Recursive Trees for Practical ORAM

Tarik Moataz[1], Erik-Oliver Blass[2] and Guevara Noubir[3]

[1]Telecom Bretagne, IMT, France,
and Colorado State University, CO, USA,
tmoataz@cs.colostate.edu
[2]Airbus Group Innovations, 81663 Munich, Germany,
erik-oliver.blass@airbus.com
[3]Northeastern University, MA, USA,
noubir@ccs.neu.edu

Abstract

We present a new, general data structure that reduces the communication cost of recent tree-based ORAMs. Contrary to ORAM trees with constant height and path lengths, our new construction r-ORAM allows for trees with varying shorter path length. Accessing an element in the ORAM tree results in different communication costs depending on the location of the element. The main idea behind r-ORAM is a recursive ORAM tree structure, where nodes in the tree are roots of other trees. While this approach results in a worst-case access cost (tree height) at most as any recent tree-based ORAM, we show that the average cost saving is around 35% for recent binary tree ORAMs. Besides reducing communication cost, r-ORAM also reduces storage overhead on the server by 4% to 20% depending on the ORAM's client memory type. To prove r-ORAM's soundness, we conduct a detailed overflow analysis. r-ORAM's recursive approach is general in that it can be applied to all recent tree ORAMs, both constant and poly-log client memory ORAMs. Finally, we implement and benchmark r-ORAM in a practical setting to back up our theoretical claims.

1 Introduction

Outsourcing data to external storage providers has become a major trend in today's IT landscape. Instead of hosting their own data center, clients such as businesses and governmental organizations can rent storage from, e.g., cloud storage providers like Amazon or Google. The advantage of this approach for clients is to use the providers' reliable and scalable storage, while benefiting from flexible pricing and significant cost savings.

The drawback of outsourced storage is its potential security implication. For various reasons, a client cannot always fully trust a cloud storage provider. For example, cloud providers are frequent targets of hacking attacks and data theft [14, 15, 31]. While encryption of data at rest is a standard technique for data protection, it is in many cases not sufficient. For example, an "adversary" might learn and deduce sensitive information just by observing the clients' access pattern to their data.

Oblivious RAM (ORAM) [10], a traditional technique to hide a client's access pattern, has recently received a revived interest. Its worst-cast communication complexity, dominating the monetary cost in a cloud scenario, has been reduced from being linear in the total number of data elements N to being poly-logarithmic in N [7, 8, 18, 20, 28–30]. With constant client memory complexity, some results achieve $O(\log^3 N)$ communication complexity, e.g., Shi et al. [28] and derivatives, while poly-logarithmic client memory allows for $O(\log^2 N)$ communication complexity, e.g., Stefanov et al. [30]. Although poly-logarithmic communication complexity renders ORAMs affordable, further reducing (monetary) cost is still important in the real

world. Unfortunately, closing the gap between current ORAM techniques and the theoretical lower-bound of $\Omega(\log N)$ [10] would require another major breakthrough.

Consequently, we focus on practicality of tree-based ORAMs. In general, to access an element in a tree-based ORAM, the client has to download the whole path of nodes, from the root of the ORAM tree to a specific leaf. Each node, also called a *bucket*, contains a certain number of entries (blocks). In case of constant client memory, there are $O(\log N)$ [28] entries per bucket, otherwise there are a small constant z many entries, e.g., $z = 5$ [30]. In any case, downloading the whole path of nodes is a costly operation, involving the download of multiple data entries for each single element to be accessed. Communication cost primarily depends on the height of the tree and, correlated, the number of entries per tree node and the eviction mechanism.

Contrary to recent κ-ary tree ORAMs, in this paper, we propose a new, different data structure called *recursive trees* that reduces tree height and therewith cost compared to regular trees. In addition to reducing communication overhead, recursive trees also improve storage overhead. Our new data structure r-ORAM offers variable height and therefore variable communication complexity, introducing the notion of *worst* and *best* cases for ORAM trees. r-ORAM is general in that it is a flexible mechanism, applicable to all recent tree-based ORAMs and possibly future variations thereof.

A second cost factor for a client is the total storage required on the cloud provider to hold the ORAM construction [21]. For an N element tree-based ORAM with entries of size l bits, the total storage a client has to pay for is at least $(2N - 1) \cdot \log N \cdot l$ [28] or $(2N - 1) \cdot z \cdot l$ [30]. In addition, a "map" translating ORAM addresses to leaves in the tree needs to be stored, too.

Technical Highlights: We present a new data structure reducing the *average* or *expected* path length, therefore reducing the cost to access blocks. Our goal is to support both constant and poly-log client memory ORAMs. Straightforward techniques to reduce the tree height, e.g., by using κ-ary trees [8], require poly-logarithmic client memory due to the more complex eviction mechanism. The idea behind our technique called r-ORAM is to store blocks in a recursive tree structure. The proposed recursive data structure substitutes traditional κ-ary ($\kappa \geq 2$) trees with better communication. Starting from an *outer* tree, each node in a tree is a root of another tree. After r trees, the recursion stops in a *leaf* tree. The worst-case path length of r-ORAM is equal to $c \cdot \log N$, with $c = 0.78$, yet this worst-case situation occurs only rarely. Instead in practice, the *expected* path length for the majority of operations is $c \cdot \log N$, with $c = 0.65$ for binary trees. The shortest paths in binary trees have length $0.4{\cdot}\log N$. In addition to saving on communication, the r-ORAM approach also saves up to 0.8 on storage due to fewer nodes in the recursive trees. To support our theoretical claims, we have also implemented r-ORAM and evaluated its performance. The source code is available for download [25].

r-ORAM is a general technique that can be used as a building block to improve any recent tree-based ORAM, both with $O(1)$ client memory such as Shi et al. [28], $O(\log N)$ client memory such as Stefanov et al. [30], and $O(\log^2 N)$ client memory such as Gentry et al. [8] – and variations of these ORAMs. In addition to binary tree ORAM, r-ORAM can also be applied to κ-ary trees. Targeting practicality, we abide from non-tree based poly-log ORAMs, such as Kushilevitz et al. [18]. While they achieve $O(\frac{\log^2 N}{\log \log N})$ worst-cast communication cost, their approach induces a large constant ~ 30.

2 Recursive Binary Trees

A Naive Approach: To motivate the rationale behind r-ORAM, we start by describing a straightforward attempt to reduce the path length and therewith communication cost. Currently, data elements added to an ORAM are inserted to a tree's root and then percolate down towards a randomly chosen leaf. As a consequence, whenever a client needs to read an element, the whole path from the tree's root to a specific leaf needs to be downloaded. This results in path lengths of $\log N$.

A naive idea to reduce path lengths would be to percolate elements to any node in the tree, not only leaves, but also interior nodes. To cope with added elements destined to interior nodes, the size of nodes, i.e., the number of elements that can be stored in such *buckets*, would need to be increased. At first glance, this reduces the path length. For

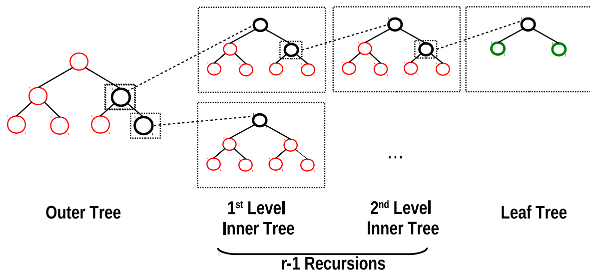


Figure 1: Structure of an r-ORAM

example, the minimum path length now becomes 1. However, the distribution of path lengths with this approach is biased to its maximum length of $\log N$: for a tree of N nodes, roughly $\frac{N}{2}$ are at the leaf level. Thus, the expected path length would be $\approx \log(N) - 1$, resulting in negligible savings. This raises the question whether a better technique exists, where the distribution of path lengths can be "adjusted".

r-ORAM Overview: We first give an overview about the structure of our new recursive ORAM constructions. In r-ORAM, parameter r denotes the recursion factor. Informally, an r-ORAM comprises a single *outer* binary tree, where each node (besides the root) is the root of an *inner* binary tree. Recursively, a node in an inner tree is the root of another inner tree, cf. Fig. 1. After the outer tree and $r-1$ inner trees, the recursion ends in a binary *leaf* tree. That is, each node (besides the root) in an $(r-1)^{\text{th}}$ inner tree is the root of a leaf tree. The fact that a root of a tree is never a (recursive) root of another tree simply avoids infinite duplicate trees.

Let the outer tree have y leaves and height $\log y$, where y is a power of two and log the logarithm base 2. Also, inner trees have y leaves and height $\log y$. Leaf trees have x leaves, respectively, and height $\log x$. The number of elements N that can be stored in an r-ORAM equals the total number of leaves in all leaf trees, similarly to related work on tree-based ORAM [28].

2.1 Operations

First, r-ORAM is an ORAM tree height optimization, applicable to any kind of tree-based ORAM scheme. r-ORAM follows the same semantics of previous tree-based ORAMs [7, 8, 28, 30], i.e., it supports the operations *Add*, *ReadAndRemove*, and *Evict*. For a given address a and a data block d, to simulate ORAM $Read(a)$ and $Write(a,d)$, the client performs a $ReadAndRemove(a)$ followed by $Add(a,d)$. For the correctness of tree ORAM schemes, the client has to invoke an *Evict* operation after every *Add* operation. Also, r-ORAM uses the same strategy of address mapping as the one defined in previous tree-based ORAMs – we detail this in Section 2.6. For now, assume that every leaf in r-ORAM has a unique identifier called *tag*. Every element stored in an r-ORAM is uniquely defined by its address a. We denote by $\mathcal{P}(t)$ the path (the sequence of nodes) containing the set of buckets in r-ORAM starting from the root of the *outer tree* to a leaf of a *leaf tree* identified by its tag t. If $\mathcal{P}(t)$ and $\mathcal{P}(t')$ represent two paths in r-ORAM, the least common ancestor, $LCA(t,t')$, is uniquely defined as the deepest (from the root of the outer tree) bucket in the intersection $\mathcal{P}(t) \bigcap \mathcal{P}(t')$. In this paper, we use the terms node and bucket interchangeably. Each bucket comprises a set of z entries. We start the description of r-ORAM by briefly explaining *Add*, *ReadAndRemove*, and *Evict* operations.

Operations *Add* and *ReadAndRemove* are similar to previous work, and details can be found in, e.g., Shi et al. [28].

- $Add(a,d)$: To add data d at address a in r-ORAM, the client first downloads and decrypts the bucket ORAM of the root of the outer tree from the server. The client then chooses a uniformly random tag t for a. The tag t uniquely identifies a leaf in r-ORAM where d will percolate to. The client writes d and t in an empty entry of the bucket, IND-CPA encrypts the whole bucket, and uploads the result to the root bucket. Finally, the recursive map is updated, i.e., the address a is mapped to t.

- $ReadAndRemove(a)$: To read an element at address a, the client fetches its tag t from the recursive map which identify a unique leaf in r-ORAM. The client then downloads and decrypts the path $\mathcal{P}(t)$. This algorithm outputs d, the data associated to a, or $\perp$ if the element is not found.

We apply r-ORAM to two different ORAM categories. The first one is a "memoryless setting",

where the client has constant size (in N) memory available. The second one, "with memory", assumes that the client has a local memory storage that is poly-log in N. For each category, we use different eviction techniques that we present in the following two paragraphs.

Constant Client Memory: The eviction operation is directly performed *after* an *Add* operation. Let us denote by t the leaf tag and by χ the eviction rate.

Evict(χ, t): Let $\mathcal{S} = \{\mathcal{P}$, such that $|\mathcal{P}| = |\overrightarrow{Rt}|\}$ be the set of all paths from the root R of the outer tree, to any leaf of a leaf tree that have the same length than the path from R to the leaf tagged with t. We call the distance from a node on a path in $\mathcal{S}$ its *level* L.

For each level $L, 1 \leq L \leq |\overrightarrow{Rt}|$, the client chooses from all nodes that are on the same level L, respectively, random subsets of $\chi \in \mathbb{N}$ nodes. For every chosen node, the client randomly selects a single block and evicts it to one of its children. The client write dummy elements to all other children to stay oblivious.

Poly-Log Client Memory: For the case of poly-log client memory, the eviction operation follows that of Gentry et al. [8] and Stefanov et al. [30]:

Evict(t): Let $\mathcal{P}(t)$ denote the path from the root of the outer tree R to the leaf with tag t. Every element of a node in $\mathcal{P}(t)$ is defined by its data and unique tag t'. For eviction, the client pushes every element in the nodes in the path $\mathcal{P}(t)$, which are tagged with leaf t', to the bucket $LCA(t, t')$.

The eviction operation is performed at the *same* time as an *Add* operation. Instead of storing the element in the root bucket during the *Add* operation, the client performs an *Evict*. Thus, they store, and at the same time evict, all elements as far as possible "down" on the path. Eviction can be deterministic [8] or randomized [30].

2.2 Security Definition

As in any ORAM construction, r-ORAM should meet the typical obliviousness requirement, restated below.

Definition 2.1. *Let* $\overrightarrow{a} = \{(op_1, d_1, a_1), (op_2, d_2, a_2), \ldots, (op_M, d_M, a_M)\}$ *be a sequence of M accesses (op_i, d_i, a_i), where op_i denotes a ReadAndRemove or an Add operation, a_i the address of the block, and d_i the data to be written if $op_i = Add$ and $d_i = \perp$ if $op_i = ReadAndRemove$.*

Let $A(\overrightarrow{a})$ be the access pattern induced by sequence $\overrightarrow{a}$, s_p is a security parameter, and $\epsilon(s_p)$ negligible in s_p. We say that r-ORAM is secure iff, *for any PPT adversary $\mathcal{D}$ and any two same-length sequences $\overrightarrow{a}$ and $\overrightarrow{b}$, access patterns $A(\overrightarrow{a})$ and $A(\overrightarrow{b})$, $|Pr[\mathcal{D}(A(\overrightarrow{a})) = 1] - Pr[\mathcal{D}(A(\overrightarrow{b})) = 1]| \leq \epsilon(s_p)$.*

As standard in ORAM, all blocks are IND-CPA encrypted. Every time a block is accessed by any type of operation, its bucket is re-encrypted.

2.3 Storage Cost

For a total number of N elements, we have N corresponding leaves in r-ORAM. To compute the total number of nodes ν, we start by counting the number of leaf trees in r-ORAM. For the outer tree, we have $2y - 2$ possible nodes which are the root for another recursive inner tree. Each inner tree has also $2y - 2$ nodes, and since we have $r - 1$ levels of recursion aside from the outer tree, the following equality holds:

$$N = (2y-2) \cdot (2y-2)^{r-1} \cdot x = (2y-2)^r \cdot x \quad (1)$$
$$= 2^r \cdot x \cdot (y-1)^r. \quad (2)$$

Each of the nodes in an r-ORAM is a *bucket* ORAM of size z, where z is a security parameter, e.g., $z = O(\log N)$ [28]. The total number of nodes ν, with N leaves, in an r-ORAM (main tree) is the sum of all nodes of all leaf trees plus the nodes of all inner trees, the outer tree, and its root, i.e.,

$$\begin{aligned} \nu(N) &= (2y-2)^r \cdot (2x-2) + \sum_{i=0}^{r} (2y-2)^i \\ &\stackrel{(1)}{=} (2N - 2 \cdot \frac{N}{x}) + \frac{(2y-2)^{r+1} - 1}{(2y-2) - 1} \\ &= 2N + (\frac{2y-2}{2y-3} - 2) \cdot \frac{N}{x} - \frac{1}{2y-3}. \end{aligned}$$

Thus, the total storage cost for r-ORAM is $\nu(N) \cdot z \cdot l$ with blocks (bucket entries) of size l bits. This storage does not take into account the position map. The total storage of the entire r-ORAM structure equals $\nu(N) \cdot z \cdot l + \sum_{i=1}^{\lceil \frac{\log N}{\log \beta} \rceil - 1} z \cdot \nu(\frac{N}{\beta^i}) \cdot$

$\log \frac{N}{\beta^{i-1}}$, where β is the position map factor. For $l = \omega(\log^2 N)$ the sum in the storage complexity is negligible. The total storage then equals $\nu(N) \cdot z \cdot l$.

For appropriate choices of x and y, discussed in the next section, r-ORAM reduces the storage cost in comparison with the $(2N-1) \cdot z \cdot l$ bits of storage of related work. So for example, with $x = 2$ and $y = 4$, the storage is equal to $\frac{8N}{5}$ resulting in a reduction by 20% of the number of nodes compared to existing tree-based ORAMs. However, this does not mean the same reduction for storage overhead. In fact, Section 4 will show that the size of the bucket can be reduced for Shi et al. [28]'s ORAM and increased for Path ORAM. Consequently, our storage saving varies between 4% to 20% depending on the ORAM r-ORAM.

As of Eq. (2), for a given number of elements N, r-ORAM depends on three parameters: recursion factor r, the number of leaves of an inner/outer tree y, and the number of leaves of a leaf tree x. We will now describe how these parameters must be chosen to achieve maximum communication savings.

2.4 Communication Cost

In ORAM, the "communication cost" is the number of bits transferred between client and server. We now determine the communication cost of reading an element in r-ORAM, e.g., during a *ReadAndRemove* operation. Reading an element implies reading the entire path of nodes, each comprising of z entries, and each entry of size l bits. In related work, any element requires the client to read a *fixed* number of $\log N \cdot l \cdot z$ bits. For the sake of clarity in the text below, we only compute the number of nodes read by the client, i.e., without multiplying by the number of entries z and the size of each entry l. Since the main data tree and the position map have different block sizes, computing the height of r-ORAM independently of the block size enable us to tackle both cases at the same time. At the end, to compute the exact communication complexity of any access we can just multiply the height with the appropriate block sizes, see Section 2.7.

A path going over a node on the ith level in the outer tree requires reading one bucket ORAM less than a path going over a node on the $(i+1)^{\text{th}}$ level in the outer tree. Consequently with r-ORAM, we need to analyze its best-case communication cost (shortest path), worst-case cost (longest path), and most importantly the average-case cost (average length).

The worst-case cost to read an element in r-ORAM occurs when the path comprises nodes of the full height of every inner tree until its leaf level, before finally reading the corresponding leaf tree. The worst-case cost $\mathcal{C}$ equals

$$\mathcal{C}(r, x, y) = r \cdot \log y + \log x. \tag{3}$$

The best-case occurs when the path comprises one node of every inner tree before reading the leaf tree. The best-case cost $\mathcal{B}$ equals

$$\mathcal{B} = r + \log x. \tag{4}$$

The worst-case cost in this setting is a function of three parameters that must be carefully chosen to minimize worst- and best-case cost. Theorem 2.1 summarizes how the recursion factor r, the number of leaves y in inner trees, and the number of leaves in leaf trees x have to be selected.

Minimizing the worst-case path length is crucially important, as it also determines the average path-length. We will see later that the distribution of paths' lengths (and therewith the cost) follows a normal distribution. That is, minimizing the worst case also leads to a minimal expected case and therewith the best configuration for r-ORAM. Similarly, as the paths' lengths follow a normal distribution, average and median cost are equivalent.

A client can use the minimal worst-case parameters to achieve the "cheapest configuration" for a r-ORAM structure storing a given number of elements N.

Theorem 2.1. *If $r = \log((\frac{N}{2})^{\frac{1}{2.7}})$, $x = 2$, and $y = \frac{1}{2} \cdot (\frac{N}{2})^{\frac{1}{r}} + 1$, the worst-case cost $\mathcal{C}$ is minimized and equals*

$$\mathcal{C} = 1 + 2.08 \cdot \log((\frac{N}{2})^{\frac{1}{2.7}}) \approx 0.78 \cdot \log N.$$

The best-case cost $\mathcal{B}$ is

$$\mathcal{B} = 1 + \log((\frac{N}{2})^{\frac{1}{2.7}}) \approx 0.4 \cdot \log N.$$

We refer the reader to Appendix A.1 for the proof.

2.5 Average-Case Cost

While the parameters for a minimal worst-case cost also lead to a minimal average-case cost, we still have to compute the average-case cost. The cost of reading an element ranges from $\mathcal{B}$, the best-case cost, to $\mathcal{C}$, the worst-case cost. Also, due to the recursive structure of the r-ORAM, the average-case cost of accessing a path is not uniformly distributed.

In order to determine the average-case cost, we count, for each path length i, the number of leaves that can be reached. That is, we compute the *distribution* of leaves in an r-ORAM with respect to their path length starting from the root of the outer tree. Let non-negative integer $i \in (\mathcal{B}, \mathcal{B}+1, \ldots, \mathcal{C})$ be the path length and therewith communication cost. We compute $\mathcal{N}(i)$, the number of leaves in a leaf tree that can be reached by a path of length i. Thus, the average cost, $\mathcal{A}_v$ can be written as $\mathcal{A}_v = \frac{\sum_{i=\mathcal{B}}^{\mathcal{C}} i \cdot \mathcal{N}(i)}{N}$, where N is the total number of elements and therefore leaves in the r-ORAM.

Theorem 2.2. *For:*

$$\mathcal{N}(i) = 2^i \cdot \sum_{j=0}^{r} (-1)^j \binom{r}{j} \binom{i - \log(x) - j \cdot \log(y) - 1}{r-1},$$

the average cost of a r-ORAM access is $\mathcal{A}_v = \frac{\sum_{i=\mathcal{B}}^{\mathcal{C}} i \cdot \mathcal{N}(i)}{N}$.

Proof. Counting the number of leaves for a path of length i is equivalent to counting the number of different paths of length i. The intuition behind our proof below is that the number of different paths of length i can be computed by the number of different paths in the r recursive trees $\mathcal{R}(i)$ times the number of different paths in the leaf tree, $\mathcal{N}(i) = \mathcal{R}(i) \cdot \mathcal{W}(i)$.

As stated earlier, the leaf tree has x leaves, $\mathcal{W}(i) = 2^{\log x} = x$.

To compute $\mathcal{R}(i)$, we introduce an array A_r of r elements. For a path $\mathcal{P}$ of length i, element $A_r[j]$, $1 \leq j \leq r$, stores the number of nodes in the jth inner tree that have to be read, i.e., the maximum level in the jth tree that $\mathcal{P}$ covers. For a path $\mathcal{P}$ of length i, we have $i = \sum_{j=1}^{r} A_r[j] + \log(x)$. For all j, $1 \leq A_r[j] \leq \log(y)$. For any path $\mathcal{P}$ of length i, we can generate $2^{i-\log(x)}$ other possible paths covering exactly the same number of nodes in every recursive inner tree, but taking different routes on each of them. For illustration, let path $\mathcal{P}$ go through two levels in the second inner tree – this means that there are actually 2^2 other paths that go through the same number of nodes. Therefore, if we denote the possible number of *original* paths of length i by $\mathcal{K}(i)$, the *total* number of paths equals $\mathcal{R}(i) = 2^{i-\log(x)} \cdot \mathcal{K}(i)$, for any integer $i \in \{\mathcal{B}, \ldots, \mathcal{C}\}$. We compute $\mathcal{K}(i)$, by computing the number of solutions of equation

$$A_r[1] + A_r[2] + \cdots + A_r[r] = i - \log x$$
$$\Leftrightarrow$$
$$(A_r[1] - 1) + \cdots + (A_r[r] - 1) = i - r - \log x \quad (5)$$

Computing the number of solutions of Eq. (5) is equivalent to counting the number of solutions of packing $i - r - \log x$ (indistinguishable) balls in r (distinguishable) bins, where each bin has a finite capacity equal to $\log(y) - 1$. Here, $A_r[j] - 1$ denotes the size of the bin. This can be counted using the stars-and-bars method leading to $\mathcal{K}(i) = \sum_{j=0}^{r} (-1)^j \binom{r}{j} \binom{i-\log(x)-j\cdot\log(y)-1}{r-1}$. With $\mathcal{N}(i) = 2^i \cdot \mathcal{K}(i)$, we conclude our proof. □

The average as formalized in the previous theorem does not give any intuition about the behavior of the average cost. For illustration, we plot the exact combinatorial behavior of the distribution of the leaf nodes. We present two cases that show the behavior of the leaf density, i.e., the probability to access a leaf in a given level in r-ORAM. We compute as well the average cost of accessing r-ORAM in two different cases, for $N = 2^{32}$ and $N = 2^{42}$, see Fig. 2.

We can simplify our average-case equation. The number of possibilities $\mathcal{K}$ of indistinguishable balls packing in distinguishable bins can be approximated by a normal distribution [2, 3]. For a given level $i \in \{\mathcal{B}, \cdots, \mathcal{C}\}$ we have

$$\mathcal{K}(i) \approx \frac{A}{s\sqrt{2\pi}} \cdot e^{-\frac{(i-r-\log(x)-\frac{c}{2})^2}{2s^2}}, \quad (6)$$

where $c = r \cdot (\log(y) - 1)$, $s = \frac{\frac{c}{2}+1}{\varpi}$, $A = r \cdot \log(y)$, and ϖ being the solution of the equation $\varpi \cdot e^{-\frac{\varpi^2}{2}} = \frac{\sqrt{2\pi}\cdot(\frac{c}{2}+1)}{A}$.

Since the number of leaves in the ith level of r-ORAM (over 2^i) follows a normal distribution with

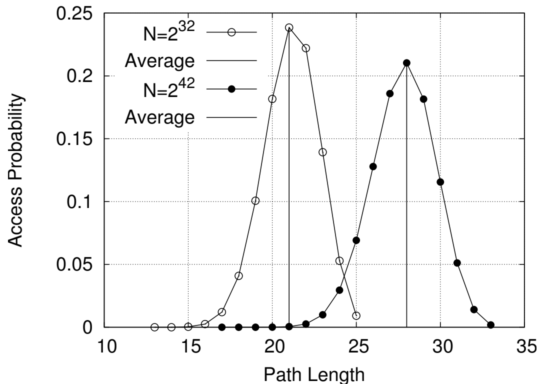


Figure 2: r-ORAM path length distribution

a mean $\frac{c}{2}$, which roughly equals the worst case over 2. The average case is the mean of the Gaussian distribution, therefore minimizing the worst case is equivalent to minimizing the average case. Thus, we can use the same parameters obtained in Th. 2.1 to compute the minimal value of the average case.

As both best- and worst-case path lengths are in $O(\log N)$, the average-case length is in $\Theta(\log(N))$. Further simplification of the average cost will result in very loose bounds. Targeting practical settings, we calculate the average page lengths for various configurations and compare it to related work in Table 1. While this table is based on our theoretical results, the actual experimental results of r-ORAM height are presented in Fig. 7.

Notice that our structure is a generalization of a binary tree for $x = 1$ and $y = 2$. Throughout this paper, the values x, y, and r equal the resulting optimal values given by Theorem 2.1.

2.6 r-ORAM Map addressing

In order to access a leaf in the r-ORAM structure, we have to create an encoding which uniquely maps to every leaf. This will enable us to retrieve the path from the root to the corresponding leaf node. The encoding is similar to the existing ones in [8, 28, 30]. The main difference is the introduction of the new recursion, which we have to take into account. Every node in the outer or inner trees can have either two children in the same inner tree or/and two other children as a consequence of the recursion. Consequently, we need *two bits* to encode every possible choice for each node from the root of the outer tree to a leaf. For the non-recursive leaf trees, one bit is sufficient to encode each choice.

For tree-based ORAM constructions with full binary-trees, to map N addresses, a $\log N$ bit size encoding is sufficient for this purpose. This encoding defines the leaf tag to which the real element is associated.

In r-ORAM, we define a vector v composed of two parts, a variable-size part v_v and a constant-size part v_c, such that $v = (v_v, v_c)$. For the encoding, we will associate to every node in the outer and inner trees two bits. For every node in the leaf tree only one bit. Above, we have shown that the shortest path to a leaf node has length $r + \log(x)$ while the longest path has length $r \cdot \log(y) + \log(x)$. Consequently, for the variable-size vector v_v, we need to reserve at least $2 \cdot r$ bits and up to $2 \cdot r \cdot \log(y)$ bits for the worst case.

The total size of the mapping vector v, $|v| = |v_v| + |v_c|$, is bound by $2r + \log(x) \leq |v| \leq 2r \cdot \log(y) + \log(x)$, which is in $\Theta(\log(N))$. Figure 3 shows an address mapping example for two leaf nodes. The size of the block in the r-ORAM position map is upper bounded by $2 \cdot \log N$ bits. Finally, the mapping is stored in a position map structure following the recursive construction in [30]. To access the position map, the communication cost has, as in r-ORAM, a best-case cost of $O(\mathcal{B} \cdot \log^2(n) \cdot z)$ bits and worst-case cost of $O(\mathcal{C} \cdot \log^2(n) \cdot z)$ bits, where z is the number of entries. This complexity is in term of bits, not blocks. For larger blocks, we can neglect the position map. In Path ORAM or Shi et al. constructions, the size to access the position map is in $O(z \cdot \log^3 N)$ which is the result of accessing a path containing $\log N$ buckets a $\log N$ number of time. Each bucket has z blocks where each has size equal to $O(\log N)$.

2.7 Communication complexity

First, we briefly formalize that the height can be seen as a multiplicative factor over all the recursion steps taking into consideration the eviction. Let N be the number of elements in the ORAM, denote by z the size of a bucket, β the position map factor, h the tree-structure height, l the block size and $\chi \geq 1$ the number of eviction, then for all tree-based ORAM the communication complexity C_T

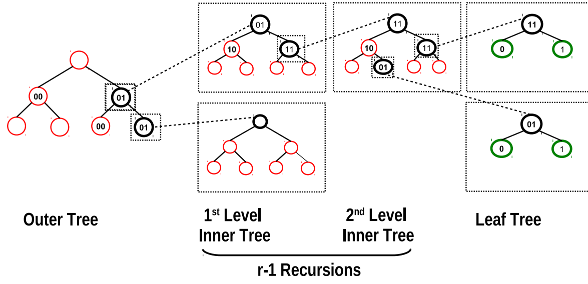


Figure 3: r-ORAM Map addressing

can be formulated as follows:

$$C_T = O(\underbrace{\chi \cdot z \cdot h \cdot l}_{\text{Data access}} + \underbrace{\beta \cdot z \cdot h \cdot \chi \cdot \log N}_{\text{Recursion}})$$

Reducing the height h decreases the entire communication overhead.

In this section, we are interested on computing the exact communication complexity (downloading/uploading) to access one block of size l. We will use for our computation the average height which is equal to $\approx 0.65 \cdot \log N$, see Table 1. In the following, we compute the communication complexities $C_{1,r}$ of r-ORAM over Path ORAM [30] and $C_{2,r}$ for r-ORAM over Shi et al. [28]. We denote the communication complexity for one access of Path ORAM and Shi et al. [28] by C_p and C_s. For an access, we download the entire path and upload it again. For Path ORAM, the eviction occurs at the same time when writing back the path. There is no additional overhead in the eviction. In the following equations, we take into consideration the variation of the bucket size. We later show in Section 4 that the size of r-ORAM applied to Path ORAM buckets increases by a factor of 1.2, while it expectedly decreases by 30% if applied to Shi et al. [28]. The variation of the bucket size impacts the height reduction in both cases as follows:

$C_{1,r} \approx 2 \cdot 0.65 \cdot \log N \cdot l \cdot z_{1,r} + \sum_{i=1}^{\lceil \frac{\log N}{\log \beta} \rceil - 1} 2 \cdot 0.65 \cdot z_{1,r} \cdot \log \frac{N}{\beta^i} \cdot \log \frac{N}{\beta^{i-1}} \approx 0.65 \cdot \frac{z_{1,r}}{z_p} \cdot C_p = 0.78 \cdot C_p$.

For Shi et al. [28]'s ORAM, for an eviction rate equal to 2, we are downloading 6 paths, plus the first one from which we have accessed the information. Thus, for each access, one has to download a total of 7 paths.

$C_{2,r} \approx 2 \cdot 0.65 \cdot 7 \cdot \log N \cdot l \cdot z_{2,r} + \sum_{i=1}^{\lceil \frac{\log N}{\log \beta} \rceil - 1} 2 \cdot 0.65 \cdot 7 \cdot z_{2,r} \cdot \log \frac{N}{\beta^i} \cdot \log \frac{N}{\beta^{i-1}} \approx 0.65 \cdot \frac{z_{2,r}}{z_s} \cdot C_s \approx 0.5 C_s$.

In this result, we make use of an approximation due to the size of the position map. In Section 2.6, we have shown that that to map an element, approximately $2 \cdot \log N$ bits is needed instead of $\log N$. We will show that these results match the experimental results in Section 5.

3 κ-ary Trees

So far, we have used a *binary* tree for the recursion in r-ORAM, i.e., leaf and inner trees are full binary trees. In this section, we extend r-ORAM to κ-ary trees, cf. Gentry et al. [8]. Generally, the usage of κ-ary trees reduces the height by a multiplicative factor equal to $\frac{1}{\ln(\kappa)}$. For example, if we choose a branching factor $\kappa = \log N$, the communication complexity decreases by a multiplicative factor equal to $\log(\log N)$. We will now show that applying r-ORAM to a κ-ary tree will further decrease the communication complexity compared to the original κ-ary construction.

For parameters x and y defined above, the number of elements N can be computed by calculating the number of nodes in the outer and inner κ-ary tree fora recursion factor r:

$$\begin{aligned} N &= (\sum_{i=0}^{\log_\kappa y} \kappa^i - 1)^r \cdot x = (\frac{1-\kappa^{1+\log_\kappa y}}{1-\kappa} - 1)^r \cdot x \\ &= (\frac{\kappa}{\kappa - 1} \cdot (y-1))^r \cdot x \qquad (7) \end{aligned}$$

Th. 3.1 shows how one should choose the recursion factor r, the height of the inner trees $\log y$ and leaf trees $\log x$ to minimize the cost of reading a path of κ-ary r-ORAM structure. In section 2.7, we have shown that the height factors over the total communication overhead reduction. Thus, any reduction applies for the the entire communication overhead computation. Also, we show in Section 4 based on our security analysis that r-ORAM's bucket size over Gentry et al. [8]'s ORAM decreases, thereby decreasing communication cost even more.

Theorem 3.1. *Let $f(\kappa) > 1$ be a decreasing function in κ. If $r = \log_\kappa((\frac{N}{\kappa})^{\frac{1}{f(\kappa)}})$, $x = 2$, and $y = \frac{\kappa-1}{\kappa} \cdot (\frac{N}{\kappa})^{\frac{1}{r}} + 1$, the optimum values for the*

best and worst-case cost equal

$$\mathcal{C} = 1+\log_\kappa((\frac{N}{\kappa})^{\frac{1}{f(\kappa)}})\cdot\log_\kappa((\kappa-1)\cdot\kappa^{f(\kappa)-1}+1), \text{ and}$$

$$\mathcal{B} = 1 + \frac{1}{f(\kappa)} \cdot \log_\kappa(\frac{N}{\kappa}).$$

The decreasing function f depends on the choice of κ, the branching factor. For $\kappa = 4$, $f(4) \approx 2$, while for $\kappa = 16$, $f(16) \approx 1.6$. The proof of the Theorem 3.1 is similar to the proof of Theorem 2.1, so we will only provide a sketch, highlighting the differences, see Appendix A.2.

Example: For $\kappa = 4$, the optimal values for the best and worst-case cost respectively equal $\mathcal{B} \approx 0.55 \cdot \log_\kappa N$ and $\mathcal{C} \approx 0.95 \cdot \log_\kappa N$.

4 Security Analysis

4.1 Privacy Analysis

Theorem 4.1. *r-ORAM is a secure ORAM following Definition 2.1, if every node (bucket) is a secure ORAM.*

Proof (Sketch). If the ORAM buckets are secure ORAMs, we only need to show that two access patterns induced by two same-length sequences $\overrightarrow{a}$ and $\overrightarrow{b}$ are indistinguishable. To prove this, we borrow the idea from Stefanov et al. [30] and show that the sequence of tags t in an access pattern is indistinguishable from a sequence of random strings of the same length.

To store a set of N elements, r-ORAM will comprise N leaves and N different paths. During *Add* and *ReadAndRemove* ORAM operations, tags are chosen uniformly and independently from each other. Since the access pattern $A(\overrightarrow{a})$ induced by sequence $\overrightarrow{a}$ consists of the sequence of tags (leaves) "touched" during each access, an adversary observes only a sequence of strings of size $\log N$, chosen uniformly from random. The nodes in r-ORAM are bucket ORAMs, i.e., for an ORAM operations they are downloaded as a whole, IND-CPA re-encrypted, and uploaded exactly as in related work, they are secure ORAMs. □

4.2 Overflow probability

To show that our optimization is a general technique for tree-based ORAMs, we compute the overflow probabilities of buckets and stash for both constant and poly-logarithmic client memory schemes. Specifically, we analyze r-ORAM for the constructions by Shi et al. [28], Gentry et al. [8], and Stefanov et al. [30]. Surprisingly, for the first scheme, we are able to show in Theorem 4.4 that r-ORAM will reduce the bucket size while maintaining the exact same overflow probability. This is significant from a storage and communication perspective: it shows that r-ORAM can improve storage and communication overhead not only due to a reduction of the number of nodes (as shown in Section 2.3 and 2.7), but also by reducing the number of entries in every bucket.

For the second scheme which uses a "temporary" poly-log stash during eviction (needed to compute the least common ancestor), we show in Theorem 4.6 that r-ORAM offers improved communication complexities and a slightly better bucket size.

Finally for Path ORAM, we prove that the stash size increases only minimally and remains small. In Theorem 4.8, we show that this small increase is outweighed by smaller tree height.

We now determine the ORAM overflow probability for two cases, (1) r-ORAM applied to the constant client memory approach, and (2) to the poly-log client memory approach. For the first case, we consider an eviction similar to the one used by Shi et al. [28]. That is, for every level, we will evict χ buckets towards the leaves, where χ is called the eviction rate. For the second case, we consider a deterministic reverse-lexicographic eviction similar to Gentry et al. [8] and Fletcher et al. [7]. In particular, for the poly-logarithmic setting, we investigate the application of r-ORAM over two different schemes. The first case consists of the application of r-ORAM over the scheme by Gentry et al. [8]. For this, we study the overflow probability of the buckets and we show that the recursive structure offers better bucket size bounds. The second case represents the application of r-ORAM over Path ORAM. We determine the overflow probability of the memory, dubbed *stash*, where each bucket in r-ORAM has a constant number of entries z. Using deterministic reverse-lexicographic eviction greatly simplifies the proof while insuring the same bounds as the ones in randomized eviction [30].

To sum up, we are studying three different cases. (1) r-ORAM over Shi et al. [28] construction, (2) r-ORAM over Gentry et al. [8] construction and

(3) r-ORAM over Path ORAM [30]. For the first two, we have to quantify the bucket size while for the third one we have to quantify the stash size and the size of the bucket as well. For each setting, an asymptotic value of the number of entries z is provided. The main difference between the computation of the overflow probability in r-ORAM and related work is the irregularity of path lengths of our recursive trees. To better understand the differences, we start by presenting a different model of our construction in 2-dimensions.

Description: A 2-dimensional representation of r-ORAM consists of putting all the recursive inner trees as well as the leaf trees in the same dimension as the outer tree. Consequently, the outer tree, the recursive inner trees, as well as the leaf trees will together constitute only one single tree we call the *general tree*. The main difficulty of this representation is to determine to which level a given recursive inner tree is mapped to in the general tree.

The general tree, by definition, will have leaves in different levels. This can be understood as a direct consequence of the recursion, i.e., some leaves will be accessed with shorter paths compared to others. Moreover, the nodes of the recursive trees will be considered as interior nodes of the general tree with either 4 children or 2 children. Any interior node of an inner or outer tree is a root for a recursive inner tree which means that any given interior node of an inner/outer tree has 2 children related to the recursion as well as another 2 children related to its inner/outer tree. These 4 children belong to the same level in our general tree.

Also, leaf nodes of inner or outer trees have only 2 children. Ultimately, we will have different distributions of interior nodes as well as leave nodes throughout the general tree. In the following, we will use the term of *interior node* as well as a *leaf node* in the proofs of our theorems to denote an interior or leaf node of the general tree. Figure 4 illustrates the topology of the general tree model of r-ORAM.

In the ith level, we may have leaf nodes as well as interior nodes. Also, the leaf/interior nodes reside in different levels with different non-uniform probabilities. Therefore, we will first approximate the distribution of the nodes in a given level of the r-ORAM structure by finding a relation between the leaf nodes and interior nodes of any level of r-ORAM. Then, we compute the relation between the number of nodes in the ith and $(i+1)^{\text{th}}$ level. This last step will help us to compute the expected value of number of nodes in any interior nodes in poly-log client memory scenarios. Finally we will conclude with the overflow theorems and their proofs for each scenario.

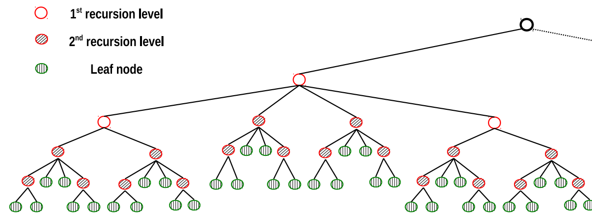


Figure 4: Structure of an r-ORAM

We present a relation between $I(i)$, the number of interior nodes, and $\mathcal{N}(i)$, the number of leaf nodes, for a level $i > r$, where r is the recursion factor. Notice that, for other levels $i \le r$, there cannot be leaf nodes. Also, the leaves of the general tree are the leaves of the leaf trees. The maximum value of i equals the worst case $\mathcal{C}$.

Lemma 4.2. *Let* $f(r,x,y) = \frac{1+r\log(y)-r-2\log(x)}{2s^2}$ *and* $s > 0$. *For any* $i > r$, $e^{-f(r,x,y)} \le \frac{I(i)}{\mathcal{N}(i)} \le 2^{-\log(x)} \cdot r$.

The proof of the lemma is in Appendix A.3.

We will now show that, once we have a relation between leaves and interior nodes of the same level, finding the relation between any nodes of two different levels will be straightforward. We write the number of nodes as a sum of leaf nodes and interior nodes, such that $L(i) = \mathcal{N}(i) + I(i)$. Recall that for $i \le r$, we have $\mathcal{N}(i) = 0$. We write $\mu = \frac{L(i+1)}{L(i)}$ (this will represent the expected value of the number of real elements in any interior nodes in Theorem 4.6). We present our result in the following lemma.

Lemma 4.3. *Let* $\mu = \frac{L(i+1)}{L(i)}$ *and* $X(i) = 1 - \frac{\mathcal{N}(i)}{L(i)}$. *For* $1 \le i \le \mathcal{C}$, μ *is bounded by* $2 \cdot X(i) \le \mu \le 4 \cdot X(i)$.

We refer the reader to the Appendix A.4 for the proof.

From this result, for $i \le r$, we have $2 \le \mu \le 4$, as $\mathcal{N}(i) = 0$.

We are now ready to present our three main theorems: the first one will tackle the constant client

memory setting, and we compute the overflow probability of interior nodes. The overflow probability computation for leaf nodes, either for constant client memory or with poly-log client memory, is similar to the one presented by Shi et al. [28], based on a standard balls-into-bins argument. We omit details for this specific case. The last two theorems tackle tree Based ORAM constructions with memory.

Constant client memory: First, we compute the overflow probability of interior nodes. Then, a corollary underscoring the number of entries z will be presented.

Theorem 4.4. *For eviction rate χ, if the number of entries in an interior node is equal to z, the overflow probability of an interior node in the ith level is at most θ_i^z, where, for $i \leq r$ and $s = \lceil \log_4(\chi) \rceil$, $\theta_i = \frac{2^s}{2\chi}$, and for $i > r$: $\theta_i = \frac{2^s}{2\chi} \cdot (\frac{1}{1+\frac{x}{r}})^{i-r}$.*

We refer the reader to Appendix A.5 for the proof.

In practice, the eviction rate χ equals 2. So, s is then equal to 1. In this case, the number of entries z in each bucket has the following size.

Corollary 4.5. *r-ORAM with N elements overflows with a probability at most $\omega \ll 1$ if the size of each interior bucket z in the ith level equals $\log \frac{N}{\omega}$ for $i \leq r$ and $z \approx \frac{1}{i-r+1} \cdot \log \frac{N}{\omega}$ for $i > r$.*

Sketch. By applying the union bound over the entire r-ORAM interior buckets, the probability of overflow is at most $N \cdot \theta_i^z$. Setting this value to the target overflow ω gives us the results for both underlined cases in Th. 4.4. For the second equality, the approximation follows from the remark $\log(1 + \frac{x}{r}) < 1$, since $x \leq r$ in our optimal setting of Th. 2.1. □

The size of the internal buckets in r-ORAM are smaller compared to those of Shi et al. [28] by a multiplicative factor of approximately $\frac{1}{i-r+1}$ for $i > r$.

For $\omega = 2^{-64}$, $N = 2^{20}$, and $r = 7$, the size of the bucket equals 84 blocks for $i \leq 7$ while for, e.g., $i = 11$, the bucket size equals ≈ 17 blocks. For $i \leq r$, the bucket size is equal to the constant client memory construction, i.e., in $O(\log \frac{N}{\omega})$.

Poly-logarithmic client memory: Let us now tackle the case where r-ORAM is applied over tree ORAMs with poly-logarithmic client memory. For this, we consider two scenarios. The first deals with r-ORAM applied over Gentry et al. [8]'s ORAM. The second one deals with r-ORAM over Path ORAM. In both cases, our overflow analysis is based on a deterministic reverse lexicographic eviction.

Th. 4.6 determines the overflow probability of buckets in r-ORAM over Gentry et al. [8] scheme. For each access, the eviction is done deterministically independently of the accessed path. We show that the overflow probability varies for buckets in different levels due to the interior/leaf node distribution. The parameter δ represents the unknown that should be determined for a given (negligible) overflow probability.

Theorem 4.6. *Let $f(r,x,y) = \frac{1+r\log(y)-r-2\log(x)}{c}$, and $c > 0$. For any $\delta > 0$, for any interior node v, the probability that a bucket has size at least equal to $(1+\delta)\cdot\mu$ is at most $e^{-\frac{\delta^2\cdot\mu}{2+\delta}}$, where $F_1 \leq \mu \leq F_2$.*
For $i \leq r$: $F_1 = 2$ and $F_2 = 4$,
for $i > r$:

$$F_1 = 4\cdot(1 - \frac{1}{1+2^{-\log(x)}\cdot r}) \text{ and } F_2 = 2\cdot(1-\frac{1}{1+e^{-f(x,y,r)}}),$$

We refer the reader to Appendix A.6 for the proof.

Corollary 4.7. *Le μ_i be the expected size of buckets in the ith level. r-ORAM with N elements overflows with a probability at most ω, if the size of each interior bucket z in the ith level equals $\mu_i + \ln \frac{N}{\omega}$ for $F_1 \leq \mu_i \leq F_2$.*

Proof (Sketch). By using the union bound, the probability that the system overflows equals $\omega = N\cdot e^{-\frac{\delta^2\cdot\mu}{2+\delta}}$. This is a quadratic equation in δ that has one valid root (non-negative) approximately equal to $\frac{1}{\mu_i}\cdot\ln\frac{N}{w}$, where μ_i is the expected value of ith level. The size of the bucket in this case equals $z = (1+\delta)\cdot\mu_i = \mu_i + \ln\frac{N}{\omega}$. □

For r-ORAM over Path ORAM [30] with a deterministic reverse-lexicographic eviction [7], Theorem 4.8 calculates the probability of stash overflow for a fixed bucket size. The goal of this theorem is to determine the optimal bucket size and therefore the stash size for a fixed overflow probability.

Theorem 4.8. *For buckets of size $z = 6$ and tree height $L = \lceil \log N \rceil$, the stash overflow probability computes to*

$$\Pr(\mathsf{st}(r\text{-}ORAM_L^6) > R) \leq 1.17 \cdot 0.88^R \cdot (1 - 0.54^N).$$

We refer the reader to Appendix A.7 for the proof.

Discussion: The probability is negligible in R (since 0.88 ¡1 and $1 - 0.54^N \xrightarrow{\infty} 1$). So, for a fixed overflow probability $\omega \ll 1$, we have to define the corresponding value of R by solving the equation $\omega = 1.17 \cdot 0.88^R \cdot (1 - 0.54^N)$. An r-ORAM stash with N elements overflows with probability at most $\omega \ll 1$, if the size of each bucket is 6, and the stash has size $R = \frac{1}{\ln 0.88} \cdot \ln \frac{\omega}{1.17 \cdot (1 - 0.54^N)}$. For large values of N, $R \in \Omega(\ln(\omega^{-1}))$.

We have made a number of approximations in our proof that slightly bias the choice of the bucket size and round the upper bound. We could improve our upper bound by a more accurate approximation of the number of subtrees in r-ORAM. Also, we assume the worst expected value for each bucket on all levels which is 4. Theorem 4.8 is valid for any bucket size $z \geq 6$.

5 Performance Analysis

We now analyze the behavior of r-ORAM when applied to different tree-based ORAMs. As a start, we compute the communication complexity of r-ORAM access, based on the average height, and estimate the monetary cost of access with r-ORAM on Amazon S3 cloud storage infrastructure. This first part is based on our r-ORAM theoretical results above. For *all previous* binary tree-based ORAMs, the communication complexity for a number of elements is always constant for fixed N. With previous ORAMs, you must always download an entire path. Following our theoretical estimates, we go on to present our r-ORAM implementation results and compare with Path ORAM [30]. We compare both the average height and the resulting communication improvements, and, finally, also evaluate the behavior of the stash.

5.1 Theoretical Results

Even if the worst-case complexity is in $O(\log N)$, the underlying constants gained with r-ORAM are significant. Table 1 compares between the height of a binary tree as with [7, 20, 28, 30] and the height of r-ORAM. Also, we compare r-ORAM on κ-ary trees, instead of binary ones, and we show that the recursive κ-ary tree r-ORAM gives better performances in terms of height access and communication cost.

Table 1: Tree height comparison

		Number of elements			
		2^{10}	2^{20}	2^{40}	2^{60}
Binary ORAM trees [7, 20, 28, 30]		10	20	40	60
Binary r-ORAM tree	Best case	5	8	16	23
	Average case	6	14	26	40
	Worst case	8	16	31	47
4-ary ORAM tree [7, 8, 30]		5	10	20	30
4-ary r-ORAM tree	Best case	3	6	11	16
	Average case	5	8	16	24
	Worst case	5	10	19	28

Table 1 has been generated using parameters from Theorems 2.1 and 3.1. This table compares only the complexity of accessing an element in the tree, i.e., going from the root to the leaf. It does not take the communication overhead of accessing the position map into account which we will deal with later. Moreover, Table 1 computes only the number and not the size of nodes accessed. The overall communication complexities will vary from one scheme to the other, and we detail costs below, too. Table 2 shows the gain (in %) of r-ORAM applied to binary trees ORAM, not distinguishing whether a scheme has constant or poly-log memory complexity.

As shown in Table 2, we improve on average 35% when r-ORAM is applied to any binary tree ORAM and 20% when applied to 4-ary ORAM trees. Compared to binary trees, the gain for κ-ary trees is smaller due to the reduction of the height of the tree. Trees are already "flat", so the benefit of recursion diminishes.

We present the *total* communication overhead comparison and a monetary comparison of communication overhead between tree-based ORAM constructions (with constant and poly-log client memory). For this, we use blocks with size 1 KByte.

Table 2: Tree-based ORAM gain

	Gain in %		
	Best case	Average case	Worst case
Binary ORAM trees [7, 20, 28, 30]	60	35	22.5
4-ary ORAM trees [7, 8, 30]	45	20	5

The number of entries (blocks) in every node varies depeding on the scheme. We apply the result of Theorem 4.4 and Theorem 4.6 to vary the size of the buckets accordingly. For the poly-logarythmic client memory, the size of the buckets of r-ORAM over Path ORAM are set to $z = 6$ based on Theorem 4.8. We take communication and storage overhead of the position map into account as well as the overhead induced by eviction (eviction rate equal to 2 for the constant client memory case).

Figure 5 depicts the communication cost per access, i.e., the number of bits transmitted between the client and the server for any read or write operation. The graph shows that r-ORAM applied to Path ORAM ($z = 6$) gives the smallest communication overhead. For example, with a dataset of 1 GByte, an access will cost 100 KByte in total. Moreover, if we set the number of entries z to 3 instead of 6, see [7], communication costs are divided by 2.

The storage overhead of tree-based ORAMs is still significant. Poly-log client memory ORAMs perform better, but still induce roughly a factor of 10. r-ORAM reduces this overhead down to a factor of 9.6, i.e., a reduction by 4%. For r-ORAM over Shi et al. [28] scheme, the saving is greater than 50% since we are reducing not only the height but also the size of the bucket.

Finally, we calculate the cost in US Dollar (USD) associated with every access, cf. Fig. 6. As we obtain smallest communication overhead by using r-ORAM on top of Path ORAM, one would naïvely expect this to be the cheapest construction. However, Amazon S3 pricing is based not only on communication in terms of transferred bits (Up to 10 TB/month, 0.090 USD per GBytes), but also on the number of HTTP operations performed (GETs and PUTs), 0.005 USD per 1,000 requests for PUT and 0.004 USD per 10,000 requests per month for GET. Surprisingly, the construction by Gentry et al. [8] with branching factor $\kappa = \log(N)$ is cheaper as it involves fewer HTTP operations compared to Path ORAM (however, in practice, the branching factor cannot be large since it will increase the size of the bucket).

5.2 Experimental Results

For a real-world comparison, we have implemented Path ORAM and r-ORAM including the position map in Python. Our source code is available for download [25]. Experiments were performed on a 64 bit laptop with 2.8 GHz CPU and 16 GByte RAM running Fedora Linux. For each graph, we have simulated 10^{15} random access operations. The standard deviation of the r-ORAM height (communication complexity) was low at 0.015. The relative standard deviation for the average height (communication complexity) for 93312$\approx 2^{16.5}$ elements equals to 0.125.

The experiments begin with an empty ORAM. We randomly insert the corresponding number of elements. This step represents the initialization phase. Afterwards, we run multiple random accesses to analyze the height behavior and the stash size for r-ORAM over Path ORAM.

Fig. 7 shows three curves: the height of binary tree ORAM (Path ORAM) from one hand and r-ORAM average and worst case height from the other hand. The height curves for r-ORAM are the result of 10^{15} accesses with a standard deviation of 0.015.

Our second comparison tackles communication including the recursion induced by the position map as well as the eviction per single access for different bucket sizes, see Fig. 8. Figures 12 and 13 show that r-ORAM improves communication even with larger block size, see Appendix B. The eviction in r-ORAM is performed at the same time the path is written back. Also, we consider both the upload/download phases. For example, with $N = 2^{14}$ and 4096 Bytes block size, the client has to download/upload 438 KByte with r-ORAM, instead of 640 KByte with Path ORAM, a ratio corresponding to the ratio of average heights, i.e., 31% of cost saving. Moreover, if we compare the curves associated to the minimum theoretical bounds for r-ORAM and Path ORAM, i.e., $z = 6$ and $z = 5$, the sav-

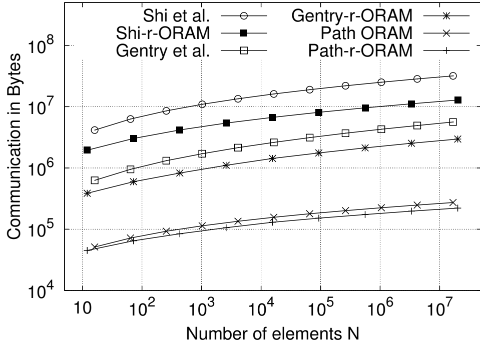


Figure 5: Communication per access

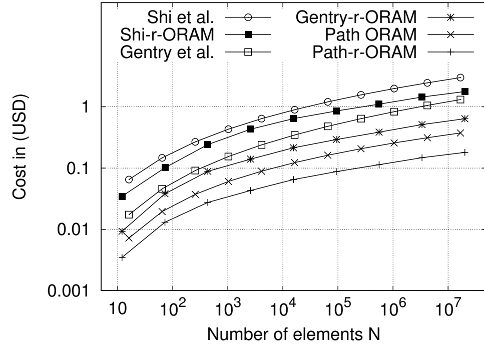


Figure 6: Communication cost per 100 accesses

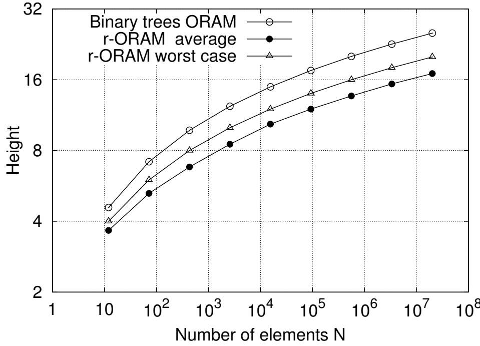


Figure 7: Average height comparison

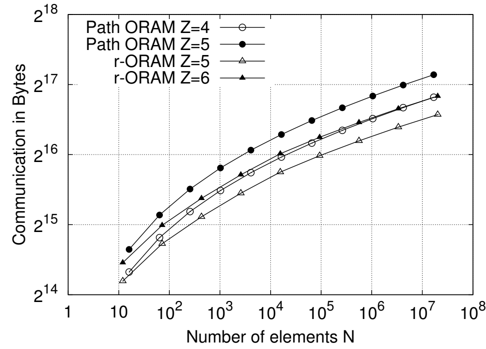


Figure 8: Communication per access, block 512 Bytes

ing in terms of communication complexity is 20%. These curves represent the average of 10^{15} random accesses.

Finally, we measure r-ORAM's stash size for a number of random accesses between 2^{10} and 2^{20}. The number of operations represent a security parameter for our scenario, the more operations we perform the more likely the stash size increases. The upper bound of Th. 4.8 depends of the number of elements N, however for $N > 2$ the stash will have the same size independently of N because $1 - 0.54^N \approx 1$ for larger N. Thus, the stash in r-ORAM over Path ORAM has a logarithmic behavior in function of the security parameter, see Theorem 4.8.

Our experimental results confirm the upper bound given by Theorem 4.8, namely $R = \frac{1}{\ln 0.88} \cdot \ln \frac{\omega}{1.17 \cdot (1-0.54^N)} \approx \frac{1}{\ln 0.88} \cdot \ln \frac{\omega}{1.17}$. For example, for a probability of overflow equal to $\omega = 2^{-20}$,the security parameter here equals 20, the theoretical stash size R equals $\sim$110 blocks for any $N > 10$. In Fig. 9, you can see that, for bucket size $z = 6$ (See Appendix B for larger bucket size), we have exactly a logarithmic behavior as shown in the theorem. This figure shows the stash behavior based on the maximum, minimum, and median values. For a confidence level of 95%, the margin error is around 1.25. For 2^{20} operations, the maximum stash value equals 40 which is smaller than 110, the theoretical value, which is not surprising since some loose bounds have been used in the proof. For $z = 5$, see Appendix B. The stash seems to increase logarithmically with the number of operations. However, theoretically the stash size behavior is not bounded. The graphs are logarithmic in the number of operations. In Fig. 10, we show the average behavior of the stash size, to also indicate its logarithmic behavior.

Compared to Path ORAM with a similar bucket size, r-ORAM's stash requires up to 10 blocks

more. This will not have any repercussion on the communication complexity. One might argue that the overall client memory size $\mathcal{M}$ has to be larger. However, the client memory size is defined as the stash *plus* the downloaded path during the operation such that $\mathcal{M} = R + \mathcal{P}$ where R is the stash size and $\mathcal{P}$ the number of blocks downloaded for a given path p. We have $\mathcal{P} = z \cdot |p|$ blocks where $|p| = \log N$ for Path ORAM and $|p| \approx 0.78 \cdot \log N$ for r-ORAM (worst-case). For a number of elements $N = 2^{20}$ and a bucket size $z = 5$, Path ORAM has to have 20 more blocks than r-ORAM and this will increase for greater number of elements.

6 Related Work

ORAM, first introduced by Goldreich and Ostrovsky [10], recently received a revived interest from the research community [1, 4, 5, 7–13, 18, 20, 24, 26, 28, 30, 32, 33]. The current state of the art on ORAM can be divided into two main categories. The first one comprises schemes where a client is restricted to constant local memory, while the second allows the client (sub-linear) local memory.

Constant client memory: Constant client-side memory schemes are very useful for scenarios with very limited memory devices such as embedded devices. Recent works have been able to enhance amortized and worst-case communication complexity [11, 12, 18, 20, 24, 26, 28]. Goodrich and Mitzenmacher [11] and Pinkas and Reinman [26] introduce schemes with a poly-logarithmic *amortized* cost in $O(\log^2(N))$. However, the worst-case cost remains linear. Goodrich et al. [12] present a better worst-case communication overhead, $O(\sqrt{N} \cdot \log^2(N))$.

All schemes prior to the one by Shi et al. [28] differentiate between worst-case and amortized-case overhead. The worst-case scenario in these ORAM constructions occurs when a *reshuffling* is performed. Shi et al. [28] present a tree-based ORAM, where the nodes of the tree are small bucket ORAMs, see also [10, 23]. Accessing an element in this structure implies accessing a path of the tree. After each access, a partial reshuffling, confined to only the path accessed in the tree, is performed. The worst-case and amortized case overhead achieved with such construction are both equal and poly-logarithmic, i.e., $O(\log^3(N))$.

Mayberry et al. [20] improve the complexity of the tree-based ORAM by Shi et al. [28]. Instead of using traditional ORAM bucket nodes in the tree, a PIR [19] is used to retrieve a data element from a specific node. Mayberry et al. [20] show that therewith the worst-case communication complexity equals $O(\log^2(N))$. Note that this complexity can be enhanced by using a κ-ary tree instead of a binary tree.

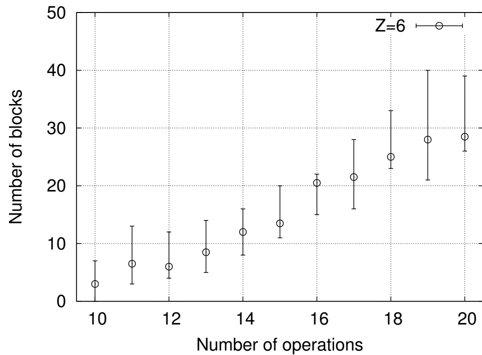
Kushilevitz et al. [18] present a hierarchical solution that enhances the asymptotic communication complexity defined in previous works with a worst case equal to $O(\frac{\log^2(N)}{\log(\log(N))})$. For large block sizes, the scheme is in practice significantly less efficient compared to, e.g., [20, 28].

Devadas et al. [6] present a tree-based construction, Onion ORAM, with constant communication complexity. However, their block size is large with $B = \Omega(\log^6(N))$. Recently, Moataz et al. [22] improved Onion ORAM by reducing the server computation as well as the block size by a factor of $log^2 N$. Note that our recursive tree structure can be applied to both of these ORAMs, too, further reducing communication costs.

Sub-linear client memory Past research with $O(\sqrt{N})$ client-side memory [32, 33] has sub-linear amortized communication complexity, but linear worst-case complexity. Boneh et al. [1] improve the worst-case to be in $O(\sqrt{N})$, however still with $O(\sqrt{N})$ client-side memory. Stefanov et al. [29] show how to reduce the amortized cost to be poly-logarithmic in $O(\log^2(N))$, but still with a large $O(\sqrt{N})$ client memory.

Gentry et al. [8] enhance previous tree-based ORAM work by modifying the structure of the tree. Instead of a binary tree, a multi-dimensional tree with a branching factor κ is used. If the number of data elements stored in every node and the branching factor are equal to $O(\log N)$, the worst-case communication overhead is in $\frac{\log^3(N)}{\log(\log(N))}$. Gentry et al. [8] also introduce a reverse lexicographic eviction that is used in recent poly-logarithmic client memory schemes. The poly-logarithmic client memory in $O(\log^2 N)$ is due to its eviction algorithm, where the client has to memorize elements in order to percolate them towards the leaves.

Stefanov et al. [30] present Path ORAM, another tree-based ORAM construction, requiring a client-side memory *stash* of size $O(\log(N))$. This results in $O(\log^2(N))$ communication complexity. Fletcher



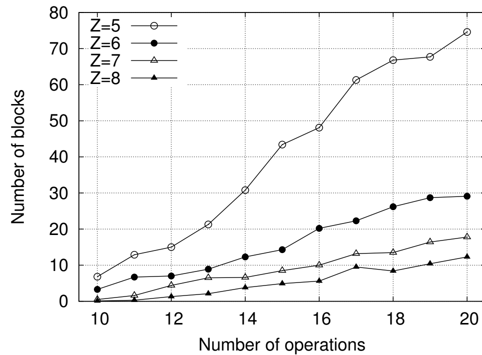


Figure 9: Stash size, $z = 6$, number of operations in $\log_2$

Figure 10: Average stash size, operations in $\log_2$

et al. [7] further optimized Path ORAM, reducing the communication cost by $6 \sim 7\%$. Similarly, Ren et al. [27] improve upon Path ORAM and SSS ORAM [29], resulting in better bandwidth parameters but still in $O(\log N)$ and with $O(\sqrt{N})$ for large block sizes. Also for this ORAM, our framework can enhance communication overhead even more.

7 Conclusion

r-ORAM is a general technique for tree-based ORAM costs optimization. r-ORAM improves both communication cost as well as storage cost. We formally show that r-ORAM preserves the same overflow probability as related work. r-ORAM is general and can be applied to any existing as well as future derivations of tree-based ORAMs. For any binary tree-based ORAM, the average cost is reduced by 35%, and storage cost is reduced by 4 to 20%. As future work, we plan to investigate the dynamics of r-ORAM, i.e., instead of considering constant height for outer and inner trees, we aim at dynamically varying the height, which has the potential of further cost reductions.

Acknowledgement. This work was partially supported by NSF grant 1218197

References

[1] D. Boneh, D. Mazières, and R.A. Popa. Remote oblivious storage: Making oblivious RAM practical, 2011. `http://dspace.mit.edu/bitstream/handle/1721.1/62006/MIT-CSAIL-TR-2011-018.pdf`.

[2] K. Brown. Balls in bins with limited capacity, 2014. `http://www.mathpages.com/home/kmath337.htm`.

[3] G. Casella and R.L. Berger. *Statistical inference*. Duxbury advanced series in statistics and decision sciences. Thomson Learning, 2002. ISBN 9780534243128.

[4] K.-M. Chung and R. Pass. A Simple ORAM. *IACR Cryptology ePrint Archive*, 2013:243, 2013.

[5] I. Damgård, S. Meldgaard, and J.B. Nielsen. Perfectly Secure Oblivious RAM without Random Oracles. In *Proceedings of Theory of Cryptography Conference –TCC*, pages 144–163, Providence, USA, March 2011.

[6] Srinivas Devadas, Marten van Dijk, Christopher W. Fletcher, and Ling Ren. Onion ORAM: A constant bandwidth and constant client storage ORAM (without FHE or SWHE). *IACR Cryptology ePrint Archive*, 2015:5, 2015.

[7] Christopher W. Fletcher, Ling Ren, Albert Kwon, Marten van Dijk, Emil Stefanov, and Srinivas Devadas. RAW Path ORAM: A Low-Latency, Low-Area Hardware ORAM Controller with Integrity Verification. *IACR Cryptology ePrint Archive*, 2014:431, 2014.

[8] Craig Gentry, Kenny A. Goldman, Shai Halevi, Charanjit S. Jutla, Mariana Raykova, and Daniel Wichs. Optimizing ORAM and Using It Efficiently for Secure Computation. In *Proceedings of Privacy Enhancing Technologies*, pages 1–18, 2013.

[9] O. Goldreich. Towards a Theory of Software Protection and Simulation by Oblivious RAMs. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing – STOC*, pages 182–194, New York, USA, 1987.

[10] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *J. ACM*, 43(3):431–473, May 1996. ISSN 0004-5411. doi: 10.1145/233551.233553. URL `http://doi.acm.org/10.1145/233551.233553`.

[11] M.T. Goodrich and M. Mitzenmacher. Privacy-preserving access of outsourced data via oblivious ram simulation. In *Proceedings of Automata, Languages and Programming –ICALP*, pages 576–587, Zurick, Switzerland, 2011.

[12] M.T. Goodrich, M. Mitzenmacher, Olga Ohrimenko, and Roberto Tamassia. Oblivious ram simulation with efficient worst-case access overhead. In *Proceedings of the 3rd ACM Cloud Computing Security Workshop – CCSW*, pages 95–100, Chicago, USA, 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 Symposium on Discrete Algorithms –SODA*, pages 157–167, Kyoto, Japan, 2012.

[14] Google. A new approach to China, 2010. `http://googleblog.blogspot.com/2010/01/new-approach-to-china.html`.

[15] D. Gross. 50 million compromised in Evernote hack, 2013. `http://www.cnn.com/2013/03/04/tech/web/evernote-hacked/`.

[16] J Hsu and P Burke. Behavior of tandem buffers with geometric input and markovian output. *Communications, IEEE Transactions on*, 24(3):358–361, 1976.

[17] L. Kleinrock. *Theory, Volume 1, Queueing Systems*. Wiley-Interscience, 1975. ISBN 0471491101.

[18] 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 Symposium on Discrete Algorithms – SODA*, pages 143–156, Kyoto, Japan, 2012.

[19] Eyal Kushilevitz and Rafail Ostrovsky. Replication is not needed: Single database, computationally-private information retrieval. In *Proceedings of Foundations of Computer Science –FOCS*, pages 364–373, Miami Beach, USA, 1997.

[20] T. Mayberry, E.-O. Blass, and A.H. Chan. Path-pir: Lower worst-case bounds by combining oram and pir. In *Proceedings of the Network and Distributed System Security Symposium*, San Diego, USA, 2014.

[21] T. Moataz, T. Mayberry, E.-O. Blass, and A.H. Chan. Resizable Tree-Based Oblivious RAM. In *Proceedings of Financial Cryptography and Data Security*, Puerto Rico, USA, 2015.

[22] Tarik Moataz, Travis Mayberry, and Erik-Oliver Blass. Constant Communication ORAM with Small Blocksize. In *ACM Conference on Computer and Communications Security*, Denver, CO, USA, 2015.

[23] R. Ostrovsky. Efficient computation on oblivious rams. In *Proceedings of the Symposium on Theory of Computing –STOC*, pages 514–523, Baltimore, USA, 1990.

[24] R. Ostrovsky and V. Shoup. Private information storage (extended abstract). In *Proceedings of the Symposium on Theory of Computing –STOC*, pages 294–303, El Paso, USA, 1997.

[25] PASMAC. r-ORAM source code, 2015. `http://pasmac.ccs.neu.edu/resources/r-ORAM.zip`.

[26] B. Pinkas and T. Reinman. Oblivious ram revisited. In *Advances in Cryptology – CRYPTO*, pages 502–519, Santa Barbara, USA, 2010.

[27] Ling Ren, Christopher W. Fletcher, Albert Kwon, Emil Stefanov, Elaine Shi, Marten van Dijk, and Srinivas Devadas. Ring ORAM: closing the gap between small and large client storage oblivious RAM. *IACR Cryptology ePrint Archive*, 2014:997, 2014.

[28] E. Shi, T.-H.H. Chan, E. Stefanov, and M. Li. Oblivious RAM with $O(\log^3(N))$ Worst-Case Cost. In *Proceedings of Advances in Cryptology – ASIACRYPT*, pages 197–214, Seoul, South Korea, 2011. ISBN 978-3-642-25384-3.

[29] E. Stefanov, E. Shi, and D.X. Song. Towards practical oblivious ram. In *Proceedings of the Network and Distributed System Security Symposium*, San Diego, USA, 2012. The Internet Society.

[30] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher W. Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: an extremely simple oblivious RAM protocol. In *ACM Conference on Computer and Communications Security*, pages 299–310, 2013.

[31] Techcrunch. Google Confirms That It Fired Engineer For Breaking Internal Privacy Policies, 2010. http://techcrunch.com/2010/09/14/google-engineer-spying-fired/.

[32] P. Williams and R. Sion. Usable pir. In *Proceedings of the Network and Distributed System Security Symposium*, San Diego, USA, 2008.

[33] P. Williams, R. Sion, and B. Carbunar. Building castles out of mud: practical access pattern privacy and correctness on untrusted storage. In *ACM Conference on Computer and Communications Security*, pages 139–148, Alexandra, USA, 2008.

A Proofs

A.1 Proof of Theoram 2.1

Proof. Function $\mathcal{C}$ depends on three variables that we can reduce to two by substituting Eq. (2) into Eq. (3). From Eq. (2), we have $\log x = \log(N) - r - r\cdot\log(y-1)$. The worst-case cost then computes to

$$\mathcal{C}(r, y) = \log(N) - r + r \cdot \log(\frac{y}{y-1}). \quad (8)$$

By fixing $r > 0$, the worst-case cost is a non-increasing function in y, since $y \mapsto \log(\frac{y}{y-1})$ is a non-increasing function for $y > 1$. Thus, for any non-negative r, the minimum value of the worst cost is smaller for larger values of y.

Also, with $x \geq 2$, the number of the leaves of inner trees y is upper bounded: $N \geq 2\cdot(2y-2)^r \Rightarrow y \leq \frac{1}{2}\cdot(\frac{N}{2})^{\frac{1}{r}} + 1$.

For small x, we therewith get a larger upper bound for y. Therefore, we have to fix x to its minimum value which equals 2. This could not be inferred from Eq. 3 while not decreasing the number of variables of the linear system. The optimum number of leaves for the inner trees then equals $y = \frac{1}{2}\cdot(\frac{N}{2})^{\frac{1}{r}} + 1$. Putting these values back in Eq. (8), results in $\mathcal{C}$ depending on only one variable r, the recursion factor:

$$\mathcal{C}(r) = 1 + r \cdot \log(\frac{1}{2}\cdot(\frac{N}{2})^{\frac{1}{r}} + 1) \quad (9)$$

Finally, we derive the minimum of the worst-case cost by computing the first derivative of the convex function $\mathcal{C}(r)$. The derivative is $\frac{d\mathcal{C}}{dr}(r) = \log(\frac{1}{2}\cdot(\frac{N}{2})^{\frac{1}{r}} + 1) - \frac{\ln(\frac{N}{2})\cdot(\frac{N}{2})^{\frac{1}{r}}}{2r\cdot(\frac{1}{2}\cdot(\frac{N}{2})^{\frac{1}{r}}+1)}$.

We achieve $\frac{d\mathcal{C}}{dr}(r) \approx 0$ for $r' \approx \log((\frac{N}{2})^{\frac{1}{2.7}})$. Since $\mathcal{C}(r)$ is convex, the value of r' is the minimum for any $r \leq \log(N) - 1$. Replacing r' in equations (4) and (9) gives the worst-case and best-case costs of the theorem, therefore completing the proof. □

Careful readers will notice that we have bounded x to be at least equal to 2 in our theorem's proof. If we consider that $x = 1$, we do not therefore have any leaf tree at the end. Thus, there are some nodes in the last recursion that will behave as leaves and interior buckets at the same time. This will have some critical issues in term of security proofs. Considering a node as interior and leaf node at the same time will basically double the size (because in the analysis we have to consider the disjunction of both events). Fixing $x = 1$, seems a good idea that simplifies greatly the analysis, however it will not help in optimizing the communication overhead. In fact, the size, in bits, to download a path will be equal

to the same path with $x = 2$ with two times more elements.

A.2 Proof of Theorem 3.1

(Sketch). The first step in the proof is to represent the number of leaves x as a function of N, y, r, and κ the branching factor. That is, we reduce the number of variables in our optimization problem by one. Taking the logarithm of Eq. (7) leads to $\log_\kappa(x) = \log_\kappa(\frac{\kappa-1}{\kappa}N) - r \cdot \log_\kappa(y-1)$. Since our first goal is the minimization of the worst-case cost, we substitute $\log_\kappa(x)$ in the worst-case cost Eq. (3) by the value computed in the above equation and minimize the new expression. Note that the logarithm is base κ instead of 2 in the worst-case cost formula.

For simplicity, we consider the branching factor as a (given) constant, as it has an impact on the overflow probability. So, we assume a fixed branching factor matching a given bucket size. Finally, we follow the same steps as the proof of Theorem 2.1 to find the optimal recursive factor r, the number of leaf tree leaves x, and the number of inner/outer tree leaves y. □

A.3 Proof of Lemma 4.2

Proof. First, we determine the number of interior nodes for $i > r$. In the same spirit as the proof of Theorem 2.2, we denote by A_j an array of $j \in [r]$ positions that has all positions initialized to zero. A_j represents the number of possible paths to a given level. The difference between counting the number of leaves and the number of interior nodes consists of the fact that an interior node may exist in any level without going through all recursions, i.e., it may happen that we reach a level without going through the last level of recursions. This means that elements of the array are equal to zero.

Counting of interior nodes boils down to divide Eq. (5) of Theorem 2.2 in r sub-equations, where each will count the number of ways to reach a specific level while all the positions of the array are still equal to 1. Therefore, the set of solutions of the following sub-equations has an empty set intersection.

$$\begin{aligned} A_1[1] - 1 &= i - 1 - \log(x), \\ (A_2[1]-1) + (A_2[2]-1) &= i - 2 - \log(x), \\ &\cdots \\ (A_r[1]-1) + \cdots + (A_r[r]-1) &= i - r - \log(x), \end{aligned}$$

where, for each $j \in [r]$, we have $1 \leq A_j[i] \leq \log(y)$.

Discussion: To have an intuition about these partitions, consider an example where $r = 4$ and $y = 16$. We have 4 sub-equations, where each represents the possible ways to reach an interior node in, e.g., the 4th level. The first array has only one position that can take values from 1 to 4. The first sub-equation will count the number of ways to get to an interior node at level 4 under the constraint that we have to stay in one recursion. In this case, the array can have only one value which is 4. For the second equation, we can have different combinations such as $(2,2)$, $(3,1)$, etc., but we do not have $(4,0)$, because it is already accounted for in the first sub-equation. We follow the same reasoning for the other sub-equations.

So, $I(i) = \mathcal{S}_1 + \cdots + \mathcal{S}_r$, the total number of solutions of the sub-equations. Also, we have $S_r \geq S_j$ for any $j \in [r-1]$, that is, $I(i) \leq r \cdot S_r$. From Theorem 2.2, we know that the number of solutions for the last equation S_r equals $2^{i-\log x} \cdot \mathcal{K}(i)$. Therefore, with the result of Theorem 2.2, we can conclude that $I(i) \leq 2^{i-\log(x)} \cdot r \cdot \mathcal{K}(i)$.

Also from Th. 2.2, the number of leaves $\mathcal{N}(i) = 2^i \cdot \mathcal{K}(i)$. This leads to our first inequality $\frac{I(i)}{\mathcal{N}(i)} \leq 2^{-\log(x)} \cdot r$.

For our second inequality, notice that for any interior node of any level $i > r$, $I(i) \geq \frac{\mathcal{N}(i+1)}{2}$. This follows from the property that the ancestors of leaves in the $(i+1)^{\text{th}}$ level are interior nodes in the upper level. Using equality $\mathcal{N}(i) = 2^i \cdot \mathcal{K}(i)$,

$$\begin{aligned} \frac{I(i)}{\mathcal{N}(i)} &\geq \frac{\mathcal{N}(i+1)}{2\mathcal{N}(i)} \\ &= \frac{\mathcal{K}(i+1)}{\mathcal{K}(i)}. \end{aligned}$$

We have previously shown that $\mathcal{K}$ can be approximated by a normal distribution, cf. Eq. (6). Using this approximation, we obtain $\frac{I(i)}{\mathcal{N}(i)} \geq e^{-\frac{1+2i-2r-2\log(x)-c}{2s^2}}$.

Finally, since $c = r(\log(y) - 1)$, we have for $s > 0$ $\frac{I(i)}{\mathcal{N}(i)} \geq e^{-\frac{1+r\log(y)-r-2\log(x)}{2s^2}}$. This concludes our proof. □

A.4 Proof of Lemma 4.3

Proof. This result follows from two observations. First, the total number of interior nodes for the ith level is always larger than the total number of nodes in the $(i+1)^{\text{th}}$ level divided by 4. The second observation is that the total number of interior nodes for the ith level is always smaller than the total number of nodes in $(i+1)^{\text{th}}$ divided by 2. Consequently, $\frac{L(i+1)}{4} \leq I(i) \leq \frac{L(i+1)}{2}$.

The second inequality follows from r-ORAM's structure where every interior node v has at least 2 children and at most 4 children. The recursion as previously represented in a 2-dimensional plane where an interior node in the outer or inner tree has 4 children, and every leaf node has exactly 2 children. So, every level has at least twice the number of interior nodes of the previous level.

We bound μ by algebraic transformations:

$$\frac{L(i+1)}{4} + \mathcal{N}(i) \leq L(i) \leq \frac{L(i+1)}{2} + \mathcal{N}(i)$$

$$\frac{\mu}{4} + \frac{\mathcal{N}(i)}{L(i)} \leq 1 \leq \frac{\mu}{2} + \frac{\mathcal{N}(i)}{L(i)}.$$

Finally, $2 \cdot (1 - \frac{\mathcal{N}(i)}{L(i)}) \leq \mu \leq 4 \cdot (1 - \frac{\mathcal{N}(i)}{L(i)})$. □

A.5 Proof of Theorem 4.4

Proof. The buckets of r-ORAM can be considered as queues [16]. Every bucket at the ith level has its service rate η_i and its arrival rate λ_i. The probability that the bucket contains z elements is given by: $p(z) = (1 - \rho_i) \cdot \rho_i^z$, where $\rho_i = \frac{\lambda_i}{\eta_i}$. This is a result of M/M/1 queues [17]. The probability that the bucket will have strictly less than z elements equals $\sum_{i=0}^{k-1} p(i) = 1 - \rho_i^z$. The probability to overflow (to have more than z elements) equals ρ_i^z. In the following, it suffices to compute ρ_i for every level in our r-ORAM structure.

Consider eviction rates that are powers of 2. Then, for $i \leq \lceil \log_4(\chi) \rceil$, we have $\eta_i = 1$ and $\lambda_i \leq \frac{1}{2^i}$ (because for level 1 and deeper, buckets may have up to 4 children).

For $i > \lceil \log_4(\chi) \rceil$, the chance that a given bucket will be evicted is equal to $\eta_i = \frac{\chi}{I(i)}$, where $I(i)$ is the number of interior nodes in the ith level.

$\lambda_i = \frac{I(i)}{L(i+1)} \cdot \Pr(\text{parent gets selected}) \cdot \Pr(\text{parent is not empty})$, such that $\Pr(\text{parent gets selected}) = \eta_{i-1}$ and $\Pr(\text{parent is not empty}) = 1 - p_{i-1}(0) = \rho_{i-1}$. The ratio $\frac{I(i)}{L(i+1)}$ denotes the probability for a real element to be evicted, in the case of a binary tree the ratio is equal to $\frac{1}{2}$. Then, we have $\lambda_i = \frac{I(i)}{L(i+1)} \cdot \lambda_{i-1}$. By induction, the arrival rate equals $\lambda_i = \frac{1}{L(i+1} \cdot \frac{I(i) \cdot I(i-1) \cdots I(s+1)}{L(i) \cdot L(i-1) \cdots L(s+1)} \cdot I(s) \cdot \lambda_s$, where $s = \lceil \log_4(\chi) \rceil$. With $\lambda_s \leq \frac{1}{2^s}$ and $I(s) \leq 4^s$ (because we can have at most 4 children for every interior node), this equation can be upper-bounded such that:

$$\lambda_i \leq \frac{2^s}{L(i+1)} \cdot \frac{1}{1 + \frac{\mathcal{N}(i)}{I(i)}} \cdots \frac{1}{1 + \frac{\mathcal{N}(s+1)}{I(s+1)}}. \quad (10)$$

We need to simplify the above inequality. First, notice that for every $s < i \leq r$

$$\frac{1}{1 + \frac{\mathcal{N}(i)}{I(i)}} = 1, \quad (11)$$

because $\mathcal{N}(i) = 0$ (there is no leaf node for $i \leq r$). For $i > r$, using the result of Lemma 4.2.

$$\frac{1}{1 + \frac{\mathcal{N}(i)}{I(i)}} \leq \frac{1}{1 + \frac{x}{r}}, \quad (12)$$

where x is the number of leaves. For buckets at level $i > r$, we plug the result of equations 11 and 12 in 10 and we divide by the service rate η_i such that $\rho_i \leq \frac{I(i)}{L(i+1)} \cdot (\frac{1}{1+\frac{x}{r}})^{i-r} \cdot \frac{2^s}{\chi}$.

From Lemma 4.3, we have shown that $\frac{I(i)}{L(i+1)} < \frac{1}{2}$, because there are at least twice more nodes than interior nodes in the upper level (they may be leaves or interior nodes). Then $\rho_i \leq (\frac{1}{1+\frac{x}{r}})^{i-r} \cdot \frac{2^s}{2\chi}$. In this case ρ_i is upper-bounded by $\theta_i = (\frac{1}{1+\frac{x}{r}})^{i-r} \cdot \frac{2^s}{2\chi}$, and the overflow probability is then equal to θ_i^z.

For $i \leq r$, there are no leaves (i.e. $\mathcal{N}(i) = 0$), and the arrival rate is always bounded from Eq. 10 such that $\lambda_i \leq \frac{2^s}{L(i+1)}$.

Consequently, dividing by η_i and using the result of Lemma 4.3 $\frac{I(i)}{L(i+1)} < \frac{1}{2}$ we get $\rho_i \leq \frac{2^s}{2\chi}$. Considering $\theta_i = \frac{2^s}{2\chi}$ for $i \leq r$ concludes our proof. □

A.6 Proof of Theorem 4.6

Proof. Let us fix an interior node v in r-ORAM belonging to the ith level. We are interested in the behavior of the node's load after a number of operations including eviction and adding operations. Let $L(i)$ denote the number of nodes residing in the ith level of the r-ORAM tree (these include the interior and the leaf nodes). Since the eviction is reverse-lexicographic and deterministic, we are sure that any element inserted before the time interval $\mathcal{T} = [t - L(i+1) + 1, \cdots, t]$ has been evicted from the ith level. Therefore, if we denote the number of elements residing in the node v, $S_t(v)$, we are sure that $S_t(v) = 0$ just a step before the interval $\mathcal{T}$. Consequently, it remains to determine the load of the interior node v for all the steps of the interval $\mathcal{T}$, i.e., the load of the node v in the (possible) presence of at most $L(i+1)$ elements in the ith level or above. Let us associate for every element j in $\mathcal{T}$ a random indicator variable χ_j which is equal to 1 if the element was assigned a path going through the interior node v. All elements in $\mathcal{T}$ are i.i.d. and their assignment probability is $\Pr(\chi_j = 1) = \frac{1}{L(i)}$. We have also $S_t(v) \leq \sum_{j \in [I(i+1)]} \chi_j$, which follows from the fact that all elements inserted in the interval $\mathcal{T}$ may *at most* all of them be assigned paths that go through v. In order to apply Chernoff's bound, we calculate the expected value of the sum of the indicator variables

$$E(\sum_{j \in [L(i+1)]} \chi_j) = \mu = \frac{L(i+1)}{L(i)}.$$

The exact value cannot be determined without computing the number of nodes existing in the ith level. What we can do is computing a tight bound of the expected value and then apply the Chernoff bound. Note that this expected value will be different from one level to the other.

Lemma 4.3 gives a bound on the expected value. This bound involves a relation between the leaf node and the interior nodes of the given level that we have computed in Lemma 4.2. For $i \leq r$, from Lemma 4.3, we know that $2 \leq \mu \leq 4$. For $i > r$, plug the first lemma in the second:

$$\underbrace{2 \cdot (1 - \frac{1}{1 + e^{-f(x,y,r)}})}_{F_1} \leq \mu \leq \underbrace{4 \cdot (1 - \frac{1}{1 + 2^{-\log(x)} \cdot r})}_{F_2}$$

Now, wrapping up with Chernoff's bound, for any $\delta > 0$ and for both cases $\Pr(S_t(v) \geq (1+\delta) \cdot \mu) \leq \Pr(\sum_{j \in [L(i+1)]} \chi_j \geq (1+\delta) \cdot \mu) \leq e^{-\frac{\delta^2 \cdot \mu}{2+\delta}}$. This concludes our proof. □

To get an idea about the values of F_1 and F_2, we calculate them for $N = 2^{32}$: $F_1 = \frac{2}{5}$ and $F_2 = 3.42$. The theorem above represents a general bound to understand the overflow probability behavior. Since the expected value μ varies depending on the level, buckets sizes vary on every level. Consequently, fixing the expected value for every level results in much better bounds. For example, if for level i, $\mu = 1$, then the the probability of overflow with a bucket size equal to $64 = 1 + \delta$ is at most 2^{-88}, while for $\mu = 4$, the probability of overflow with the same bucket size is equal to 2^{-82}.

A.7 Proof of Theorem 4.8

Proof. To prove this theorem, we borrow two lemmas from Stefanov et al. [30], namely their lemmas 1 and 2. We begin by giving a short overview over these two lemmas. For details and proofs, we refer to [30]. The first lemma underlines that the state of r-ORAM$_L^z$ is equal to the state of r-ORAM$_L^\infty$ after post-processing with a greedy algorithm G. r-ORAM$_L^\infty$ is r-ORAM$_L^z$ with an infinite number of entries in each block. For r-ORAM$_L^\infty$, we do not need a stash, since buckets can hold an infinite number of blocks. Algorithm G process r-ORAM$_L^\infty$ to have the same bucket construction as in regular r-ORAM$_L^z$ with deterministic reverse lexicographic eviction. Let $X(T)$ be the number of real blocks in some subtree T and $\eta(T)$ the number of nodes in subtree T. Now, Lemma 2 by Stefanov et al. [30] states that $\mathsf{st}(r\text{-ORAM}_L^z) > R$, *iff* there exists a subtree T such that $X(T) > \eta(T) \cdot z + R$. Combining the two lemmas results in

$$\Pr(\mathsf{st}(r\text{-ORAM}_L^z) > R) = \Pr(\mathsf{st}(G(r\text{-ORAM}_L^\infty)) > R) \quad (13)$$

$$\leq \sum_{T \in r\text{-ORAM}_L^\infty} \Pr(X(T) > \eta(T) \cdot z + R)$$

$$< \sum_{i=1}^{N} 4^i \max_{\{T | \eta(T) = i\}} \Pr(X(T) > i \cdot z + R).$$

The second inequality follows from the fact that the number of subtrees in a full binary tree of N

elements is upper bounder by the Catalan number $C_i < 4^i$. The upper bound in Th. 4.8 might be tighter if we consider that r-ORAM contains fewer subtrees than the ones in a full binary tree.

We now bound $\max_{\{T|\eta(T)=i\}} \Pr(X(T) > i \cdot z + R)$.

First, to find an upper bound for Eq. 13, we compute the expected value of $X(T)$ for subtree T of r-ORAM$_z^\infty$. Note that $E(X(T)) = \sum_{i=1}^{\eta(T)} E(|B_i|)$, where $|B_i|$ is the size of a bucket B_i in T. In r-ORAM$_L^\infty$, the expected value of buckets changes between levels, following a well-defined distribution of interior nodes. For ease of exposition, we now assume that all buckets have the worst bucket load. To show this, we have to take into account two cases.

(1) If a bucket is a leaf bucket, the load is binomially distributed, such that $E(|B_i|) = N \cdot 2^L = 1$.

(2) For an interior bucket on level i, we have shown in Th. 4.6 that $E(|B_i|) = \mu$ and $F_1 \leq \mu \leq F_2$ (F_2 is equal to its maximal value 4).

For both cases, we can bound the expected value of the bucket's load: $\max\{1, F_1\} \leq E(B_i) \leq 4$. That is, for any bucket in T, we obtain $\eta(T) \cdot \max\{1, F_1\} \leq E(X(T)) \leq 4 \cdot \eta(T)$.

Let $\Psi = E((X(T))$, $\eta(T) = n$, and $\xi = \frac{n \cdot z + R - \Psi}{\Psi}$. Applying Chernoff's bound to $X(T)$, we get

$$\Pr(X(T) > n \cdot z + R) = \Pr(X(T) > (1+\xi)\Psi) \leq e^{\frac{-\xi^2}{2+\xi} \cdot \Psi}.$$

With some algebraic computations, it is easy to see that $\frac{(n \cdot (z-4)+R)^2}{\Psi} \leq \xi^2 \cdot \Psi$ and $(\frac{n \cdot (z-4+8)+R}{\Psi})^{-1} \leq (\frac{n \cdot (z-4)+R+2\Psi}{\Psi})^{-1} \leq (2+\xi)^{-1}$.

For $z > 5$, we have

$$\frac{1}{8}(n \cdot (z-4) + R) \leq \frac{\xi^2 \cdot \Psi}{2+\xi} \tag{14}$$

$$e^{\frac{-\xi^2}{2+\xi} \cdot \Psi} \leq 0.88^R \cdot e^{-n \cdot (z-4)}.$$

Combining Eq. 14 with Eq. 13 results in

$$\Pr(\mathsf{st}(r\text{-ORAM}_{\mathsf{L}}^{\mathsf{6}}) > R) < 0.88^R \cdot \sum_{i=1}^{N} e^{-i(6-4-\ln 4)}$$

$$\approx 1.17 \cdot 0.88^R \cdot (1 - 0.54^N)$$

□

B Experiments

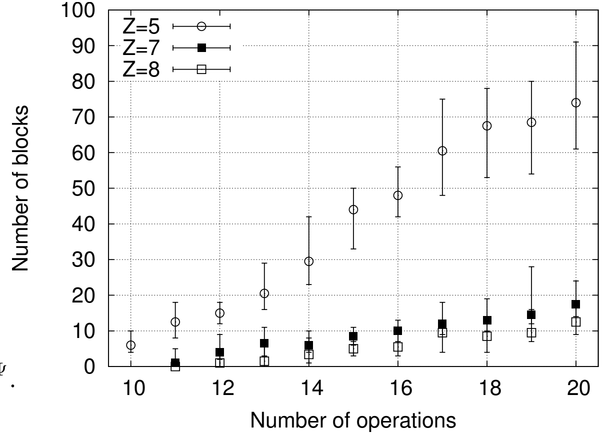


Figure 11: Stash size for $z = 5, 7$ and 8 with number of operations in $\log_2$

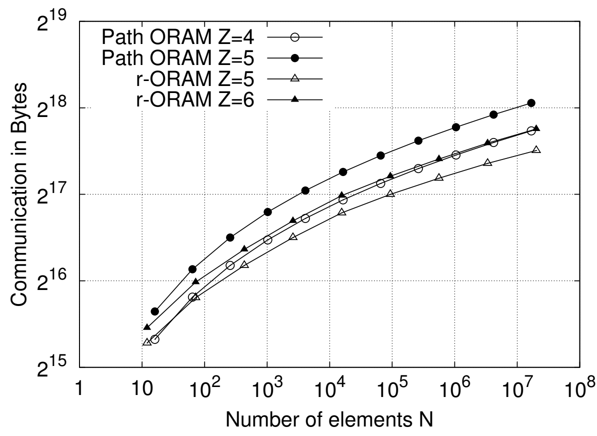


Figure 12: Communication per access, block 1024 Bytes

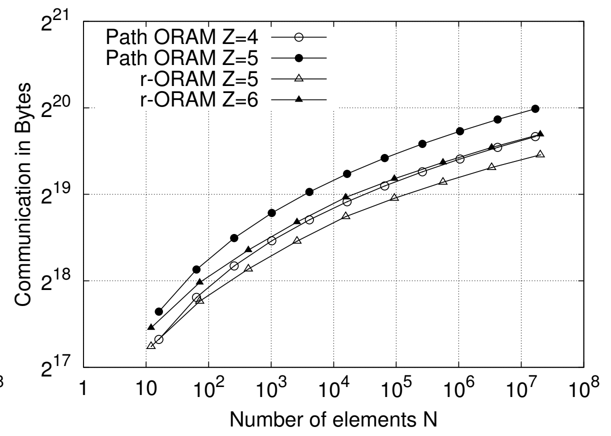


Figure 13: Communication per access, block 4096 Bytes