Resizable Tree-Based Oblivious RAM

Tarik Moataz[1,*,§], Travis Mayberry[2,§], and Erik-Oliver Blass[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|blass}@ccs.neu.edu`

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. This capability 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 Shi et al. [16] and propose several ways to support both increasing and decreasing the ORAM's size with sublinear communication. We show that increasing 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 the 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 Goldreich [6]. 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.

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

[*]Work done while at Northeastern University.

[§]Both authors are first authors.

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 Shi et al. [16], Stefanov et al. [17], 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 study 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 Shi et al. [16] ORAM), and 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 Shi et al. [16] ORAM.

Our contributions in this paper are as follows:

1. Three 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 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 Shi et al. [16] 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 Shi et al. [16], focusing on the relevant details which are necessary to understand our resizing techniques.

2.1 Preliminaries

Recall that an Oblivious RAM is a cryptographic data structure which stores 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 [16] is stored as a binary tree with N leaves. Each node in the tree represents a smaller ORAM *bucket* [7] 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 [16]. 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

Shi et al. [16]'s ORAM implements Read and Write operations by applying, first, a $\mathsf{ReadAndRemove}(a)$ operation, followed by an $\mathsf{Add}(d,a)$. The idea is that $\mathsf{ReadAndRemove}(a)$ 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 because all he will see is fresh encryptions replacing every block in the path. For the Add operation, the client uniformly at random chooses a new tag $t \xleftarrow{\$} \{0,N-1\}$ 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, modifying only the appropriate one. Again, this is done in a step-by-step 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. In [16], the analysis done in the original paper differentiates between the interior and the leaf node, while for resizing we will a general analysis that considers both cases at once.
2. For $n > N$ elements, we must determine the most effective strategy of increasing the number of nodes that enables the best storage usage and communication for the client.
3. Reducing the size of the tree is non-trivial, especially when hoping to perform it with low communication complexity and constant client memory. Careful consideration must be taken to ensure that elements can be moved from the pruned nodes into other buckets in an oblivious way which also maintains overflow probabilities and can be done efficiently.

3.2 Resizing Operations

Besides having Read and Write operations, we introduce two new basic operations that represents the mechanisms 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 holds one fewer elements.

3.3 Security Definition

Resizing an ORAM should not leak any information besides the maximum number of elements. Thus, resizable ORAM has to verify the obliviousness requirement taking into account the resizing operations.

Definition 31 *Let* $\overrightarrow{y} = \{(op_1,d_1,a_1),(op_2,d_2,a_2),...,(op_M,d_M,a_M)\}$ *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* $\overrightarrow{y}$*. We say that resizable ORAM is secure* iff*, for any PPT adversary* $\mathcal{D}$ *and any two same-length sequences* $\overrightarrow{y}$ *and* $\overrightarrow{z}$ *with Alloc and Free at the same positions such that* $\forall i \in [M]$ *verifying* $\overrightarrow{y}(i) = \mathsf{Alloc}, \overrightarrow{z}(i) = \mathsf{Alloc}$ *(the same applies for* Free*),*

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

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 [7], all blocks are IND-CPA encrypted. Also, whenever a block is accessed by any type of operation, its bucket is re-encrypted.

4 Adding

We begin by describing a "naive" solution that will add a new level of leaves when $n > N$. This leads to a problem, however: 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 need it. However, in both the naive and second solution, there are thresholds which cause huge jumps in storage space. This can be counterintuitive for users, so we propose a third solution dubbed "dynamic expansion". This strategy progressively adds leaf nodes to gradually increase capacity of the tree. This case is interesting because it results in an unbalanced tree, which requires careful analysis to ensure that the overall failure probability of the ORAM does not exceed the parameterized threshold.

4.1 Tightening the bounds

Since communication and storage complexities represent the core comparative factor between strategies, and both are dependent primarily on the bounds we can establish for sizes of the buckets, it is important to get a tight analysis for both interior and leaf buckets in each situation. The bounds given in [16] are substantially larger than necessary, as has been shown in other work [5]. Therefore, as a first contribution, we give new, tighter bounds for these interior and leaf nodes.

Interior Nodes We will first tackle the size of the interior nodes, which is governed by a queuing theory analysis. Let I_i denote the random variable for the size of interior nodes of the i^{th} level of the tree. For eviction rate ν, we can compute the probability of a bucket on levels $i > \log\nu$ having a load of at least k (i.e. the bucket has overflown) to:

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

In [16], 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. 1 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 rate $\nu > 4$ 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 can be computed to:

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

In particular for $\nu=4$, which is 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, per level, only one child can overflow. For the Eq. 3, an equality still holds since the buckets can be considered independent in the steady state[11].

Given a 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 $k=-\log(1-(1-2^{-\lambda})^{\frac{1}{\nu\cdot\log N}})$.

For example, to have an overflow probability equal to: 2^{-64}, $\lambda=64$, $N=2^{30}$, $\nu=4$, the bucket size needs only to be 36, while in [16] the bucket size has to be equal to 72 to achieve 2^{-64} 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:

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

Note that in Eq. 4, we have used the union bound inequality. Based on the same parameters as in the previous example, the size of the leaf node has to be set only to 28 to have an overflow probability equal to 2^{-64}. To compute this result, one should solve 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 the leaf node should be carefully chosen depending on N. Every time the number of elements increases by a multiplicative factor of 2^5, based on the overflow upper bound, 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 related work, the number of operations is typically considered as part of security parameter λ. The larger the number of operations performed, the larger the security parameter has to be.

4.2 1st Strategy: naive expansion

Let N and n respectively denote the number of leaf nodes and elements in the ORAM. The naive solution is simply adding a new leaf level, as soon as the condition $n > N$ occurs. The main drawback of this first naive 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 then 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 ORAM. For N leaves ORAM, the client is allowed to store up to $\alpha \cdot N$ new elements without increasing the size of the ORAM. If $n > \alpha \cdot N$, 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 $2N$, then the client will work with the same structure as long as $\alpha \cdot N < n \leq \alpha \cdot 2N$. Once $n > \alpha \cdot 2N$, a new level of leaves containing now $4N$ leaf buckets is created. To be able to store more elements, we should increase the leaf bucket size, so we can keep the same overflow probability. There is a balance to consider between increasing the size of the leaf node 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 normal ORAM tree with the same number of elements, then this technique would not be worth applying.

Gentry et al. [5] showed 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 in [5] α has been chosen to optimize the storage cost for a given overflow probability, we target instead the computation of the value α for the optimal communication complexity. Note that in our subsequent analysis, the bounds for interior and leaf nodes computed in Section 4.1 will be used.

We are first interested on finding a relation between the size of the leaf bucket x and the factor α for our 2nd 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. For calculating the overflow, we focus on the worst case which occurs when there are $\alpha\cdot N$ elements in a structure that has N leaves.

Lemma 41 *Let us denote by x the optimal size bucket for the* 2nd *strategy. Then, we can show that:*

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

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

Proof. By analogy to the balls and bins game, 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:

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

By union bound over all the leaf nodes:

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

In order to have the same overflow probability equal to $2^{-\lambda}$, we should 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 z denote the size of the interior node. The best communication complexity for the* 2nd *strategy is acquired iff the leaf bucket size x is equal to:*

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

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

$$C_1 = (\log \alpha \cdot N - 1) \cdot z + y \; and \; C_2 = (\log N - 1) \cdot z + x.$$

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

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

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

Once we have computed the optimal leaf node size, we can plug the result into Eq. 5 to compute the optimal value α. For example, for $N = 2^{30}$ leaves, the size of the leaf bucket in the naive strategy is $y = 28$, the size of the interior node $z = 36$. Applying the result of the Corollary 41 outputs the size of the leaf bucket for an optimal communication complexity which is equal to $x \approx 85$. Applying the result of Lemma 41, we obtain that $\alpha \approx 15$. The communication complexity saving compared to the naive strategy is around 7% while the storage savings is 87%.

One disadvantage of the 2^{nd} strategy is the possibility of memory underutilization. Imagine the client has $\alpha \cdot N$ elements in the ORAM, 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 structure, then performs a loop which respectively adds and deletes two elements. This loop will imply the allocation of an unused large amount of memory (in $O(N)$). Also, this loop implies leaf node pruning which is more expensive (in term of communication complexity) compared to leaf increasing (see section 5).

4.4 3^{rd} strategy: dynamic expansion

The dynamic solution tackles the underutilization of memory 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 expectation: every time capacity is increased, storage requirements increase proportionally. Unlike previous techniques, we are no longer guaranteed to have a full binary tree, which means that we must calculate the overflow probabilities of two different levels of leaf nodes.

Let us assume that we start with a full binary tree containing $N = 2^{\text{l}}$ leaf nodes. Dynamic insertion results on the creation of two different level of leaves. The first one belongs to the l^{th} level while the other one to the $(l+1)^{\text{th}}$ 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 $(l+1)^{\text{th}}$ level is equal to 2η.

At this point, we must consider how to tag new elements that are added to the tree. If we choose a uniform distribution over all the leaves, which is in this case equal to $\frac{1}{N+\eta}$, an adversary will be able to distinguish with non-negligible advantage between two elements respectively added before and after increasing the number of leaf nodes in the ORAM, because the assignment probabilities will be substantially different at varying points in the tree's lifecycle.

A direct and efficient solution to this security issue will be to keep the probability assignment of leaf nodes equally likely for any subtrees with common root. We can satisfy this constraint by setting the leaves' assignment probability in the l^{th} level to $\frac{1}{2^{l}}$ while leaves in the $(l+1)^{\text{th}}$ level to $\frac{1}{2^{l+1}}$. In the following, we are interested in studying the size of the leaf bucket with an overflow probability equal to $2^{-\lambda}$. We study the general case where we add $\eta < N$ leaf nodes to the ORAM where N denotes the initial number of leaves.

Lemma 42 *Let B_i denote the random variable describing the size of the i^{th} leaf node, we show that based on the* 3^{rd} *strategy we compute:*

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

Our proof strategy is similar to what we have done above, but with separate analysis for the two different levels of leaf nodes. We show that, with a very small increase in bucket size, we can guarantee the same overflow probability as before and support this more intuitive dynamic resizing. Refer to the full proof in the appendix B.

4.5 Strategies Comparison

We present a comparison between our three strategies in term of storage complexity (figure 2), as well as in term of communication complexity per access (figure 1). Our comparison is done in term of blocks, so we are independent of the choice of the block size.

For communication complexity, the 2^{nd} solution is the best compared to the others. This is due to a shorter path because the tree is flatter compared to the naive solution. Also, compared to the dynamic one, the leaf buckets have smaller size. In term of storage complexity, there is no clear winner. Depending on the user's usage strategy, the dynamic solution can be considered the best since it provides more intuitive and granular control over storage size. However if the insertion of elements follows a well defined pattern such that the client is always expanding their capacity by a factor of α, then, the 2^{nd} strategy will be cheaper. In general, both the 2^{nd} and 3^{rd} strategies outperform the naive one in term of communication and storage complexities.

5 Pruning

Let us assume that at the beginning the ORAM stores N elements. The client proceeds to the deletion of η elements from the ORAM structure. Consequently, the naive ORAM construction now contains $N-\eta$ elements, but still has N leaves. At this stage, the client aims to save unnecessary memory costs and free a number of nodes from the ORAM. By analogy to the previous section that studies the allocation of new

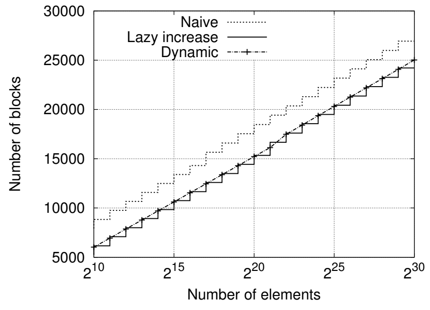

Fig. 1: Comparison of communication in blocks per access

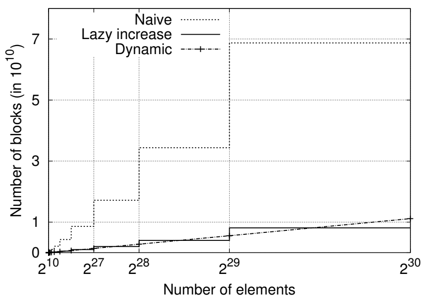

Fig. 2: Comparison of storage cost in blocks

nodes, we will tackle the pruning problem from two perspectives. The first one, a lazy pruning, that prunes the entire leaves of the deepest level and merge them with the upper level. The second solution consists on a dynamic pruning which deletes two leaf nodes for a defined number of elements removed from the ORAM. In the following, we will study the overflow probability induced by such pruning and its overall complexity.

5.1 Lazy pruning

In Section. 4.3, we have demonstrated that the leaves can store more elements while slightly increasing their size. We will use this result to construct our algorithm for lazy pruning. Let us assume that the leaf level contains N leaves for $\alpha \cdot N$ elements stored. Let us denote by η the number of elements deleted by the client. For sake of clarity, let us assume that at the beginning, we have $\eta = 0$ and N leaf nodes. The pruning technique is very similar to the lazy insertion previously described. Whenever we have $\alpha \cdot \frac{N}{2} < \eta \leq \alpha \cdot N$, 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 does not go beyond the defined interval. If the number of deletion is now equal to $\alpha \cdot \frac{N}{2}$, the client proceeds to remove an entire level of leaf nodes. The client proceeds to read every leaf node, along with its sibling, and merge them with their common parent. While it seems straightforward, oblivious merging of siblings into their parent is more complicated under constant-client memory constraint. We depict this issue in more details in the following section.

Besides, the major disadvantage of such technique is the possibility to have a pattern that oscillates around the pruning value. For example, the client deletes the $\alpha \cdot \frac{N}{2}$ elements, he 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, then the client has to again double the number of leaves. This pattern may be devastating in term of communication complexity.

5.2 Dynamic pruning

Given that pruning an entire level at once can be 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, which 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 by a full binary tree that has N leaves and $\alpha \cdot N$ elements. Let us assume that we have already deleted $\alpha(\eta-1)$ elements and we are aiming to delete an additional α elements.

Lemma 51 *Let us denote by P_η the random variable of the size of the η^{th} parent node, we show that based on the dynamic pruning strategy the probability that pruning will fail is equal to:*

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

Proof. The pruning will fail if and only if there are more than k elements between the parent and 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 must simply compute the probability that more than k elements are tagged to both children:

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

Complexity of oblivious merging The cost of dynamic pruning boils down to the cost to obliviously merge three buckets of size k, which are guaranteed to have in total no greater than k real elements, into a single bucket of size k. With some care, we can achieve this with only $O(k)$ communication. First, note that we don't have to merge all three buckets at once. All that is required is an algorithm which obliviously merges two buckets, and we can apply it successively to merge three into one. Since the adversary already knows that the two buckets being merged have no more than k real elements in them together (as we have shown above, this is true with high probability for any two buckets), the idea will be to retrieve the elements from each bucket in a more efficient way that takes advantage of this property.

Using algorithm 1 (see appendix A) 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

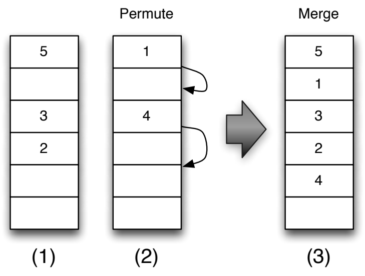

Fig. 3: Illustration of permute-and-merge process. Bucket (2) is permuted and then merged with bucket (1) to create a new, combined bucket (3).

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 they iterate 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.

As seen in figure 3, 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 show that 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 we 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. 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 B for the full proof.

5.3 Privacy analysis

Theorem 51 *Resizable ORAM is a secure ORAM following Definition 31, 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 two same-length $\overrightarrow{y}$ and $\overrightarrow{z}$ are indistinguishable. Compared to the classical ORAM, resizable ORAM includes two new operations, namely, 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 underlined that for increasing the size of the ORAM, Alloc operation for 2nd and 3rd strategies, will not induce any leakage. Also, lazy or dynamic pruning strategies will not leak any information about the load of the buckets based on the result of the proof in appendix B, i.e., Free operation is oblivious vis-a-vis the adversary. Finally, knowing on the one hand that these additional operations do not leak any other information besides the actual number of elements (or a window that frames the current number of elements for strategies 1 and 2), and on the other hand, the access patterns induced by other operations in both sequences $\overrightarrow{y}$ and $\overrightarrow{z}$ are indistinguishable (see the proof in [16]), we can conclude that resizable ORAM is a secure ORAM following the definition 31. $\square$

6 Related Work

As far as we know we are the first to investigate the topic of resizing modern, tree-based ORAMs [4, 5, 13, 16, 17] and tackle all the challenges that can arise from resizing these ORAMs. Our work especially focuses on tree-based ORAM constructions [4, 5, 13, 16, 17] for the *constant client memory* setting.

Oblivious RAM was introduced by Goldreich and Ostrovsky [7]. Much work [1–10, 12–19] 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 [8, 15, 18, 19], 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, Shi et al. [16] 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 we can improve communication complexity if poly-logarithmic client memory is acceptable [4, 5, 17]. Gentry et al. [5] optimize [16] 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 [16], for a client memory equal to $\Theta(\log N)$, Stefanov et al. [17] presented Path ORAM, a construction with communication complexity in $O(\log^2 N)$. A subsequent work by Fletcher et al. [4] 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 have shown in this paper how to dynamically resize constant-client memory tree-based Oblivious RAM. This allows for use cases where the client does not know ahead of time exactly how much storage they will need and/or wishes to scale their storage needs efficiently and cheaply. We have shown that the naive solution of adding leaf nodes when capacity is exceeded induces a considerable unnecessary overhead. Moreover, we have shown that the presented advanced strategies, lazy insertion and dynamic insertion, can save dramatically on communication and storage cost compared to the naive 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 necessary sizes for both interior and leaf nodes.

Bibliography

[1] 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 2011.

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

[3] 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.

[4] 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.

[5] 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.

[6] 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.

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

[8] 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.

[9] 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.

[10] 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.

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

[12] 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.

[13] 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.

[14] 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.

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

[16] 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.

[17] 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.

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

[19] 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.

A Oblivious-Pruning permutation generation

```
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
// We progressively mark slots
   in $A$ as ``assigned'' when a block from $B$ is assigned to it 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)

B Proofs

B.1 Dynamic expansion-proof of lemma

Lemma B1 *Let B_i denote the random variable describing the size of the i^{th} leaf node, we show that based on the* 3^{rd} *strategy we compute:*

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

Proof. After adding η leaf nodes to the structure, the ORAM contains $N+\eta$ leaves. The probability that any leaf node has a size larger than k equals:

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

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

For $1\leq i\leq 2\eta$:

$$\Pr(B_i\geq k)\leq \binom{\alpha\cdot(N+\eta)}{k}\cdot(\frac{1}{2N})^{\mathrm{k}}$$

For $2\eta+1\leq i\leq N+\eta$:

$$\Pr(B_i\geq k)\leq \binom{\alpha\cdot(N+\eta)}{k}\cdot(\frac{1}{N})^{\mathrm{k}}$$

Note that $\alpha\cdot(N+\eta)$ is the current number of elements in the ORAM. We plug both inequalities in Eq. 6:

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

The bound above is depending on η. Thus, we should find the maximum value of $\eta<N$ that maximize the bound, so the resulting upper bound is independent of η. This leads us to study the behavior of the function $g(\eta)=(\frac{2\eta}{2^k}+N-\eta)\cdot(1+\frac{\eta}{N})^k$. The function g has a local maximum value for any η s.t. $1\leq\eta\leq N$ equal to $\eta_{max}=\frac{N}{A}\cdot\frac{k-A}{A(k+1)}$ where $A=1-\frac{1}{2^{k-1}}$. We replace η_{max} in g in order to have an upper bound for any any η s.t. $1\leq\eta\leq N$ and $k\geq 2$:

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

because for $k\geq 2$ we have $(\frac{k(A+1)}{A(k+1)})^k\leq 2^k$ and $\frac{A+1}{k+1}\leq\frac{2}{k+1}$. $\square$

B.2 Oblivious permute-and-merge

Lemma B2 *Given two buckets with maximum size k and load m and n respectively, over the random configurations of those buckets, algorithm 1 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:

$$\frac{\#\text{ of configurations for which }\pi\text{ is a valid permutation}}{\text{total }\#\text{ of configurations }\times\#\text{ of valid permutations for a given configuration}} \tag{7}$$

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

$$\binom{k}{m}\binom{k-m}{n} \tag{8}$$

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

$$\binom{k-m}{n}\cdot(k-n)!\cdot n! \tag{9}$$

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

$$\frac{\binom{k}{m}\binom{k-m}{n}}{\binom{k}{m}\binom{k}{n}\binom{k-m}{n}\cdot(k-n)!\cdot n!} \tag{10}$$

With some algebraic computations, we can show that the Eq. 10 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 B1 *A permutation π chosen by the algorithm 1 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 □