Resizable Tree-Based Oblivious RAM

Tarik Moataz[1,*,§], Travis Mayberry[2,§], Erik-Oliver Blass[3], and Agnes Hui Chan[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
{travism|ahchan}@ccs.neu.edu
[3] Airbus Group Innovations, 81663 Munich, Germany
erik-oliver.blass@airbus.com

Abstract. Although newly proposed, tree-based Oblivious RAM schemes are drastically more efficient than older techniques, they come with a significant drawback: an inherent dependence on a fixed-size database. Yet, a flexible storage is vital for real-world use of Oblivious RAM since one of its most promising deployment scenarios is for cloud storage, where scalability and elasticity are crucial. We revisit the original construction by **?**] and propose several ways to support both increasing and decreasing the ORAM's size with sublinear communication. We show that increasing the capacity can be accomplished by adding leaf nodes to the tree, but that it must be done carefully in order to preserve the probabilistic integrity of data structures. We also provide new, tighter bounds for the size of interior and leaf nodes in the scheme, saving bandwidth and storage over previous constructions. Finally, we define an oblivious pruning technique for removing leaf nodes and decreasing the size of the tree. We show that this pruning method is both secure and efficient.

1 Introduction

Oblivious RAM has been a perennial research topic since it was first introduced by **?**]. ORAM allows for an access pattern to an adversarially controlled RAM to be effectively obfuscated. Conceptually, a client's data is stored in an encrypted and shuffled form in the ORAM, such that accessing pieces of data will not produce any recognizable pattern to an adversary which observes these accesses. Being a powerful cryptographic primitive, many additional uses besides storage can be envisioned for ORAM, such as an aid for homomorphic circuit evaluation, secure multi-party computation, and privacy-preserving data outsourcing. Given the advent of cloud computing and storage, and all their potential for abuse and violation of privacy, ORAM schemes are important for the real-world today.

[*]Work done while at Northeastern University.
[§]Both authors are first authors.

A crucial aspect of ORAM schemes is their implied overhead. In today's cloud settings, the choice to use the cloud is chiefly motivated by cost savings. If the overhead is enough that it negates any monetary advantages the cloud can offer, the use of ORAM will be impractical. Previous ORAM schemes have had a common, major drawback that has hindered real-world use: due to eventually necessary "reshuffling" operations, their worst-case communication complexity was linear in the size of the ORAM. Recent works on ORAM, e.g., by **?**], **?**], and many derivatives, have proposed new ORAM schemes that are tree-based and have only poly-logarithmic worst-case communication complexity.

However, new tree-based approaches have exposed another barrier to the real-world adoption of ORAMs: the maximum size of the data structure must be determined during initialization, and it cannot be changed. This is not an issue in previous linear schemes, because the client always had the option of picking a new size during the "reshuffling", being effectively a "reinitialization" of the ORAM. In tree-based ORAMs, though, a reinitialization ruins the sublinear worst-case communication complexity.

Resizability is a vital property of any ORAM to be used for cloud storage. One of the selling points of cloud services is elasticity, the ability to start with a particular footprint and seamlessly scale resources up or down to match demand. Imagine a startup company that wants to securely store their information in the cloud using ORAM. At launch, they might have only a handful of users, but they expect sometime in the long-term to increase to 10,000. With current solutions, they would have to either pay for the 10,000 users worth of *storage* starting on day one, even though most of it would be empty, or pay for the *communication* to repeatedly reinitialize their database with new sizes as they become more popular. Reinitializing the ORAM would negate any benefit from the new worst-case constructions. Additionally, one can imagine a company that is seasonal in nature (e.g., a tax accounting service) and would like the ability to downsize their storage during off-peak times of the year to save costs.

Consequently, the problem of resizing these new tree-based ORAMs is important for practical adoption in real-world settings. In light of that, we present several techniques for both increasing and decreasing the size of recent tree-based ORAMs to reduce both communication and storage complexity. We focus on constant client memory ORAM (the **?**] ORAM) since it is an interesting setting, especially for hardware-constrained devices and large block sizes or situations where multiple parties want to share the same ORAM so need to exchange the state. We are able to show that, although the resizing techniques themselves are intuitive, careful analysis is required to ensure security and integrity of ORAMs. In addition, we show that it is nontrivial to both allow for sublinear resizing and maintain the constant client memory property of **?**] ORAM.

The technical highlights of this paper are as follows:

1. Three provably secure strategies for increasing the size of tree-based ORAMs, along with a rigorous analysis showing the impact on communication and storage complexity and security.
2. A provably secure method for pruning the trees to decrease the size of a tree-based ORAM, again including rigorous analysis showing that security and integrity of the data structures is preserved.

3. A new, tighter analysis for the **?**] ORAM which allows for smaller storage requirements and less communication per query than previous work.

2 Building Blocks

We will briefly revisit the constant-client memory tree-based ORAM of **?**], focusing on the relevant details which are necessary to understand our resizing techniques.

2.1 Preliminaries

An Oblivious RAM is a cryptographic data structure storing blocks of data in such a way that a client's pattern of accesses to those blocks is hidden from the party which holds them. ORAMs offer block reads and writes. That is, they provide $\mathsf{Read}(a)$ and $\mathsf{Write}(d, a)$ operations, where a is the address of a block, and d notes some data. Let N be the total number of blocks the ORAM can store. Each ORAM block is uniquely addressable by $a \in \{0,1\}^{\log N}$, and the size of each block is ℓ bits.

Data in the ORAM [**?**] is stored as a binary tree with N leaves. Each node in the tree represents a smaller ORAM *bucket* [**?**] which holds k (encrypted) blocks. When clear from the context, we will use the terms node and bucket interchangeably. Each leaf in the tree is uniquely identified by a tag $t \in \{0,1\}^{\log N}$. With $\mathcal{P}(t)$, we denote the path which starts at the root of the tree and ends at the leaf node tagged t.

Blocks in the ORAM are associated with leaves in the tree. The association between blocks and their addresses is a lookup table with size equal to $N \cdot \log N$. This table is called the *position map*, and in order to maintain efficiency it is recursively stored in series of smaller ORAMs [**?**]. The central invariant of tree-based ORAMs is that a block tagged with tag t will always be found in a bucket somewhere on the path $\mathcal{P}(t)$. Blocks will enter the tree at the root and propagate toward the leaves depending on their tag.

2.2 Tree-based Construction

?]'s ORAM implements Read and Write operations by applying, first, $\mathsf{ReadAndRemove}(a)$ operation, followed by an $\mathsf{Add}(d, a)$. A $\mathsf{ReadAndRemove}(a)$ will first fetch the tag t from the position map, thereby determining the path $\mathcal{P}(t)$ in the ORAM tree on which that block exists. The client will download all $\log N$ nodes in $\mathcal{P}(t)$, and decrypt all blocks. For each block $a' \neq a$ on path $\mathcal{P}(t)$, the client will upload back to the server a re-encrypted version of that block. For block a, the client will upload an encrypted *dummy* block, which is a special value signifying that the block is empty. The client does this in a bucket-by-bucket, block-by-block decrypt and encrypt manner, to keep client memory constant in N. As long as the encryption is secure, the server will not learn which block the client was interested in, because all they will see is fresh encryptions replacing every block in the path. For the Add operation, the client uniformly chooses a new tag $t \overset{\$}{\leftarrow} \{0, \ldots, N-1\}$ that associates block a to a new leaf, encrypts d and inserts the resulting ciphertext block into the root.

After every access, an *eviction* is performed to percolate blocks towards the leaves, freeing up space for new blocks to enter at the root. The eviction is a random process that chooses, in every level, ν buckets and evacuates randomly one real element to the corresponding child (as determined by its tag). To stay oblivious, the eviction accesses both child buckets in turn, thereby (re-)encrypting both buckets. Again, this is done in a block-by-block manner to keep client memory constant.

3 Resizable ORAM

3.1 Technical Challenges

The challenge behind resizing tree-based ORAMs is threefold:

1. Increasing the size of the tree will have an impact on the bucket size. A leaf node may become an interior node while increasing the ORAM, and vice versa in the decreasing case. The original analysis by **?**] differentiates between interior and leaf nodes, while for resizing we will have to generalize the analysis to consider both cases at once.
2. For $n > N$ elements, we must determine the most effective strategy of increasing the number of nodes to optimize storage and communication costs for the client.
3. Reducing the size of the tree is non-trivial, especially when targeting low communication complexity and constant client memory. A mechanism is required for moving elements from pruned nodes into other buckets in an oblivious, yet efficient way while still maintaining overflow probabilities.

3.2 Resizing Operations

To allow for resizing, we introduce two new basic operations by which a client can resize an ORAM, namely Alloc and Free:

– Alloc: Increase the size of the ORAM so that it can hold one additional element of size ℓ.
– Free: Decrease the size of the ORAM so that it can hold one element fewer.

3.3 Security Definition

Resizing an ORAM should not leak any information besides the current number of elements. Thus, we need to augment the standard ORAM security definition by our resizing operations.

Definition 31 *Let* $\overrightarrow{y} = \{(op_1, d_1, a_1), (op_2, d_2, a_2), \ldots, (op_M, d_M, a_M)\}$ *be a sequence of* M *operations* (op_i, d_i, a_i)*, where* op_i *denotes a* Read*,* Write*,* Alloc *or* Free *operation,* a_i *equals the address of the block if* $op_i \in \{\mathsf{Add}, \mathsf{ReadAndRemove}\}$ *and* d_i *the data to be written if* $op_i = \mathsf{Add}$.

Let $A(\overrightarrow{y})$ be the access pattern induced by sequence $\overrightarrow{y}$. A resizable ORAM is secure iff*, for any PPT adversary $\mathcal{D}$ and any two same-length sequences $\overrightarrow{y}$ and $\overrightarrow{z}$ where $\forall i \in [M] : \overrightarrow{y}(i) = \mathsf{Alloc} \Leftrightarrow \overrightarrow{z}(i) = \mathsf{Alloc} \wedge \overrightarrow{y}(i) = \mathsf{Free} \Leftrightarrow \overrightarrow{z}(i) = \mathsf{Free}$,*

$$|Pr[\mathcal{D}(1^\lambda, A(\overrightarrow{y})) = 1] - Pr[\mathcal{D}(1^\lambda, A(\overrightarrow{z})) = 1]| \leq \epsilon(\lambda),$$

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

For sake of completeness, considering buckets in resizable ORAM as trivial ORAMs [**?**], all blocks are IND-CPA encrypted. Also, whenever a block is accessed by any type of operation, its bucket is re-encrypted block-by-block.

4 Adding

We begin by describing a *naïve* solution that will add a new level of leaves when $n > N$. However, this already leads to a problem: when n is only slightly larger than N, we are using twice as much storage as we should need. The second strategy, *lazy expansion*, will postpone creation of an entire new level until we have enough elements to really need it. In both the naïve and second solution, there are thresholds causing large "jumps" in storage space. As this can be expensive, we present a third solution dubbed *dynamic expansion*. This strategy progressively adds leaf nodes to the tree, thereby gradually increasing the tree's capacity. This last strategy is particularly interesting, because it results in an unbalanced tree, requiring careful analysis to ensure low overall failure probability of the ORAM.

4.1 Tightening the bounds

Communication and storage complexities represent the core comparative factor between strategies, and both are dependent primarily on bucket sizes. Consequently, it is important to get a tight analysis for both interior and leaf bucket sizes. The original bounds for bucket sizes given by **?**] are substantially larger than necessary. Therefore, as a first contribution, we give new, tighter bounds for interior and leaf node sizes.

Interior Nodes We first address the size of interior nodes by using standard queuing theory. Let I_i denote the random variable for the size of interior nodes of the i^{th} level in the tree. For eviction rate ν, we compute the probability of a bucket on levels $i > \log \nu$ having a load of at least k (i.e., a size k bucket overflows) to:

$$\Pr(I_i \geq k) = \nu^{-k}. \tag{1}$$

In [**?**], the eviction rate was chosen to be equal to 2 with an overflow probability equal to 2^{-k}. However, if we adjust the bucket size to be $\frac{k}{\log(\nu)}$, the overflow probability is still 2^{-k}, namely $\Pr(I_i \geq \frac{k}{\log(\nu)}) = 2^{-k}$.

This follows from Eq. **??** by replacing k by $\frac{k}{\log(\nu)}$. Also, we can investigate the optimal value for the eviction rate ν in terms of communication cost. For $\nu = 4$, we obtain the same overflow probability as with $\nu = 2$ with buckets of half the size. The communication complexity does not change, as we are evicting twice as much, but with buckets of half the size. For larger eviction rates $\nu > 4$ the communication complexity becomes larger. Note that this also reduces the storage by a factor of 2. For N elements stored in the ORAM, the probability that an interior node overflows during eviction computes to

$$\begin{aligned}\Pr(\exists i \in [\nu \cdot \log N] :\ I_i \geq \frac{k}{\log(\nu)}) &= 1 - \Pr(\forall i \in [\nu \cdot \log N] :\ I_i < \frac{k}{\log(\nu)}) \quad (2)\\ &= 1 - \prod_{i=1}^{\nu \cdot \log N} (1 - \Pr(I_i \geq \frac{k}{\log(\nu)})) \quad (3)\\ &= 1 - (1 - 2^{-k})^{\nu \cdot \log N}.\end{aligned}$$

In particular for $\nu = 4$, the optimal choice of the eviction rate,

$$\Pr(\exists i \in [4 \cdot \log N] :\ I_i \geq \frac{k}{2}) = 1 - (1 - 2^{-k})^{4 \cdot \log N}.$$

The buckets that can overflow during an access are limited to those in the paths accessed during the eviction, i.e., $\nu \cdot \log N$ buckets accessed. Also, the number of buckets taken into account is actually $\nu \cdot \log N$ instead of $2\nu \cdot \log N$. This follows from the fact that for every parent, we write only one real element to one child. Consequently, per eviction and per level, only one child can overflow. For Eq. **??**, an equality still holds since the buckets can be considered independent in steady state [**?**].

Given security parameter λ, to compute the size of interior buckets, we solve the equation $2^{-\lambda} = 1 - (1 - 2^{-k})^{\nu \cdot \log N}$ to $k = -\log\left(1 - (1 - 2^{-\lambda})^{\frac{1}{\nu \cdot \log N}}\right)$.

For example, to have an overflow probability equal to 2^{-64}, $\lambda = 64$, $N = 2^{30}$, $\nu = 4$, the bucket size needs to be only 36 while **?**] determined the bucket size be equal 72 for the same overflow probability. Moreover, since N, the number of elements in the ORAM, has a logarithmic effect on the overflow probability, the size of interior nodes will not change for large fluctuations of the number of elements N. For example, for $N = 2^{80}$, the interior node still has size 36 with overflow probability 2^{-64}.

Leaf Nodes Let B_i denote the random variable describing the size of the i^{th} leaf node. Thinking of a leaf node as a bin, a standard balls and bins game argument provides us the following upper bound

$$\Pr(B_i \geq k) \leq \binom{N}{k} \cdot \frac{1}{N^k} \leq \frac{e^k}{k^k}.$$

The second inequality follows from an upper bound of the binomial coefficient using Stirling's approximation. For N leaves, we have

$$\begin{aligned}\Pr(\exists i \in [N]:\ B_i \geq k) &= \Pr(\bigcup_{i=1}^{N} B_i \geq k) \\ &\leq \sum_{i=1}^{N} \Pr(B_i \geq k) \\ &\leq \frac{N}{e^{k\cdot(\ln(k)-1)}}.\end{aligned} \tag{4}$$

Note that in Eq. **??**, we have used the union bound. Based on the same parameters as in the previous example, the size of a leaf node has to be set only to 28 to have an overflow probability equal to 2^{-64}. To compute this result, one solves the equation $k = e^{W(\frac{\log 2^{\lambda}\cdot N}{e})+1}$, where $W(.)$ is the product log function. While the size of the interior node can be considered constant for large fluctuations of N, the size of a leaf node should be carefully chosen depending on N. Every time the number of elements increases by a multiplicative factor of 32, we have to increase the size of the leaf node by 1 to keep the same overflow probability.

Note that for both interior and leaf node size computations, we do not take into account the number of operations (accesses) performed by the client. As with related work, the number of ORAM operations is typically considered part of security parameter λ. The larger the number of operations performed, the larger the security parameter has to be.

4.2 1st Strategy: naïve expansion

Let N and n respectively denote the number of leaf nodes and elements in the ORAM. The naïve solution is simply adding a new leaf level, as soon as the condition $n > N$ occurs. The main drawback of this first naïve solution is the waste of storage which can be explained from two different perspectives. The first storage waste consists on creating, in average, more leaf nodes than elements in the ORAM. The second storage waste in the under-usage of the leaf nodes while they can hold more elements with a slight size increase. Our second strategy will try to get rid of this drawback.

4.3 2nd Strategy: lazy expansion

This technique consists of creating a new tree level when the number of elements added is equal to α times the number of leaf nodes in the tree. For a N leaves tree, the client is allowed to store up to $\alpha \cdot N$ elements in the ORAM without increasing the size of the tree. As soon as $n > \alpha \cdot N$, the client asks the server to create a new level of leaves with $2 \cdot N$ leaf nodes.

This lazy increase strategy is performed recursively. For example, if the size of the ORAM tree is now equal to $2 \cdot N$, then the client will work with the same structure as long as $\alpha \cdot N < n \leq \alpha \cdot 2 \cdot N$. Once $n > \alpha \cdot 2 \cdot N$, a new level of leaves with now $4N$ leaf buckets is created.

To be able to store more elements, our idea is to slightly increase the leaf bucket size. Therewith, we can keep the same overflow probability. Note the tradeoff between increasing the size of leaf nodes and the communication complexity of the ORAM. To read or write an element in the ORAM, the client downloads the path starting from the root to the leaf node. If the size of this path (when increasing the size of the bucket) is larger than a regular ORAM tree with the same number of elements, then this technique would not be worth applying.

?] have shown that by increasing the leaf node size from k to $\alpha + k$, we can reduce the storage overhead while handling more elements than leaf nodes. For N leaf nodes, we can have up to $\alpha \cdot N$ elements. While **?**] chose α to optimize the storage cost for a given overflow probability, we instead target the computation of the value α for the optimal communication complexity. In our subsequent analysis, the previous bounds for interior and leaf node sizes as computed in Section **??** are used.

First, we determine a relation between the size x of a leaf bucket and factor α for our 2^{nd} strategy. Then, we compute the optimal value of α as a function of the security parameter λ, the size of the interior nodes, and the current number of leaves. To calculate the overflow probability, we focus on the worst case occurring when there are $\alpha \cdot N$ elements in an ORAM with N leaves.

Lemma 41 *Let x denote the optimal leaf bucket size for the 2^{nd} strategy. Then,*

$$\alpha = \frac{x}{e} \cdot (\frac{2^{-\lambda}}{N})^{\frac{1}{x}} \tag{5}$$

holds, where λ is the security parameter and N the number of leaf nodes.

Proof. By a balls-and-bins argument, we are in a scenario where we insert uniformly at random $\alpha \cdot N$ balls into N bins. The i^{th} bin overflows if there are x balls from $\alpha \cdot N$ that went to the same i^{th} bin. The possible number of combinations equals $\binom{\alpha \cdot N}{x}$. By applying the upper bound inequality to the probability of the union of events (possible combinations), we obtain

$$\begin{aligned}\Pr(B_i \geq x) &\leq \binom{\alpha \cdot N}{x} \cdot \frac{1}{N^x} \\ &\leq (\frac{e \cdot \alpha \cdot N}{x})^x \cdot \frac{1}{N^x} \\ &= (\frac{e \cdot \alpha}{x})^x.\end{aligned}$$

Computing the union bound over all leaf nodes results in

$$\Pr(\exists i \in [N] : \ B_i \geq x) \leq N \cdot (\frac{e \cdot \alpha}{x})^x.$$

In order to have overflow probability equal $2^{-\lambda}$ as previous work, we must verify that $N \cdot (\frac{e \cdot \alpha}{x})^x = 2^{-\lambda}$ which is equivalent to $\alpha = \frac{x}{e} \cdot (\frac{2^{-\lambda}}{N})^{\frac{1}{x}}$. □

Corollary 41 *Let k denote the size of the interior node. The best communication complexity for the 2^{nd} strategy is achieved iff the leaf bucket size x equals*

$$x = \frac{\frac{k}{\ln 2} + \sqrt{k - 4 \cdot k \cdot \log \frac{2^{-\lambda}}{N}}}{2}$$

Proof. First, note that if N leaf nodes can handle $\alpha \cdot N$ elements, the tree is flatter compared to the naïve solution where the tree will have height $\log N$ instead of $\log \alpha \cdot N$. However, the downside of the 2^{nd} strategy is the leaf bucket size increase. In order to take the maximal advantage of this height reduction, we define the optimal leaf buck size x that can have the best communication complexity compared to the naïve solution. Let C_1 and C_2 denote, respectively, the communication complexity needed to download one path for the first and second strategy. For an interior node with size k and a leaf bucket for the naïve strategy with size y, the communication complexities C_1 and C_2 compute to

$$C_1 = (\log \alpha \cdot N - 1) \cdot k + y \text{ and } C_2 = (\log N - 1) \cdot k + x.$$

The best value of x for a fixed value of y, k and λ is the maximum value of the function f defined as

$$f(x) = C_1 - C_2 = y - x + k \cdot \log \alpha.$$

The first derivative of f is $\frac{df}{dx}(x) = x^2 - \frac{k}{\ln(2)} \cdot x + k \cdot \log \frac{2^{-\lambda}}{N}$. This quadratic equation has only one valid solution for a non-negative leaf buckets size and $2^{\lambda} >> N$. The only valid root for the first derivative is $x = \frac{\frac{k}{\ln 2} + \sqrt{k - 4 \cdot k \cdot \log \frac{2^{-\lambda}}{N}}}{2}$. □

Once we have computed the optimal leaf node size, we can plug the result into Eq. **??** to compute the optimal value α. For example, for $N = 2^{30}$ leaves, the size of the leaf bucket in the naïve strategy is $y = 28$, the size of the interior node $k = 36$. Applying the result of Corollary **??** outputs the size of the leaf bucket for an optimal communication complexity which is equal to $x \approx 85$. Applying the result of Lemma **??**, we obtain $\alpha \approx 15$. The communication complexity saving compared to the naïve strategy is around 7% while the storage savings is a significant 87%.

One disadvantage of the 2^{nd} strategy is the possibility of storage underutilization. Imagine the client stores $\alpha \cdot N$ elements in the ORAM tree. When adding a new element, it will trigger the creation of a new leaf level, which is a waste of storage. For example, the client can have $\alpha \cdot N + 1$ elements in his ORAM tree, then performs a loop which respectively adds and deletes two elements. This loop will imply the allocation of an unused large amount of storage (in $O(N)$). Also, this loop implies leaf node pruning which is more expensive (in term of communication complexity) compared to leaf increasing as we will see in Section **??**.

4.4 3rd strategy: dynamic expansion

Our dynamic solution tackles the underutilization of storage described in the previous section. Instead of adding entire new levels to the tree, we will progressively add pairs of leaf nodes to gradually increase the capacity of the tree. This has the advantage of matching a user's storage cost expectation: every time the ORAM capacity is increased, storage requirements increase proportionally. However, unlike our previous techniques, we are now no longer guaranteed to have a full binary tree. This implies a overflow probability recalculation of two different levels of leaf nodes.

Let us assume that we start with a full binary tree containing $N = 2^{\mathrm{l}}$ leaf nodes. Dynamic insertion results in the creation of two different levels of leaves. The first one is on the l^{th} level while the other one in on the $(l+1)^{\mathrm{th}}$ level. In general, after adding $\eta \cdot \alpha$ elements, the number of leaves in the l^{th} level is equal to $N - \eta$ while the number of leaves in the $(l+1)^{\mathrm{th}}$ level is equal to 2η.

At this point, we must carefully consider how to tag new elements that are added to the tree. If we choose tags following a uniform distribution over all the $N - \eta + 2 \cdot \eta = N + \eta$ leaves, we will violate ORAM security. An adversary will be able to distinguish with non-negligible advantage between two elements added before and after increasing the number of leaf nodes in the ORAM, as the assignment probabilities to (leaf) nodes will be different at varying points in the tree's lifecycle.

An efficient solution to this problem is to keep the probability assignment of leaf nodes equally likely for all subtrees with a common root. We implement this approach by setting a leaf's assignment probability in the l^{th} level to $\frac{1}{2^{\mathrm{l}}}$ and to $\frac{1}{2^{\mathrm{l}+1}}$ in the $(l+1)^{\mathrm{th}}$ level. We now analyze the size of leaf buckets with an overflow probability of $2^{-\lambda}$. We consider the general case where we add $\eta < N$ leaf nodes to the ORAM.

Lemma 42 *Let B_i denote the random variable describing the size of the i^{th} leaf node, $1 \le i \le N + \eta$. For the 3^{rd} strategy and a bucket of size B_i, the overflow probability computes to*

$$\Pr(\exists i \in [N+\eta] : \; B_i \ge k) \le \frac{2 \cdot N}{k+1} \cdot (\frac{2 \cdot e \cdot \alpha}{k})^k.$$

Proof. After adding η leaf nodes to the structure, the ORAM contains $N + \eta$ leaves. The probability that at least one leaf node has size larger than k is

$$\begin{aligned}\Pr(\exists i \in [N+\eta] : \; B_i \ge k) &= \Pr(\bigcup_{i=1}^{N+\eta} B_i \ge k) \\ &\le \sum_{i=1}^{2\cdot\eta} \Pr(B_i \ge k) + \sum_{i=2\cdot\eta+1}^{N+\eta} \Pr(B_i \ge k) \qquad (6)\end{aligned}$$

Note that the leaf nodes ranging from 1 to $2 \cdot \eta$ are in the $(l+1)^{\mathrm{th}}$ level with an assignment probability equal to $\frac{1}{2 \cdot N}$ while leaves ranging from $2 \cdot \eta + 1$ to $N + \eta$ belongs to the upper level and have an assignment probability equal to $\frac{1}{N}$. We obtain

$$\text{for } 1 \le i \le 2 \cdot \eta : \Pr(B_i \ge k) \le \binom{\alpha \cdot (N+\eta)}{k} \cdot (\frac{1}{2 \cdot N})^{\mathrm{k}}$$
$$\text{for } 2 \cdot \eta + 1 \le i \le N + \eta : \Pr(B_i \ge k) \le \binom{\alpha \cdot (N+\eta)}{k} \cdot (\frac{1}{N})^{\mathrm{k}}.$$

Note that $\alpha \cdot (N+\eta)$ is the current number of elements in the ORAM. We plug both inequalities in to Eq. **??** and get

$$\Pr(\exists i \in [N+\eta]:\ B_i \geq k) \leq 2\cdot\eta\cdot\binom{\alpha\cdot(N+\eta)}{k}\cdot(\frac{1}{2\cdot N})^{\mathrm{k}} + (N-\eta)\cdot\binom{\alpha\cdot(N+\eta)}{k}\cdot(\frac{1}{N})^{\mathrm{k}}$$
$$\leq (\frac{2\cdot\eta}{2^k}+N-\eta)\cdot(1+\frac{\eta}{N})^k\cdot(\frac{e\cdot\alpha}{k})^k.$$

The bound above is depending on η. Thus, we now compute the value of $\eta < N$ maximizing the bound. This leads us to the function $g(\eta) = (\frac{2\cdot\eta}{2^k}+N-\eta)\cdot(1+\frac{\eta}{N})^k$. Function g has a local maximum value for any $\eta, 1 \leq \eta \leq N$ such that $\eta_{max} = \frac{N}{A}\cdot\frac{k-A}{A(k+1)}$ where $A = 1-\frac{1}{2^{k-1}}$. We replace η_{max} in g to get an upper bound for any any η and $k \geq 2$,

$$\begin{aligned}\Pr(\exists i \in [N+n]:\ B_i \geq k) &\leq g(n_{max})\cdot(\frac{e\cdot\alpha}{k})^k\\ &\leq N\cdot\frac{A+1}{k+1}\cdot(\frac{k(A+1)}{A(k+1)})^k\cdot(\frac{e\cdot\alpha}{k})^k\\ &\leq \frac{2\cdot N}{k+1}\cdot(\frac{2\cdot e\cdot\alpha}{k})^k.\end{aligned}$$

As $k \geq 2$, we conclude with $(\frac{k(A+1)}{A(k+1)})^k \leq 2^k$ and $\frac{A+1}{k+1} \leq \frac{2}{k+1}$. $\square$

So, the overflow probability decreases exponentially when increasing bucket size k. Note that, in the proof, we have maximized the overflow probability independently of the number of nodes added (which is a function of η). In practice, k could be smaller for some intervals of insertions, but we have chosen a maximal value to avoid issues related to changing the leaves' size during insertions.

4.5 Comparison of Strategies

We present a comparison between our three strategies in terms of storage complexity (Figure **??**) and communication complexity per access (Figure **??**). We perform our comparison on a block level, thereby remaining independent of the actual block size.
Communication complexity: the 2nd strategy offers best communication complexity. This is due to shorter paths, a result of flatter trees – compared to the naïve 1st solution. Also, compared to the 3rd strategy, the leaf buckets have smaller size. For a number of elements $N = 2^{30}$ and 2^{-64} overflow probability, the interior node size equals 36 which is appropriate for all three strategies. The difference consists on the size of the leaf buckets as well as the height of the resulting tree. The bucket size for the naïve (1st), lazy (2nd) and dynamic (3rd) strategy respectively equals 28, 85 and 130 blocks. The tree's height for the naïve solution equals 30 while for the lazy and dynamic solution the tree height is 26 since $\alpha \approx 2^4$. In Figure **??**, for an eviction rate used equals 4, the entire communication complexity (upload/download) on the main ORAM respectively equals 26928, 24210 and 25020 blocks for the naïve, lazy and dynamic solution. Note that

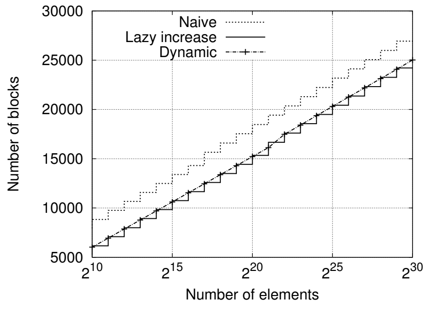

Fig. 1: Communication, blocks per access

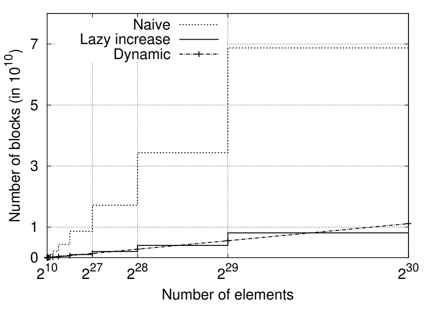

Fig. 2: Storage cost, blocks

per access, we save around 7% in communication cost. Recall that our main purpose is to reduce the storage overhead while maintaining the same communication complexity. However, our results show that storage optimization has a direct consequence on reducing the communication complexity as well.
Storage complexity: there is no "clear winner". Depending on the client's usage strategy, the dynamic (3^{rd}) strategy can be considered best, as it provides more intuitive and fine grained control over storage size. However, if the insertion of elements follows a well defined pattern where the client is always expanding their capacity by a factor of α, the 2^{nd} strategy will result in cheaper cost. The cost reduction is significant, around 87% fewer blocks compared to the naïve solutions.

Independently of the blocks size, this represents 87% of storage cost savings. Consider the following example: we fix the block size to 4096 Byte and the number of elements to $N = 2^{30}$, resulting in a dataset size equal to 4 TByte. Based on Amazon S3 pricing [**?**] where the price is equal to 0.029 USD per GByte per month, the client has to store, for the naïve solution, $\sim 2.8.10^{14} \approx 262$ TByte, implying ~ 7600 (USD) per month. With the lazy solution, the client has to store only ~ 31 TBytes, which is only 900 (USD) per month (almost 10 times cheaper than the naïve solution).

In general, both the 2^{nd} and 3^{rd} strategies outperform the naïve one in terms of communication and storage complexities.

4.6 Position Map

To maintain constant client memory, it is important to recursively store the mapping between tags and elements in a position map on the server. This position map is stored in a logarithm number of ORAMs with a number of leaves increasing exponentially from one ORAM to the other. With a position map factor τ, $N = \tau^l$, the position map is composed of $l-1$ small ORAMs where ORAM$_i$ has a number of leaves equal to τ^i, $1 \leq i \leq l-1$.

Surprisingly, resizing the position map is trivial, e.g., following one of the two subsequent strategies: (1) use the same strategy of resizing (adding/pruning) that we apply on ORAM$_{l-1}$, or (2) create a new level of recursion in the case of adding, or deleting the last level of recursion in the case of pruning. Assume N elements; each element is associated to a leaf tag that has size $\log N$ bits. We describe each solution for the case of the naïve adding strategy.

(1) When we add a new line to the main ORAM (ORAM$_l$), we have $2 \cdot N$ leaves instead of N leaves. Similarly, we increase the size of the last ORAM of the position map (ORAM$_{l-1}$) to have a new level of leaves. The only issue with this solution is that we should increase the block size. Instead of having $O(\tau \cdot \log N)$ bits, it will have now $O(\tau^2 \cdot \log N)$ bits. Every time an element is accessed, the corresponding block is modified to have the new size. Note that when we add a new level of leaves, we can always access all elements of the ORAM using the previous mapping. For this, we just append at the end of the tag fetched an additional bit 0 or 1 to access a random child (to stay oblivious and access the entire path). After accessing any "old" elements (old denotes elements with a previous mapping), the mapping is updated to have $\log N + 1$ bits instead of $\log N$.

(2) The second solution is straightforward and based on creating a new level of recursion when a new level of leaves is created. Note that blocks in this level will have $O(\tau \cdot \log N + 1)$ bits instead $O(\tau \cdot \log N)$. To access an "old" element, we use the same method described above.

5 Pruning

Assume an ORAM storing N elements. Now, the client deletes η elements from the ORAM. Consequently, the naïve ORAM construction now contains $N - \eta$ elements, but still has N leaves. Consequently, the client tries saving unnecessary storage costs and frees a number of nodes from the ORAM. Similar to adding element to the ORAM tree, we tackle pruning by presenting two different strategies. The first one, a *lazy pruning*, prunes the entire set of leaves of the lowest level l and merges content with level $l - 1$. Our second strategy consists of a *dynamic pruning* that deletes two leaf nodes for a specific number of elements removed from the ORAM. Again, we will analyze overflow probabilities induced by such pruning as well as complexities.

5.1 Lazy pruning

In Section **??**, we have demonstrated that leaves can store significantly more elements while only slightly increasing their size. We will use this observation to construct a new algorithm for lazy pruning. Assume that the leaf level contains N leaves for $\alpha \cdot N$ elements stored. Let η denote the number of elements deleted by the client. For sake of simplicity, assume that, at the beginning, we have $\eta = 0$ and N leaf nodes. Our pruning technique is similar to the "lazy" insertion described previously. Whenever $\alpha \cdot \frac{N}{2} < \eta \leq \alpha \cdot N$, we keep the same number of leaves. Within this interval, the client can add or delete elements without applying any change to the structure, as long as the number of elements remains within the defined interval. If the number of deletion equals $\alpha \cdot \frac{N}{2}$, the client proceeds to remove an entire level of leaf nodes. The client proceeds to read every leaf node, along with its sibling, and merges them with their parent node. While this appears to be straightforward, an oblivious merging of siblings into their parent is more complex under our constant-client memory constraint. We will discuss this in great detail below.

Besides, the major problem of this technique is its unfortunate behavior in case of a pattern oscillating around the pruning value. For example, the if the client deletes $\alpha \cdot \frac{N}{2}$ elements, prunes the entire level, then adds a new element back. Now the ORAM structure has more than $\alpha \cdot \frac{N}{2}$ elements in $\frac{N}{2}$ leaves, so the client has to again double the number of leaves. This pattern will result in high communication costs.

5.2 Dynamic pruning

Given that pruning an entire level at once is very inefficient, we now investigate how pruning can be done in a more gradual way. For every α elements we delete, we will prune two children and merge their contents into their parent node. The pruning will *fail* if the number of elements in both children and parent is more than k. This can only occur if there are more than k elements associated (tagged) to these children. The following lemma states the upper bound of the overflow probability for the parent node after a merging. Recall that we begin with a full binary tree of N leaves and $\alpha \cdot N$ elements. Assume that we have already deleted $\alpha(\eta - 1)$ elements, and we want to delete an additional α elements.

Lemma 51 *Let P_η denote the random variable of the size of the η^{th} parent node. For dynamic pruning, the probability that pruning will fail equals*

$$\Pr(P_\eta > k) \leq (\frac{2e \cdot \alpha}{k})^k$$

Proof. The pruning will fail *iff* there are more than a total of k elements in the parent and the children. Any element in these three buckets must be tagged for either the left or the right child. In order to compute the overflow probability of the parent, we compute the probability that more than k elements are tagged to both children.

$$\begin{aligned}\Pr(P_\eta > k) &= \binom{\alpha \cdot (N-\eta)}{k} \cdot (\frac{2}{N})^k \\ &\leq (\frac{e \cdot \alpha \cdot (N-\eta)}{k})^k \cdot (\frac{2}{N})^k \\ &\leq (1 - \frac{\eta}{N})^k \cdot (\frac{2e \cdot \alpha}{k})^k \\ &\leq (\frac{2e \cdot \alpha}{k})^k\end{aligned}$$

In conclusion, the probability decreases exponentially with bucket size k. The upper bound is independent of the number of pruned nodes η. In practice, the bounds are tighter, especially for larger values of η.

```
Input: Configuration of buckets $A$ and $B$
Output: A permutation which randomly "lines up" bucket $B$ to bucket $A$
// Slots in $A$ and $B$ start either empty or full; mark slots
   in $A$ as ``assigned'' if block from $B$ is assigned in $\pi$
$x \leftarrow$ number of empty slots in $A$ ;
$y \leftarrow$ number of full slots in $B$ ;
$d \leftarrow x - y$ ;
for $i$ from 1 to $k$ do
    if $B[i]$ is full then
        $z \overset{\$}{\leftarrow}$ all empty slots in $A$;
    else
        if $d > 0$ then
            $z \overset{\$}{\leftarrow}$ all non-assigned slots in $A$;
            $d \leftarrow d - 1$;
        else
            $z \overset{\$}{\leftarrow}$ all full slots in $A$;
        end
    end
    $\pi[i] \leftarrow z$ ;
    $A[z] \leftarrow$ assigned ;
end
return $\pi$ ;
```

Algorithm 1: GeneratePermutation(A, B)

Complexity of oblivious merging The cost of dynamic pruning boils down to the cost of obliviously merging three buckets of size k. We can achieve this with $O(k)$ communication and constant memory complexity. First, note that we do not have to merge all three buckets at once. All that is required is an algorithm which obliviously merges two buckets. We can then apply it to successively merge three buckets into one. Since the adversary already knows that the two buckets being merged have no more than k elements in them (as shown above), the idea will be to retrieve the elements from each bucket in a more efficient way that takes advantage of this property.

In Algorithm **??**, the client randomly permutes the order of the elements in one bucket, subject to the constraint that, for all indices, at most one of the elements between both buckets is real. That is, the permutation "lines up" the two buckets so that they can be merged efficiently. Special care must be given to generate this permutation using only constant memory. The client makes use of "configuration maps" which simply indicate, for every slot in a bucket, whether that slot is currently full or empty. These maps can be stored encrypted on the server and take up $O(1)$ space each in terms of blocks (because the buckets contain $O(\log N)$ elements and a single block is at least $\log N$ bits [**? ?**]). Then, the client iterates through the slots in one bucket, randomly pairing them with compatible slots in the other (i.e., a full slot cannot be lined up with another full slot). An additional twist is that an empty slot can be lined up with either a full or empty slot in the other bucket, but not at the expense of "using up" an empty slot that might be needed later since we cannot match full with full. Therefore, we have to also keep a counter of the difference between empty slots in the target bucket and full slots in the source bucket.

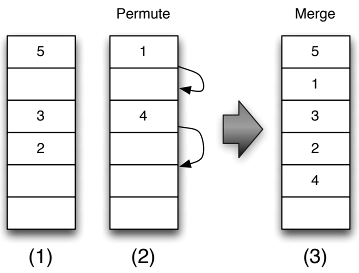

Fig. 3: Illustration of permute-and-merge process. Bucket (2) is permuted and then merged with bucket (1) to create a new, combined bucket (3).

As seen in Figure **??**, once the client generates the permutation, they can retrieves the elements pairwise from both buckets (i.e., slot i from one bucket and the slot which is mapped to i via the permutation from the other bucket), writing back the single real one to the merged bucket.

It remains to show that this permutation does not reveal any information to the adversary. If it was a completely random permutation, it would certainly contain no information. However, we are choosing from a reduced set: all permutations which cause the bucket to "line up" with its sibling.

Fortunately, we can formally prove that our permutation does not reveal any information beyond what the adversary already knows. This is because there are no permutations which are inherently "special" and are more likely to occur, over all possible initial configurations of the bucket. For every permutation and load of a bucket, there are an equal number of bucket configurations (i.e., which slots contain real elements and which do not) for which that permutation is valid.

To make this approach work, we need to slightly modify the behavior of the bucket ORAMs. Previously, when a new element was added to a bucket, it did not matter which slot it went into in that bucket. It was possible, for instance, that all the real elements would be kept at the top of the bucket and, when adding a new one, the client would simply insert that element into the first empty slot that it could find. However, to use this permutation method equalwe require that the buckets be in a random "configuration" in terms of which slots are empty and which are filled. Therefore, when inserting an element, the client should choose randomly amongst the free slots. Again, this is possible with constant client memory using our configuration maps. With this behavior, applying the above logic leads to the conclusion that the adversary learns nothing about the load of the bucket from seeing the permutation.

Refer to Appendix **??** for the full security proof.

5.3 Privacy analysis

Theorem 51 *Resizable ORAM is a secure ORAM following Definition* **??***, if every node is a secure trivial ORAM.*

Proof (Sketch). Given that ORAM buckets are secure trivial ORAMs, we have to show that two access patterns induced by $\overrightarrow{y}$ and $\overrightarrow{z}$ of the same length are indistinguishable.

Compared to traditional ORAM, resizable ORAM includes two new operations, Alloc and Free. Note that those operations should be in the same positions for both sequences, otherwise, distinguishing between the access pattern will be straightforward. Furthermore, we have already shown that, for increasing the size of the ORAM, Alloc for the 2^{nd} and 3^{rd} strategies will not induce any leakage. Also, lazy or dynamic pruning strategies will not leak any information about the load of the buckets. That is, the Free operation is oblivious. So, these additional operations do not leak any other information besides the actual number of elements (or a window that bounds the current number of elements for strategies 1 and 2). Also, the access patterns induced by other operations in both sequences $\overrightarrow{y}$ and $\overrightarrow{z}$ are indistinguishable (see the proof by **?**]). We can conclude that resizable ORAM is a secure ORAM following Definition **??**. □

6 Related Work

We are the first to rigorously investigate the topic of resizing current tree-based ORAMs [**? ? ? ? ?**] and tackle the challenges that can arise from resizing these ORAMs. Our work especially focuses on tree-based ORAM constructions [**? ? ? ? ?**] for the *constant client memory* setting.

Oblivious RAM was introduced by **?**]. Much work [**? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ?**] has been published to reduce the communication complexity between client and storage. Early schemes were able to optimize *amortized* cost to be poly-logarithmic, but still maintained linear worst-case cost [**? ? ? ?**], due to the fact that they all eventually require an expensive reshuffling. Yet, resizing these types of ORAM is straightforward. Adjusting the size can be done at the same time as reshuffling, for no cost. The only leakage in this case will be the information about the total number of elements stored in the ORAM.

Avoiding the expensive reshuffling, **?**] presented the first tree-based construction that involves *partial* reshuffling of the ORAM structure for every access. Thus, the amortized cost equals the worst-case cost with communication complexity of $O(\log^3 N)$ blocks. An additional advantage of this construction is its constant client memory requirement (in term of blocks). Constant client memory ORAM constructions are especially attractive in scenarios with, for example, embedded devices or otherwise constrained hardware.

Further results show that you can improve communication complexity if poly-logarithmic client memory is acceptable [**? ? ?**]. **?**] optimize **?**] by introducing a k-ary structure with a new deterministic eviction algorithm. This results in $O(\frac{\log^3 N}{\log\log N})$ for a branching factor equal to $O(\log N)$, but the client must have $O(\log^2 N)$ client memory available. Inspired by [**?**], for a client memory equal to $\Theta(\log N)$, **?**] presented Path ORAM, a construction with communication complexity in $O(\log^2 N)$. A subsequent work by **?**] reduces communication complexity by a factor of 2 by reducing the size of the buckets. We leave the problem of resizing these non-constant memory ORAMs to future work.

7 Conclusion

We are the first to show how to dynamically resize constant-client memory tree-based Oblivious RAM. This allows for use cases where clients do not know in advance exactly how much storage they will need and/or wishes to scale their storage needs efficiently and cheaply. We have shown that the naïve solution of adding leaf nodes induces a significant, unnecessary overhead. Instead, more advanced strategies, lazy insertion and dynamic insertion, can save dramatically on communication and storage cost compared to the naïve solution, although neither strategy is clearly superior to the other. Furthermore, we have demonstrate that the size of a tree-based ORAM can be decreased efficiently using an oblivious pruning technique. Throughout the paper, we have rigorously analyzed the overflow probability for each technique and presented a tight analysis of both interior and leaf node sizes.

Acknowledgments: This work was partially supported by NSF grant 1218197.

A Proof: Oblivious Permute-and-Merge

Lemma A1 *Given two buckets with maximum size k and load m and n respectively, over the random configurations of those buckets, Algorithm* **??** *will output a uniformly random permutation which is independent of m and n.*

Proof. We can determine the probability of a particular permutation π being chosen, given m and n, with a counting argument. It will be equal to

$$\frac{\text{\# of configurations for which } \pi \text{ is a valid permutation}}{\text{total \# of configurations } \times \text{ \# of valid permutations for a given configuration}}$$

The number of configurations for which π is a valid permutation depends on m and n, but not on π itself. This can be seen if you consider that applying the permutation to a fixed configuration of the bucket simply creates another, equally likely configuration. The number of configurations for the sibling bucket that will "match" with that bucket are exactly the same no matter what the actual configuration of the first bucket is. Knowing this, combined with the fact that the probabilities must sum to one, tells us immediately that every permutation is equally likely. However, we can continue and express the total quantity for our first expression as

$$\binom{k}{m}\binom{k-m}{n}$$

This can be thought of as choosing the m full slots for one bucket freely and then choosing the n full slots in the second bucket to line up with the free slots in the already chosen first bucket. The number of valid permutations per configuration can equally be determined via a counting argument as

$$\binom{k-m}{n} \cdot (k-n)! \cdot n!$$

That is, choosing free slots for the n elements in the second bucket and then all permutations of those elements times the permutations of the free blocks. That gives us a final expression for the probability of choosing permutation π of

$$\frac{\binom{k}{m}\binom{k-m}{n}}{\binom{k}{m}\binom{k}{n}\binom{k-m}{n} \cdot (k-n)! \cdot n!} \tag{7}$$

With some algebraic computations, we can show that the Eq. **??** can be simplified to $\frac{1}{k!}$. That is, this shows that the number of permutations, for any random distribution of load in a bucket, is independent of the current load. Again, since this does not depend on π (but only on the size of the bucket), every permutation must be equally likely over the random configurations of the buckets. □

Corollary A1 *A permutation π chosen by Algorithm* **??** *gives no information about the load of the buckets being merged.*

Proof. By our above lemma, independent of the load each permutation is chosen uniformly over the configurations of the two buckets. Therefore the permutation cannot reveal any information about the load. □