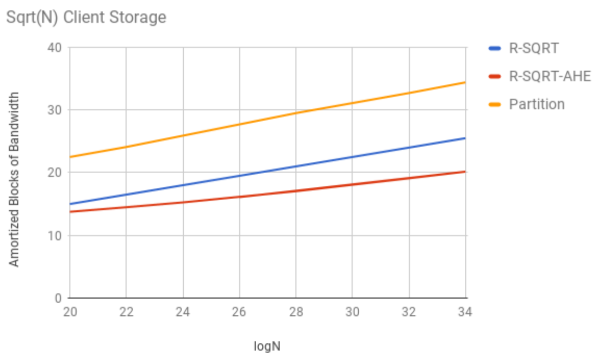
Theorem 11. *TR-D-SQRT with AKS requires $O\left(\frac{\log^3 N}{\log B}\right)$ worst case bandwidth using $O(1)$ blocks of client storage and $O(N)$ blocks of server storage.*

9 Experimental Results

We report on our experiments in comparing R-SQRT and R-SQRT-AHE with the current ORAMs with the best bandwidth overhead from among those of practical interest: Ring ORAM and Partition ORAM. The two constructions differ in the client memory requirement with Ring ORAM requiring the client to store $\omega(\log N)$ blocks and Partition ORAM requiring $\Theta(\sqrt(N))$ blocks. In addition, both constructions require the client to store the position map.



(a) Comparisons with $\omega(\log N)$ client storage.



(b) Comparisons with $O(\sqrt{N})$ client storage.

Figure 2: ORAM Comparisons.

All experiments are conducted on two identical machines, one for the server and one for the client. The machine used is a Ubuntu PC with Intel Xeon CPU (12 cores, 3.50 GHz). Each machine has 32 GB RAM with 1 TB hard disk. Our experiments will measure the time required on the client and server for each query (both online and amortized costs). Additionally, we will measure the bandwidth sent between the two machines on each query. All associated ORAM programs are implemented in C++. We use AES under GCM mode for encryption and decryption. The cryptographic functions are used from the BoringSSL library (a fork of OpenSSL 1.0.2). The length of keys used are 128 bits.

9.1 Comparing with Ring ORAM

We note that Ring ORAM [22] requires $\omega(\log N)$ blocks of client memory. We instantiate R-SQRT by picking $c = 2$ and $l = \log_2 N - \log_2 \log_2 N - 1$. This choice of parameters requires client memory $O(\log N)$, which is smaller than Ring ORAM and gives amortized bandwidth overhead $1.5(\log_2 N - \log_2 \log_2 N)$. On the other hand, Ring ORAM is able to achieve $2.5 \log_2 N$ using at least $100N$ blocks of server storage. The authors of Ring ORAM note that performance approaches $2 \log_2 N$ for larger server storage. However, since $100N$ blocks of server storage is prohibitive, we do not consider such an overhead for Ring ORAM. Using $8N$ blocks of server storage, Ring ORAM achieves $6 \log_2 N$ blocks of amortized bandwidth. Our experiments use 4 KB block sizes, the suggested reasonable size in [22].

Furthermore, Ring ORAM can decrease the amortized bandwidth by a factor of 2 at the cost of increasing client storage to $2\sqrt{N}$. However, when allowed to use more client storage, Ring ORAM requires more bandwidth than Partition ORAM (with which we compare to in the next section).

Comparing to the Ring ORAM instantiation with at least $100N$ blocks of server storage, R-SQRT and R-D-SQRT require half the bandwidth overhead of Ring ORAM with only $4N$ blocks of server storage. Using the more realistic instantiation of Ring ORAM with $8N$ blocks of storage, R-SQRT and R-D-SQRT show a 4x improvement in blocks of bandwidth while only using half the server storage. Results can be seen in Figure 2a with server storage in parenthesis.

9.2 Comparing with Partition ORAM

To compare with Partition ORAM [22], we pick the parameter l to ensure the same amount of client memory as Partition ORAM. Partition ORAM requires $O(N/B) + O(\sqrt{N})$ client memory. In the experiments of Partition ORAM, the authors consider their system with $4\sqrt{N}$ blocks of client memory and 64 KB blocks. We will use the same setting for all experiments. Therefore, we choose $l = \frac{\log_2 N}{2} - 1$ to also get $4\sqrt{N}$ client memory for both R-SQRT and R-SQRT-AHE. All of Partition, R-SQRT and R-SQRT-AHE require

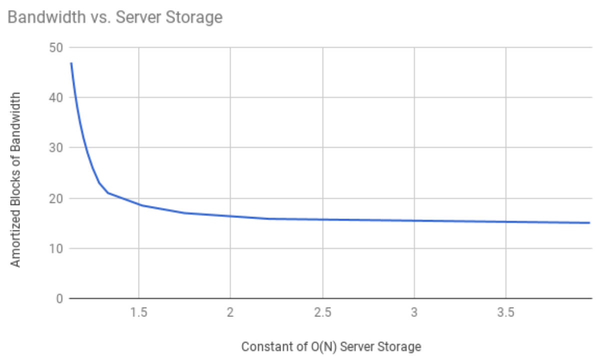


Figure 3: Bandwidth vs. Server Storage Tradeoff.

approximately $4N$ blocks of server storage. The bandwidth cost of R-SQRT is $0.75 \log_2 N$, while R-SQRT-AHE requires $0.6 \log_2 N$. On the other hand, the experimental results in Partition ORAM show that their systems require at least $\log_2 N$. R-SQRT uses 25% less bandwidth than Partition ORAM. R-SQRT-AHE uses at least 40% less bandwidth than Partition ORAM, but achieves better gains as N increases (such as 45% when $N = 2^{34}$). The results can be seen in Figure 2b.

9.3 R-SQRT Bandwidth vs. Server Storage

We investigate the effect of parameters c and l on the bandwidth of R-SQRT. For a fixed constant $c > 1$, l controls the amount of client storage required. Fixing $l = \frac{\log_c N}{2} - 1$, we ensure $4\sqrt{N}$ blocks of client storage is sufficient. Furthermore, we note that c controls the amount of server storage required. As c grows, server storage decreases. We run experiments for varying c using $N = 2^{20}$ blocks. We see that bandwidth increases as server storage decreases. To match the bandwidth cost of Partition ORAM (which uses $4N$ server storage), only $\approx 1.3N$ blocks of server storage is required using R-SQRT. The results can be seen in Figure 3.

9.4 The Power of Recursion

In this section, we demonstrate the power of recursion. We consider R-SQRT using a very small number of levels and compare with the original Square Root ORAM. Intuitively, the results should not be extremely different since three levels is not much more than the one level structure of the Square Root ORAM. However, experiments show huge improvements in just a three level construction.

To mirror the requirements of Square Root ORAM, we consider a three level variation of R-SQRT that uses $O(N)$ blocks of server storage and $O(\sqrt{N})$ blocks of client storage. We call this the 3-Level-R-SQRT construction. Using basic algebra, we see that $c = O(N^{1/6})$ satisfies these requirements. Note, to isolate the power of recursion, we do not use the XOR technique. Both constructions use $\mathsf{CS}_{\mathsf{K}}^{\mathsf{K}}$ with $K = \sqrt{N}$. In the experiments, we use blocks of size 1 KB. The results are shown in Figure 4. We see that 3-Level-R-SQRT has bandwidth costs that are significantly better than SQRT. Even with a couple extra levels, the power of recursion improves performance dramatically.

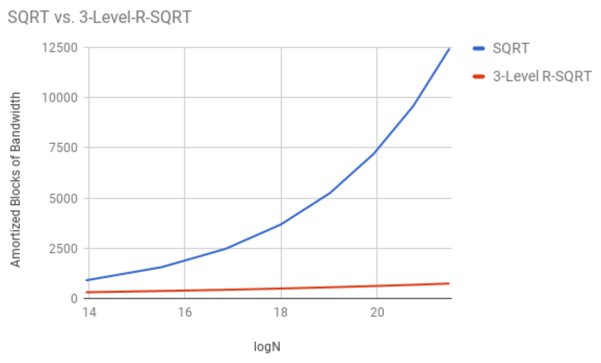


Figure 4: 3-Level R-SQRT and SQRT ORAM Comparison.

References

[1] M. Ajtai, J. Komlós, and E. Szemerédi. An O(N log N) sorting network. In *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing*, STOC '83, pages 1–9. ACM, 1983.

[2] K. E. Batcher. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*, AFIPS '68 (Spring), pages 307–314. ACM, 1968.

[3] J. Black and P. Rogaway. *CBC MACs for Arbitrary-Length Messages: The Three-Key Constructions*, pages 197–215. Springer Berlin Heidelberg, Berlin, Heidelberg, 2000.

[4] Z. Brakerski, C. Gentry, and V. Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, ITCS '12, pages 309–325, New York, NY, USA, 2012. ACM.

[5] I. Damgård and M. Jurik. A generalisation, a simplification and some applications of paillier's probabilistic public-key system. In *Proceedings of the 4th International Workshop on Practice and Theory in Public Key Cryptography: Public Key Cryptography*, PKC '01, pages 119–136, London, UK, UK, 2001. Springer-Verlag.

[6] J. L. Dautrich Jr, E. Stefanov, and E. Shi. Burst oram: Minimizing oram response times for bursty access patterns. In *USENIX Security Symposium*, pages 749–764, 2014.

[7] S. Devadas, M. van Dijk, C. W. Fletcher, L. Ren, E. Shi, and D. Wichs. Onion ORAM: A constant bandwidth blowup oblivious RAM. In *Theory of Cryptography - 13th International Conference, TCC 2016-A, Tel Aviv, Israel, January 10-13, 2016, Proceedings, Part II*, pages 145–174, 2016.

[8] S. Garg, P. Mohassel, and C. Papamanthou. *TWORAM: Efficient Oblivious RAM in Two Rounds with Applications to Searchable Encryption*, pages 563–592. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016.

[9] O. Goldreich. Towards a Theory of Software Protection and Simulation by Oblivious RAMs. In *STOC '87*, pages 182–194, 1987.

[10] O. Goldreich and R. Ostrovsky. Software Protection and Simulation on Oblivious RAMs. *J. ACM*, 43(3), 1996.

[11] M. T. Goodrich and M. Mitzenmacher. Privacy-preserving access of outsourced data via oblivious ram simulation. In *Proceedings of the 38th International Conference on Automata, Languages and Programming - Volume Part II*, ICALP'11, pages 576–587, Berlin, Heidelberg, 2011. Springer-Verlag.

[12] M. T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia. Oblivious ram simulation with efficient worst-case access overhead. In *Proceedings of the 3rd ACM Workshop on Cloud Computing Security Workshop*, CCSW '11, pages 95–100, 2011.

[13] M. T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia. Privacy-preserving group data access via stateless oblivious ram simulation. In *Proceedings of the Twenty-third Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '12, pages 157–167, Philadelphia, PA, USA, 2012. Society for Industrial and Applied Mathematics.

[14] V. T. Hoang, B. Morris, and P. Rogaway. *An Enciphering Scheme Based on a Card Shuffle*, pages 1–13. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[15] E. Kushilevitz, S. Lu, and R. Ostrovsky. On the (in)security of hash-based oblivious ram and a new balancing scheme. In *Proceedings of the Twenty-third Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '12, pages 143–156, Philadelphia, PA, USA, 2012. Society for Industrial and Applied Mathematics.

[16] B. Morris and P. Rogaway. *Sometimes-Recurse Shuffle*, pages 311–326. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

[17] B. Morris, P. Rogaway, and T. Stegers. How to encipher messages on a small domain. In *Advances in Cryptology - CRYPTO 2009, 29th Annual International Cryptology Conference*, volume 5677 of *Lecture Notes in Computer Science*, pages 286–302. Springer, 2009.

[18] O. Ohrimenko, M. T. Goodrich, R. Tamassia, and E. Upfal. The Melbourne Shuffle: Improving Oblivious Storage in the Cloud. In *Automata, Languages, and Programming: 41st International Colloquium, ICALP 2014, Copenhagen, Denmark, July 8-11, 2014, Proceedings, Part II*, pages 556–567, 2014.

[19] R. Ostrovsky. Efficient computation on oblivious rams. In *STOC '90*, pages 514–523, 1990.

[20] R. Ostrovsky and V. Shoup. Private information storage (extended abstract). In *STOC '97*, pages 294–303.

[21] S. Patel, G. Persiano, and K. Yeo. Cacheshuffle: An oblivious shuffle algorithm using caches. *CoRR*, abs/1705.07069, 2017.

[22] L. Ren, C. Fletcher, A. Kwon, E. Stefanov, E. Shi, M. van Dijk, and S. Devadas. Constants count: Practical improvements to oblivious ram. In *USENIX Security 15*, pages 415–430, 2015.

[23] T. Ristenpart and S. Yilek. *The Mix-and-Cut Shuffle: Small-Domain Encryption Secure against N Queries*, pages 392–409. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[24] E. Shi, T. H. H. Chan, E. Stefanov, and M. Li. Oblivious ram with o((logn)3) worst-case cost. In D. H. Lee and X. Wang, editors, *ASIACRYPT 2011*, pages 197–214, 2011.

[25] E. Stefanov and E. Shi. Oblivistore: High performance oblivious cloud storage. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, pages 253–267, Washington, DC, USA, 2013. IEEE Computer Society.

[26] E. Stefanov, E. Shi, and D. X. Song. Towards practical oblivious RAM. In *19th Annual Network and Distributed System Security Symposium, NDSS 2012, San Diego, California, USA, February 5-8, 2012*, 2012.

[27] E. Stefanov, M. van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas. Path oram: An extremely simple oblivious ram protocol. In *CCS '13*, pages 299–310, 2013.

[28] X. Wang, H. Chan, and E. Shi. Circuit oram: On tightness of the goldreich-ostrovsky lower bound. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 850–861, New York, NY, USA, 2015. ACM.

[29] X. S. Wang, Y. Huang, T.-H. H. Chan, A. Shelat, and E. Shi. Scoram: Oblivious ram for secure computation. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 191–202, New York, NY, USA, 2014. ACM.

[30] X. S. Wang, K. Nayak, C. Liu, T.-H. H. Chan, E. Shi, E. Stefanov, and Y. Huang. Oblivious data structures. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 215–226, New York, NY, USA, 2014. ACM.

[31] S. Zahur, X. S. Wang, M. Raykova, A. Gascón, J. Doerner, D. Evans, and J. Katz. Revisiting square-root ORAM: efficient random access in multi-party computation. In *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*, pages 218–234, 2016.

A Pseudorandom Permutations

Permutations are an important primitive for Oblivious RAM constructions that have been used in many previous protocols [10,26]. The naive approach to storing a permutation over N elements is very expensive, as it requires an array of N words. By information theoretical lower bounds, storing a true random permutation does require N words. These lower bounds can be avoided by generating pseudorandom permutations.

Black and Rogaway [3] present a pseudorandom permutation requiring only three stored keys. However, their approach only provided security guarantees for up to $N^{1/4}$ queries. A series of results [14, 17, 23] were able to push the security guarantees for up to N queries. Finally, the *Sometimes-Recurse Shuffle* by Morris and Rogaway [16] allowed accessing any permuted value using $O(\log N)$ AES evaluations by storing exactly one AES key. Throughout the rest of this work, we will assume that random permutations are generated using the Sometimes-Recurse Shuffle.

B Oblivious Shuffling

In this section we describe the K-Oblivious Shuffling algorithm that uses bandwidth $2N$ to shuffle, according to permutation σ, N data blocks $B_1, \ldots, B_N$ originally stored according to permutation π. We assume that the adversary knows the value of π for K touched blocks. We assume that K is smaller than the number of blocks that can be stored in client memory.

More precisely, we assume that array $\mathsf{sServer}$ of length N holds the data blocks stored according to π; that is, $\mathsf{sServer}[\pi(i)]$ hosts an encryption of block B_i. Similarly, we denote by $\mathsf{dServer}$ the array of length N that, at the end of the algorithm, will hold the N data blocks stored according to σ; that is, $\mathsf{dServer}[\sigma(i)]$ hosts an encryption of block B_i.

The algorithm takes as input the initial permutation π, the set $\mathsf{Touched}$ of indices of touched blocks, and their positions $\pi(\mathsf{Touched})$ in $\mathsf{sServer}$ and the new permutation σ. The algorithm works into two phases.

In the first phase, algorithm $\mathsf{CacheShuffle}$ downloads the encryptions of the touched blocks from server memory; that is, the encryptions of B_i stored as $\mathsf{sServer}(\pi(i))$, for all $i \in \mathsf{Touched}$. Each block is decrypted, re-encrypted using fresh randomness and stored in client memory. Throughout its execution, the $\mathsf{CacheShuffle}$ maintains the set tbDown of indices of data blocks that have not been downloaded yet. The set tbDown is initialized to $[N] \setminus \mathsf{Touched}$.

The second phase consists of N steps, for $i = 1, \ldots, N$. At the end of the i-th step, $\mathsf{dServer}[i]$ contains an encryption of block $B_{\sigma^{-1}(i)}$. Let us use s as a shorthand for $\sigma^{-1}(i)$. Three cases are possible. In the first case, an encryption of B_s is not in client memory, that is $s \in \mathsf{tbDown}$; then the algorithm sets $r = s$. If instead, an encryption of B_s is already in client memory, that is $s \notin \mathsf{tbDown}$, and $\mathsf{tbDown} \neq \emptyset$, the algorithm randomly selects $r \in \mathsf{tbDown}$. In both cases, the algorithm downloads an encryption of block B_r found at

$\mathsf{sServer}[\pi(r)]$, decrypts it and re-encrypts it using fresh randomness, stores it in client memory and updates tbDown by setting $\mathsf{tbDown} = \mathsf{tbDown} \setminus \{r\}$. In the third case in which $s \notin \mathsf{tbDown}$ and $\mathsf{tbDown} = \emptyset$, no block is downloaded. Then, an encryption of B_s is uploaded to $\mathsf{dServer}[i]$. Note that at this point, the client memory certainly contains an encryption of B_s.

B.1 Analysis

Theorem 12. CacheShuffle *uses K blocks of client memory, $2N$ blocks of server memory and $2N$ blocks of bandwidth.*

Proof. Initially the client downloads exactly K blocks and then at each step exactly one block is uploaded and at most one is downloaded. Therefore, client memory never exceeds K.

Each block is downloaded exactly once and uploaded exactly once. So bandwidth is exactly $2N$ blocks. $\square$

Theorem 13. CacheShuffle *is a K-Oblivious Shuffling.*

Proof. We prove the theorem by showing that the accesses of CacheShuffle to server memory are independent from σ, for randomly chosen π, given the sets $\mathsf{Touched}$ and $\pi(\mathsf{Touched})$. This is certainly true for the downloads of the first phase as they correspond to $\pi(\mathsf{Touched})$. For the second phase, we observe that at the i-th step an upload is made to $\mathsf{dServer}[i]$, which is clearly independent from σ. Regarding the downloads, we observe that the set tbDown initially contains $N - K$ elements and it decrease by one at each step until its empty. Therefore, it will be empty for the last K steps and thus no download will be performed. For $i \leq N - K$, the download of the i-th step is from $\mathsf{sServer}[\pi(r)]$. In the first case r is a random element of tbDown and thus independent from σ; in the second case, $r = s$ and thus the download is from $\mathsf{sServer}[\pi(s)]$, with $s = \sigma^{-1}(i)$. Since $s \notin \mathsf{Touched}$, for otherwise an encryption B_s would have been in client memory, the value $\pi(s)$ is independent from σ. $\square$

C Pseudocode

Init 1 Initialize R-SQRT protocol.

```
Input: $\lambda, c, l, B_1, \ldots, B_N$
  for $i := 1, \ldots, l$ do
    Generate $K_i$ randomly as key of length $\lambda$.
    Initialize $c_i = 0$.
    Initialize $\texttt{dCnt}_i = \frac{N}{c^{(i-1)}} + 1$.
    Initialize $\texttt{nSh} = 1$.
    Generate $\pi_i$ randomly.
    Initialize $\mathsf{M}_i$ on the server as array of size $\frac{N}{c^{(i-1)}}$.
  end for
  for $i := 1, \ldots, N$ do
    Initialize $\mathsf{PM}[i] = (1, \pi_1(i))$.
  end for
  for $i := 1, \ldots, N + S$ do
    Compute $j = \pi_1^{-1}(i)$.
    if $j \leq N$ then
      Initialize $\mathsf{M}_1[i] = \mathsf{Enc}(F(K_1, j), B_j)$.
    else
      Initialize $\mathsf{M}_1[i] = \mathsf{Enc}(F(K_1, j), \mathbf{0})$.
    end if
  end for
  for $i := 2, \ldots, l$ do
    for $j := 1, \ldots, N(1/c^{(i-1)} + 1/c^i)$ do
      Initialize $\mathsf{M}_i[j] = \mathsf{Enc}(F(K_i, j), \mathbf{0})$.
    end for
  end for
  Initialize $\mathsf{Ram}_{l+1} = \emptyset$.
```

Query **2** Query to R-SQRT protocol.

```
Input: $q, \mathsf{op}, B'_q$
  Retrieve $(\mathtt{lev}_q, \mathtt{pos}_q) = \mathsf{PM}[q]$.
  Update $\mathsf{PM}[q] = (l+1, \mathtt{nSh})$.
  Initialize $\mathsf{qList} = \varnothing$.
  for $i := 1, \dots, l$ do
    Increment $c_i$ by 1 modulo $\frac{N}{c^{(i-1)}}$.
    if $i \neq \mathtt{lev}_q$ then
      $\mathsf{qList}[i] = \pi_i(\mathtt{dCnt}_i)$.
    else
      $\mathsf{qList}[i] = \mathtt{pos}_q$.
    end if
  end for
  Send $\mathsf{qList}$ to the server.

  Initialize $R = \mathsf{M}_1[\mathsf{qList}[1]]$.
  for $i := 2, \dots, l$ do
    $R = R \oplus \mathsf{M}_i[\mathsf{qList}[i]]$.
  end for
  Return $R$ to the client.

  for $i := 1, \dots, l$ do
    if $i \neq l_q$ then
      $R = R \oplus \mathsf{Enc}(F(K_i, \pi_i(\mathtt{dCnt}_i)), \mathbf{0})$.
      Increment $\mathtt{dCnt}_i$ by 1.
    end if
  end for
  if $\mathtt{lev}_q = l + 1$ then
    $R = \mathsf{Ram}_{l+1}[\mathtt{pos}_q]$.
  end if
  $X = \mathsf{Dec}(F(K_{\mathtt{lev}_q}, \mathtt{pos}_q), R)$.
  if $\mathsf{op} = \mathsf{write}$ then
    $\mathsf{Ram}_{l+1}[\mathtt{nSh}] = B'_q$.
  else
    $\mathsf{Ram}_{l+1}[\mathtt{nSh}] = X$.
  end if
  Increment $\mathtt{nSh}$ by 1.
  for $i := 1, \dots, l$ do
    if $c_i = 0$ then
      Run $\mathsf{ModifiedObliviousShuffle}(\mathsf{Ram}_i, \dots, \mathsf{Ram}_{l+1})$.
      Set $\mathtt{nSh} = 0$.
      for $j := i + 1, \dots, l$ do
        Generate $K_j$ randomly as key of length $\lambda$.
        Set $\mathtt{dCnt}_j = 0$.
      end for
      break
    end if
  end for
  Return $X$.
```