Recursive Trees for Practical ORAM

Tarik Moataz[1,*] Erik-Oliver Blass[2] Guevara Noubir[2]
[1]Dept. of Computer Science, Colorado State University, Fort Collins, CO
and IMT, Telecom Bretagne, France
tmoataz@cs.colostate.edu

[2]College of Computer and Information Science, Northeastern University, Boston, MA
{blass|noubir}@ccs.neu.edu

***Abstract*—We present a general construction to reduce the communication cost of recent tree-based ORAMs. Contrary to trees with constant height and path lengths, our new construction r-ORAM provides varying, shorter path lengths. Accessing an element in the ORAM tree will have different communication cost 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 demonstrate that the expected cost saving is around 35% for binary tree ORAMs. For a k-ary tree-based ORAM, we still can reduce cost with r-ORAM, e.g., 20% for $k = 4$. Besides reducing communication cost, r-ORAM also reduces storage overhead on the server by 20%. To prove r-ORAM's soundness, we conduct a detailed overflow analysis. We stress that r-ORAM is general and can be applied to all recent tree ORAMs, both constant memory or poly-log client memory ORAMs.**

I. 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, 27]. 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 delicate 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 new attention. 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, 19, 20, 24–26]. One idea is to store the N elements in a (k-ary) tree of N leaves. With constant client memory complexity, some results achieve $O(\log^3 N)$ communication complexity, e.g., Shi et al. [24] and derivatives, while poly-logarithmic client memory allows for $O(\log^2 N)$ communication complexity, e.g., Stefanov et al. [26]. Although poly-logarithmic communication complexity renders ORAMs affordable, further reducing (monetary) cost is still important for the real-world. 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 $\log N$ [24] or z [26] data elements, where z is a (small) security parameter such as 4. So, downloading the whole path of nodes is a costly operation, involving the download of multiple data elements for each single element to be accessed.

A second cost factor for a client is the total storage required on the cloud provider to hold the ORAM construction. For an N element tree based ORAM with data elements of length l bits, the total storage a client has to pay for computes to, e.g., at least $(2N-1)\cdot \log N \cdot l$ [24] or $(2N-1)\cdot z \cdot l$ [26]. In addition, a "map" translating ORAM addresses to leaves in the tree needs to be stored, too. Although the overhead compared to the minimum storage requirements of $N \cdot l$ is small, further cost reductions are important topic in practice.

Technical Highlights: In this paper, we present a novel technique to reduce the *average* or *expected* path length, therefore reducing the cost to access elements in an ORAM in practice. The idea behind our technique called r-ORAM is to store data elements in a recursive tree structure. 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 that contains the actual ORAM data elements. The maximum, worst-case path length of r-ORAM is equal to $c \cdot \log N$, with $c = 0.78$ where 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 storage, due to fewer nodes in the recursive trees, by a factor of 0.8.

r-ORAM is general in that it 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. [24], $O(\log N)$ client memory such as Stefanov et al. [26], 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 k-ary trees.

II. Recursive Binary Trees

A straightforward approach: To motivate the rationale behind r-ORAM, we start by describing a straightforward attempt to reduce the path length and therewith communication

*Work done while at Northeastern.

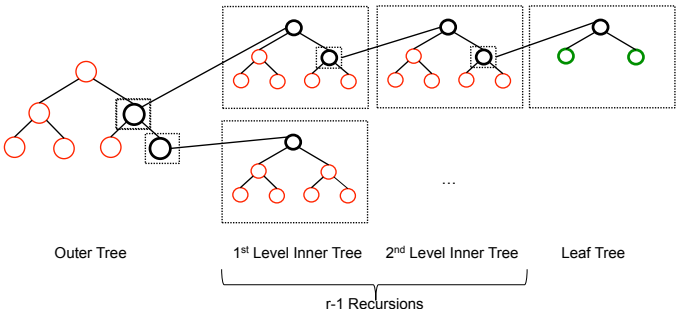

Fig. 1. Structure of an r-ORAM

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 $\log N$.

Now, one straightforward idea to reduce path lengths would be to percolate elements to any node in the tree, not only leafs, 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 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 average or expected path length would be $\approx \log(N) - 1$, not given much of a saving.

We can already see that another approach is required, 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 represents 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 a 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 in our construction.

Let the outer tree have y leaves and height $\log y$, where y is a power of two and $\log$ the logarithm base two. Also, inner trees have y leaves and height $\log y$. Leaf trees have x leaves, respectively, and height $\log x$. As in related work on tree-based ORAM [24], the number of elements N that will be stored in an r-ORAM equals the total number of leaves in all leaf trees.

A. r-ORAM Operations

First of all, 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, 24, 26], i.e., it supports the operations *Add*, *ReadAndRemove*, and *Evict*. It follows the same strategy of address mapping as the one defined in previous tree-based ORAMs – we detail this in Section II-F. For now, assume that every leaf in r-ORAM has a unique identifier called *tag*. Every element a stored in an r-ORAM is uniquely defined by its address d. 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 k slots.

We follow ORAM operations introduced by Shi et al. [24] that simulate the traditional *Read* and *Write* operations of classical ORAMs [10]. To simulate *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.

First, we detail *Add* and *ReadAndRemove* operations, and we postpone the eviction operation to a subsequent paragraph due to its complexity. We only *overview* the operations, for details refer to Shi et al. [24].

- *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 slot 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 poly-logarithmic in N local memory available that he may use during operations. For each category, we use different eviction techniques that we present in the following two paragraphs.

a) Constant Client Memory: The eviction operation is directly performed *after* an *Add* operation. The algorithm *Evict* is similar to the one by Shi et al. [24]:

Evict(χ, t): Let $\mathcal{S} = \{\mathcal{P}, \text{such that } |\mathcal{P}| = |\overrightarrow{Rt}|\}$ 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 (from paths in $\mathcal{S}$) 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.

b) Poly-Logarithmic Client Memory: For the case of poly-logarithmic client memory, the eviction operation follows that of Gentry et al. [8] and Stefanov et al. [26]:

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'. As eviction, the client pushes every element from nodes in $\mathcal{P}(t)$ that is 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 read or written in the root bucket during the *Add* operation, the client applies *Evict*, so he stores and at the same times evicts all elements as far as possible "down" on the path. Note that eviction can also be deterministic [8], or randomized [26].

B. Security definition

As any ORAM construction, r-ORAM should meet the typical obliviousness requirement that we briefly repeat for completeness sake.

Definition II.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* $\overrightarrow{a}$*. 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* $A(\overrightarrow{b})$*,*

$$|Pr[\mathcal{D}(A(\overrightarrow{a})) = 1] - Pr[\mathcal{D}(A(\overrightarrow{b})) = 1]| \leq \epsilon(s),$$

where s *is a security parameter, and* $\epsilon(s)$ *a negligible function in* s*.*

On a side note, we also assume that, 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.

C. 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:

$$\begin{aligned} N &= (2y-2) \cdot (2y-2)^{r-1} \cdot x \\ &= (2y-2)^r \cdot x \qquad (1) \\ &= 2^r \cdot x \cdot (y-1)^r. \qquad (2) \end{aligned}$$

Each of the nodes in an r-ORAM is a *bucket* ORAM of size k, where k is a security parameter, e.g., $k > \log N$ [24]. Note that for $x = 2$ and $y = 2$, the r-ORAM structure is exactly a full binary tree with a height equal to $r + 1$. The total number of nodes ν in an r-ORAM 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.,

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

Thus, the total storage cost for r-ORAM is $\nu \cdot k \cdot l$ with data elements of size l bits. For appropriate choices of x and y that will be discussed in the next section, r-ORAM saves storage costs compared to the $(2N-1) \cdot k \cdot l$ bits of storage of related work. In particular, $x \geq 2$, $y \geq 2$ and $\frac{2y-2}{2y-3} - 2 < 0$. So for example, with $x = 2$ and $y = 4$, the storage is equal to $\frac{8N}{5}$ resulting in a storage reduction by 20% compared to existing tree-based ORAMs.

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.

D. 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 k entries, and each entry of size l bits. Note that in related work, any element requires the client to read a *fixed* number of $\log N \cdot l \cdot k$ 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 k and the size of each entry l.

A path going over a node on the i^{th} level in the outer tree requires reading one bucket ORAM more 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 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 before its leaf level, before finally reading the corresponding leaf tree. The worst-case cost $\mathcal{C}$ equals

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

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

$$\mathcal{B} = r + \log x. \qquad (4)$$

The worst-case cost in this setting is in function of three parameters that must be carefully chosen to minimize worst- and best-case cost. The following Theorem II.1 summarizes how recursion factor r, the number of leaves y in inner trees, and the number of leaves in leaf trees x have to be chosen.

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.

So, 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 II.1. *If* $r = \log((\frac{N}{2})^{\frac{1}{2.7}})$, $x = 2$, *and* $y = \frac{1}{2} \cdot (\frac{N}{2})^{\frac{1}{r}} + 1$, $\mathcal{C}$ *is minimal 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 $\mathcal{B}$ *equals*

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

Proof: Function $\mathcal{C}$ depends on three variables that we can reduce to two by plugging 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 equals

$$\mathcal{C}(r, y) = \log(N) - r + r \cdot \log(\frac{y}{y-1}). \tag{5}$$

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:

$$\begin{aligned} N &\geq 2 \cdot (2y - 2)^r \\ \Rightarrow \; y &\leq \frac{1}{2} \cdot \frac{N}{2}^{\frac{1}{r}} + 1 \end{aligned}$$

For small x, we therewith get a larger upper bound for y. By fixing x to its minimum 2, the optimum number of leaves for the inner trees is $y = \frac{1}{2} \cdot (\frac{N}{2})^{\frac{1}{r}} + 1$. Putting these values back in Eq. (5), results in $\mathcal{C}$ depending on only one variable r, the recursion factor:

$$\mathcal{C}(r) = 1 + r \cdot \log(\frac{1}{2} . (\frac{N}{2})^{\frac{1}{r}} + 1) \tag{6}$$

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 $r \leq \log(N) - 1$. We replace r' in equations (4) and (6), and this concludes our proof. ∎

E. 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. Note that due to the recursive structure of the r-ORAM, the average-case cost is not uniformly distributed.

In order to determine the average-case cost, we count, for each path length i, the number of leaves that 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 II.2. *Let* $\log y$ *be the hight of inner trees and* $\log x$ *the height of leaf trees. The average cost is*

$$\mathcal{A}_v = \frac{\sum_{i=\mathcal{B}}^{\mathcal{C}} i \cdot \mathcal{N}(i)}{N},$$

where

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

Proof: Note that 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. So, $\mathcal{N}(i) = \mathcal{R}(i) \cdot \mathcal{L}(i)$.

As a binary leaf tree has height $\log x$, $\mathcal{L}(i) = 2^{\log x} = x$.

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 j^{th} inner tree that have to be read, i.e., the maximum level in the j^{th} tree that $\mathcal{P}$ covers. For a path $\mathcal{P}$

of length i, we have $i = \sum_{j=1}^{r} A_r[j] + \log(x)$. Note that 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 $\mathcal{I}(i)$, the *total* number of paths equals

$$\mathcal{R}(i) = 2^{i-\log(x)} \cdot \mathcal{I}(i),$$

for any integer $i \in \{\mathcal{B}, \ldots, \mathcal{C}\}$.

We can compute $\mathcal{I}(i)$, by computing the number of solutions of the equation

$$\begin{gathered} A_r[1] + A_r[2] + \cdots + A_r[r] = i - \log x \\ \Leftrightarrow \\ (A_r[1] - 1) + (A_r[2] - 1) + \cdots + (A_r[r] - 1) \\ = i - r - \log x. \end{gathered} \tag{7}$$

Computing the number of solutions of Eq. (7) 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 leads to

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

With $\mathcal{N}(i) = 2^i \cdot \mathcal{I}(i)$, 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.

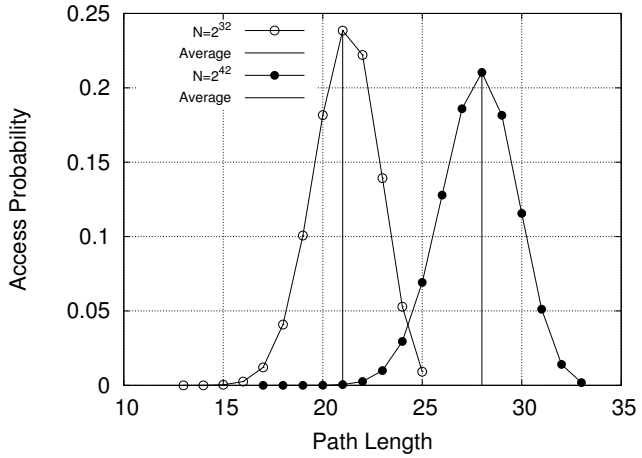

Fig. 2. r-ORAM path length distribution, $N = 2^{32}$ and $N = 2^{42}$ elements

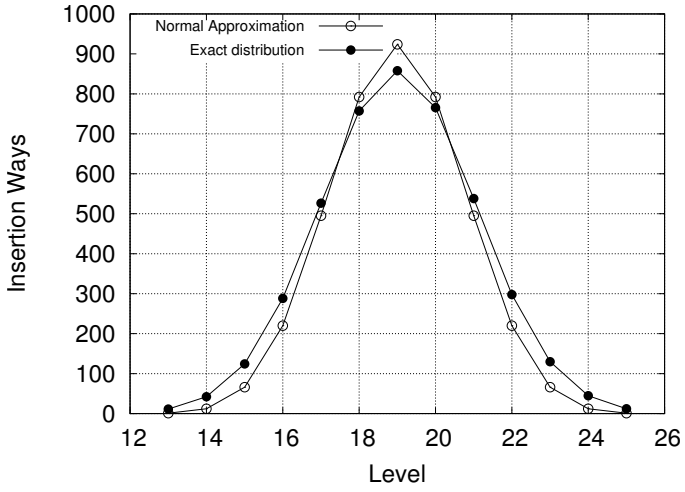

Fig. 3. Normal approximation

Our average-case equation can be simplified: the number of possibilities $\mathcal{I}$ of indistinguishable balls packing in distinguishable bins can be approximated by a normal distribution [3, 4], such that for a given level $i \in \{\mathcal{B}, \cdots, \mathcal{C}\}$ we have:

$$\mathcal{I}(i) \approx \frac{A}{s\sqrt{2\pi}} \cdot e^{-\frac{(i-r-\log(x)-\frac{c}{2})^2}{2s^2}}, \tag{8}$$

where $c = r \cdot (\log(y) - 1)$, $s = \frac{\frac{c}{2}+1}{z}$, $A = r \cdot \log(y)$, and z being the solution of the equation

$$ze^{-\frac{z^2}{2}} = \frac{\sqrt{2\pi} \cdot (\frac{c}{2} + 1)}{A}.$$

Figure 3 depicts the accuracy of a normal approximation, by comparing $\mathcal{I}(i)$ and the normal approximation for $N = 2^{32}$.

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 average page lengths for various configurations and compare it to related work in Table I.

F. 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 [8, 24, 26]. The main difference is the introduction of the new recursion that 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.

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

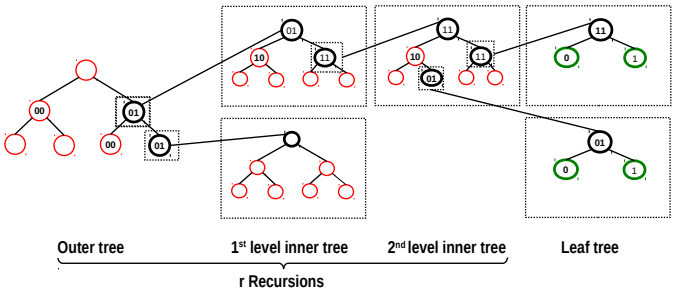

Fig. 4. r-ORAM Map addressing

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 4 shows an address mapping example for two leaf nodes.Finally the mapping is stored in a position map structure following the recursive construction in [26]. To access the position map, the communication cost has, as in r-ORAM, a best-case equal to $O(\mathcal{B} \cdot \log^2(n) \cdot k)$ bits and a worst case equal to $O(\mathcal{C} \cdot \log^2(n) \cdot k)$ bits, where k is the number of entries.

III. K-ARY TREES

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

For parameters x and y as above, the number of elements N can be computed by calculating the number of nodes in the outer/inner k-ary tree for r recursions:

$$\begin{aligned} N &= (\sum_{i=0}^{\log_k y} k^i - 1)^r \cdot x \\ &= (\frac{1 - k^{1+\log_k y}}{1-k} - 1)^r \cdot x \\ &= (\frac{k}{k-1} \cdot (y-1))^r \cdot x \end{aligned} \tag{9}$$

The following theorem underlines how one should choose the recursion factor r, the height of the inner trees $\log x$ and leaf trees $\log x$ to minimize cost.

Theorem III.1. *If* $r = \log_k((\frac{N}{k})^{\frac{1}{f(k)}})$, $x = 2$, *and* $y = \frac{k-1}{k} \cdot (\frac{N}{k})^{\frac{1}{r}} + 1$, *the optimum values for the best and worst-case cost equal*

$$\mathcal{C} = 1 + \log_k((\frac{N}{k})^{\frac{1}{f(k)}}) \cdot \log_k((k-1) \cdot k^{f(k)-1} + 1),$$

and

$$\mathcal{B} = 1 + \frac{1}{f(k)} \cdot \log_k(\frac{N}{k}),$$

where $f(k) > 1$ *is a decreasing function in* k.

The decreasing function f depends on the choice of k, the branching factor. For $k = 4$, $f(4) \approx 2$, while for $k = 16$, $f(16) \approx 1.6$. The proof of the Theorem III.1 is similar to the proof of Theorem II.1, so we will only sketch this proof, highlighting differences.

Proof (Sketch): The first step in the proof is to represent the number of leaves x as a function of N, y, r, and k the branching factor. That is, we decrease the number of variables in our optimization problem by one, such that $\log_k(x) = \log_k(\frac{k-1}{k}N) - r \cdot \log_k(y-1)$ – which is just the result of applying the logarithm over Eq. (9). Since our first goal is the minimization of the worst-case cost, we substitute $\log_k(x)$ in the worst-case cost Eq. (3) by the value computed in the above equation and we try to minimize the new function of the worst-case cost, which has one variable less. Note that the logarithm is base k 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 II.1 to find the optimal recursive factor r, the number of leaf tree leaves x, and the number of inner/outer tree leaves y. ■

Example: For $k = 4$, the optimum values for the best and worst-case cost are $\mathcal{C} = 0.95 \log N$ and $\mathcal{B} = 0.55 \cdot \log N$.

IV. PERFORMANCE ANALYSIS

Even if the worst-case complexity is in $O(\log(N))$, the constants gained with r-ORAM are significant. Table I compares between the height of a binary tree, as the one used in recent tree-based ORAMs [7, 20, 24, 26] and the new height of the r-ORAM structure. Also, we compare r-ORAM on k-ary trees, instead of binary ones, and we show that the recursive k-ary tree r-ORAM provides better performances in terms of height access communication cost.

Table I has been generated using parameters from theorems II.1 and III.1. Note that 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. Also, Table I 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. In Table II, we show the gain in percentage of r-ORAM applied

		Number of elements			
		2^{10}	2^{20}	2^{40}	2^{60}
Binary ORAM tree [7, 20, 24, 26]		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, 26]		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 I. TREE'S HEIGHT COMPARISON BETWEEN THE BINARY/4-ARY ORAM TREE AND BINARY/4-ARY r-ORAM

	Gain in %		
	Best-case	Average-case	Worst-case
Binary Tree based ORAM [7, 20, 24, 26]	60	35	22.5
4-ary Tree based ORAM [7, 8, 26]	45	20	5

TABLE II. TREE BASED ORAM GAIN PERCENTAGE

to binary ORAM trees, not distinguishing whether the scheme has constant or poly-log memory complexity..

As you can see, we gain 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 k-ary trees is less due to the reduction of the height of the tree. Trees are already "flat", so the recursion effect loses its impact.

We also present a monetary comparison of communication and storage overhead between tree based ORAM constructions (with constant and poly-log client memory). For this, we have taken blocks with size equal to 1 KBytes, and the number of entries (blocks) in every node is equal to $\log(N)$ where N is the number of elements stored in the ORAM. For this evaluation, we take communication and storage overhead of the position map into account as well as the overhead induced by eviction.

First, Fig. 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 ($z = 4$) gives the smallest communication overhead. For example, with a dataset of 1 TB, an access will cost 178 KBytes in total. Moreover, if we set the number of entries z to 2 instead of 4 [7], communication cost will be divided by 2.

The storage overhead induced by tree based ORAMs is still significant. Poly-log client memory ORAMs perform better, but still induce a factor of 10. r-ORAM reduces this overhead down to a factor of 8, i.e., a reduction by 20%. Based on Amazon prices [1], storing 1 TB of data cost around 225 USD per month.

Finally, we calculate the cost in USD associated to every access, shown in Fig. 6. Since we obtain the best communication overhead using r-ORAM over Path ORAM, one would expect this to be the cheapest construction, too. However, Amazon S3 prices rules are based not only on the communication in term of transferred bits, but also on the number of HTTP operations performed, e.g., GETs and PUTs. Surprisingly, the construction by Gentry et al. [8] with a k-ary and branching factor $k = \log(N)$, is cheaper as it involves fewer operations compared to Path ORAM.

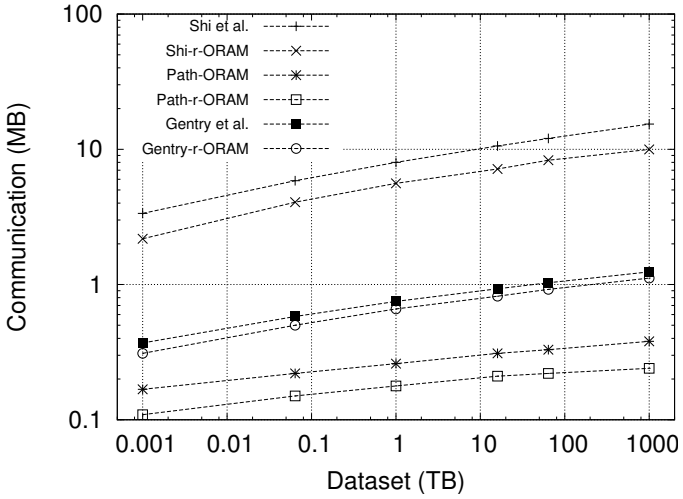

Fig. 5. Comparison of communication in MB per access

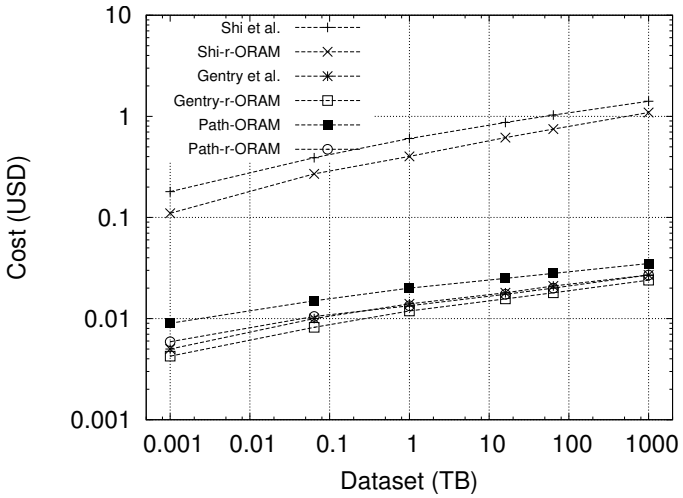

Fig. 6. Comparison of communication cost in USD per access

V. SECURITY ANALYSIS

A. Privacy analysis

Theorem V.1. *r-ORAM is a secure ORAM following Definition II.1, if every node (bucket) in r-ORAM 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. [26] and show that the sequence of tags t in an access pattern is a 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, too. ■

B. Overflow probability

We now determine the ORAM overflow probability for two cases, r-ORAM applied in a constant client memory and with poly-log client memory.

For the first case, we consider an eviction similar to the one used by Shi et al. [24]. 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 as the one used by Gentry et al. [8]. 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.

c) 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. At the end, 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. Refer to Fig. 7 for a high-level overview of this general tree model of r-ORAM.

In the i^{th} level, we may have leaf nodes as well as interior nodes. Also note that 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 i^{th} 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.

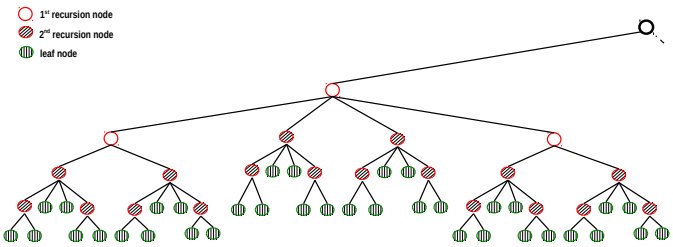

Fig. 7. Structure of an r-ORAM

We present a relation between $I(i)$, the interior nodes, and $\mathcal{N}(i)$, the leaf nodes, for a level $i > r$, where r is the recursion factor. Note that, for other levels $i \leq r$, there cannot be leaf nodes. Also, note that the leaves of the genral tree are the leaves of the leaf trees.

Lemma V.1. *For any $i > r$, the following inequalities hold.*

$$e^{-f(r,x,y)} \leq \frac{I(i)}{\mathcal{N}(i)} \leq 2^{-\log(x)} \cdot r, \tag{10}$$

where $f(r,x,y) = \frac{1+r\log(y)-r-2\log(x)}{2s^2}$ *and* $s > 0$.

Proof: First, we determine the number of interior nodes for $i > r$. In the same spirit as the proof of Theorem II.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. (7) of Theorem II.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 4^{th} 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 that $(2,2)$ or $(3,1)$ etc. but we do not have $(4,0)$ because it is already accounted for in the first sub-equation. We continue like this 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 II.2, we know that the number of solutions for the last equation S_r equals $2^{i-\log x} \cdot \mathcal{I}(i)$. Therefore, with the result of Theorem II.2, we can conclude that:

$$I(i) \leq 2^{i-\log(x)} \cdot r \cdot \mathcal{I}(i).$$

Also from Th. II.2, the number of leaves $\mathcal{N}(i) = 2^i \cdot \mathcal{I}(i)$. 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 $\mathcal{N}(i) = 2^i \cdot \mathcal{I}(i)$,

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

We have previously shown that $\mathcal{I}$ can be approximated by normal distribution, cf. Eq. (8). Using this approximation, we continue

$$\frac{I(i)}{\mathcal{N}(i)} \geq e^{-\frac{1+2i-2r-2\log(x)-c}{2s^2}}.$$

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. ∎

Now, we will show that once we have found 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 $i \leq r$, we have $\mathcal{N}(i) = 0$. We write $\mu = \frac{L(i+1)}{L(i)}$ (this will represent the expected value in Theorem V.3). We present our result in the following lemma.

Lemma V.2. *For $1 \leq i \leq \mathcal{C}$, μ is bounded by*

$$2 \cdot X(i) \leq \mu \leq 4 \cdot X(i), \tag{11}$$

where $X(i) = 1 - \frac{\mathcal{N}(i)}{L(i)}$.

Proof: This results follows from two observations. First, the total number of interior nodes for the i^{th} 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 i^{th} 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,

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

∎

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

Now, we are ready to present our two 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. [24], based on a standard balls into bins game. We omit details for this specific case.

Theorem V.2. *For eviction rate χ, if the number of data elements in an interior node is equal to k, the overflow probability of an interior node in the i^{th} level is at most θ_i^k, where, for $i \leq r$*

$$\theta_i = \frac{2^s}{2\chi},$$

and for $i > r$

$$\theta_i = \frac{2^s}{2\chi} \cdot (\frac{1}{1+\frac{x}{r}})^{i-r},$$

with $s = \lceil \log_4(\chi) \rceil$

Proof:

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

more than k elements) equals ρ_i^k. 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 i^{th} 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:

$$\lambda_i \leq \frac{2^s}{L(i+1)} \cdot \frac{1}{1 + \frac{\mathcal{N}(i)}{I(i)}} \cdots \frac{1}{1 + \frac{\mathcal{N}(s+1)}{I(s+1)}}. \tag{12}$$

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

$$\frac{1}{1 + \frac{\mathcal{N}(i)}{I(i)}} = 1, \tag{13}$$

because $\mathcal{N}(i) = 0$ (there is no leaf node for $i \leq r$). For $i > r$, using the result of Lemma V.1.

$$\frac{1}{1 + \frac{\mathcal{N}(i)}{I(i)}} \leq \frac{1}{1 + \frac{x}{r}}, \tag{14}$$

where x is the number of leaves. For buckets at level $i > r$, we plug the result of equations 13 and 14 in 12 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 V.2, 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^k.

For $i \leq r$, there are no leaves (i.e. $\mathcal{N}(i) = 0$), and the arrival rate is always bounded from Eq. 12 such that

$$\lambda_i \leq \frac{2^s}{L(i+1)}.$$

Consequently, dividing by η_i and using the result of Lemma V.2 $\frac{I(i)}{L(i+1)} < \frac{1}{2}$ 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.

■

Note that in practice the eviction rate is equal to 2 and s is then equal to 1.

Let us now take the case where r-ORAM is applied over tree ORAMs *with* client memory.

Theorem V.3. *For any interior node v, the probability that the node has size at least equal to $(1+\delta) \cdot \mu$ is at most $e^{-\frac{\delta^2 \cdot \mu}{1+\delta}}$, where*

$$F_1 \leq \mu \leq F_2,$$

For $i \leq r$:

$$F_1 = 2 \text{ and } F_2 = 4,$$

for $i > r$:

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

where $f(r,x,y) = \frac{1 + r\log(y) - r - 2\log(x)}{c}$ and $c > 0$

Proof:

Let us fix an interior node v in r-ORAM belonging to the i^{th} 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 i^{th} 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 i^{th} 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 i^{th} level or above. Let us associate for every element j in $\mathcal{T}$, an indicator random variable χ_j which is equal to 1 if the element was assigned path going through the interior node v. All elements in $\mathcal{T}$ are i.i.d. and their assignment probability, $\Pr(\chi_j = 1) = \frac{1}{L(i)}$. 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 compute Chernoff bound, we calculate the expected value of the sum of the indicator random 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 i^{th} 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 V.2 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 V.1. For $i \leq r$, from Lemma V.2, 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 the Chernoff bound, for any $\delta > 0$ and for both cases:

$$\begin{aligned} \Pr(S_t(v) \geq (1+\delta) \cdot \mu) &\leq \Pr(\sum_{j \in [L(i+1)]} \chi_j \geq (1+\delta) \cdot \mu) \\ &\leq e^{-\frac{\delta^2 \cdot \mu}{2+\delta}}. \end{aligned}$$

This concludes our proof. ∎

d) Discussion: 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 the 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}.

VI. Related Work

ORAM, first presented by Goldreich and Ostrovsky [10] has recently received a lot of attention [2, 5–13, 19, 20, 22–24, 26, 28, 29]. The current state of the art on ORAM can be divided into two mains categories. The first one comprises schemes where a client is only required to feature constant memory on his side. The second category assumes the client having sub-linear local memory.

A. 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, 19, 20, 22–24]. Goodrich and Mitzenmacher [11] and Pinkas and Reinman [23] 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. [24] 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. [24] present a tree-based ORAM, where the node of the tree are small bucket ORAMs, see also [10, 21]. Accessing an element in this structure consists of 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., $O(\log^3(N))$.

Gentry et al. [8] enhance previous work by modifying the structure of the tree. Instead of a binary tree, a multi-dimensional tree with a branching factor k 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. However, the scheme by Gentry et al. [8] suffers from not being fully applicable in a memoryless setting. This is due to its eviction algorithm, where the client has to memorize elements in order to percolate them towards leaves.

Mayberry et al. [20] improve complexity of the tree-based ORAM by Shi et al. [24]. Instead of using traditional ORAM bucket nodes in the tree, a PIR [18] 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 k-ary tree instead of a binary tree.

Kushilevitz et al. [19] present a hierarchical solution that enhances the asymptotic communication complexity defined in previous works with a worst case equal to $O(\frac{\log^2(N)}{\log(\log(N))})$. In practice, the scheme suffers from hidden constants that render it *practically* less efficient compared to, e.g., [20, 24].

B. Sub-linear client memory

Recent research with $O(\sqrt{N})$ client-side memory [28, 29] has sub-linear amortized communication complexity, but linear worst-case complexity. Boneh et al. [2] improve the worst case to be in $O(\sqrt{N})$, however still with $O(\sqrt{N})$ client-side memory. Stefanov et al. [25] present how to reduce amortized cost to be poly-logarithmic in $O(\log^2(N))$, but again with a large $O(\sqrt{N})$ client memory.

Stefanov et al. [26] present a Path ORAM, a seminal tree-based ORAM construction, based on Shi et al. [24], but including a client-side memory *stash* of size $O(\log(N))$. This

results in $O(\log^2(N))$ communication complexity. Fletcher et al. [7] present some enhancements upon Path ORAM. They enhance communication costs by $6 \sim 7\%$. Also, authors reduce "latency" by decreasing the number of encryptions performed on the client side.

Our work r-ORAM can be considered as a general technique that is applicable to and improves any tree-based ORAM construction – either in settings with constant client memory or sub-linear client memory.

VII. Conclusion

r-ORAM is a technique for ORAM cost reduction in practice. r-ORAM improves both communication cost as well as storage cost. For any binary tree based ORAM, the average cost is reduced by 35%, and storage cost is reduced by 20%. We have formally shown 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. In future work, we investigate the dynamics of r-ORAM, i.e., instead of considering constant height for outer and inner trees, we aim at varying the height, which promises even further cost reductions.

References

[1] Amazon. Amazon s3 pricing. http://aws.amazon.com/s3/pricing/, 2014. [Online; accessed 31-July-2014].

[2] Dan Boneh, David Mazières, and Raluca Ada Popa. Remote oblivious storage: Making oblivious RAM practical. http://dspace.mit.edu/bitstream/handle/1721.1/62006/MIT-CSAIL-TR-2011-018.pdf, March 1996.

[3] Kevin Brown. Balls in bins with limited capacity. http://www.mathpages.com/home/kmath337.htm, 2014. [Online; accessed 30-July-2014].

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

[5] Kai-Min Chung and Rafael Pass. A Simple ORAM. *IACR Cryptology ePrint Archive*, 2013:243, 2013.

[6] Ivan Damgård, Sigurd Meldgaard, and Jesper Buus Nielsen. Perfectly Secure Oblivious RAM without Random Oracles. In *Proceedings of Theory of Cryptography Conference –TCC*, pages 144–163, Providence, RI, USA, March 2011.

[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] Oded 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, NY, USA, 1987.

[10] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *J. ACM*, 43(3): 431–473, 1996.

[11] Michael T. Goodrich and Michael 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] Michael T. Goodrich, Michael 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, IL, USA, 2011.

[13] Michael T. Goodrich, Michael Mitzenmacher, Olga Ohrimenko, and Roberto 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] Leonard Kleinrock. *Theory, Volume 1, Queueing Systems*. Wiley-Interscience, 1975. ISBN 0471491101.

[18] 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, FL, USA, 1997.

[19] Eyal Kushilevitz, Steve Lu, and Rafail 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.

[20] Travis Mayberry, Erik-Oliver Blass, and Agnes Hui Chan. Path-pir: Lower worst-case bounds by combining oram and pir. In *Proceedings of the Network and Distributed System Security Symposium*, San Diego, CA, USA, 2014.

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

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

[23] Benny Pinkas and Tzachy Reinman. Oblivious ram revisited. In *Advances in Cryptology – CRYPTO*, pages 502–519, Santa Barbara, CA, USA, 2010.

[24] 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.

[25] Emil Stefanov, Elaine Shi, and Dawn Xiaodong Song. Towards practical oblivious ram. In *Proceedings of the Network and Distributed System Security Symposium*,

San Diego, CA, USA, 2012. The Internet Society.

[26] 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.

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

[28] Peter Williams and Radu Sion. Usable pir. In *Proceedings of the Network and Distributed System Security Symposium*, San Diego, CA, USA, 2008.

[29] Peter Williams, Radu Sion, and Bogdan 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, Virginia, USA, 2008.