A Simple Recursive Tree Oblivious RAM

Benny Pinkas
Bar Ilan University
Ramat Gan 52900, Israel
benny@pinkas.net

Tzachy Reinman
The Hebrew University of Jerusalem
Jerusalem 91904, Israel
reinman@cs.huji.ac.il

June 3, 2014

Abstract

Oblivious RAM (ORAM) has received increasing attention in the past few years. The goal of oblivious RAM is to enable a client, that can locally store only a small (preferably constant) amount of data, to store remotely N data items, and access them while hiding the identities of the items that are being accessed. Most of the earlier ORAM constructions were based on the hierarchical data structure of Goldreich and Ostrovsky [3]. Shi et al. [9] introduced a binary tree ORAM, which is simpler and more efficient than the classical hierarchical ORAM. Gentry et al. [2] have followed them and improved the scheme. In this work, we improve these two constructions. Our scheme asymptotically outperforms all previous tree based ORAM schemes that have constant client memory, with an overhead of $O(\log^{2+\epsilon} N \log^2 \log N)$ per operation for a $O(N)$ storage server. Although the best known asymptotic result for ORAM is due to the hierarchical structure of Kushilevitz et al. [6] ($O(\frac{\log^2 N}{\log \log N})$), tree based ORAM constructions are much simpler.

Keywords: Oblivious RAM.

1 The Tree Oblivious RAM schemes of Shi et al and Gentry et al.

The goal of ORAM constructions is to enable a client to access items stored at the server in a way that prevents the server from learning the client's access pattern, i.e., prevents the server from distinguishing between two different access patterns of the same length. The number of items stored at the server is denoted N. The items that are stored at the server are always kept encrypted using a key known only to the client. When the client accesses an item, it is decrypted, and when it is written back to the server storage it is re-encrypted. Encryption is done using a semantically secure encryption scheme which ensures that different encryptions of the same plaintext are indistiguishable from encryptions of different plaintexts.

The classical Oblivious RAM construction of Goldreich and Ostrovsky [3] is based on a hierarchical data structure, where there are a number of levels (basically $\log N$ levels) and each level is larger then the previous level by a constant multiplicative factor. The main invariants of the system are that in each access to an item every level must be touched, and that two accesses to the same item are indistinguishable from two accesses to different items. To achieve this property, an item must be relocated whenever it is accessed, and the simple way to do this is to move the item to the top layer. In order to prevent overflows of layers, items have to be gradually moved down to lower levels. Many works have been following this paradigm

(e.g., [11, 12, 8, 4, 6]). The main goal of those results was to improve performance, mainly in terms of the amount of data that the server must store, and the number of actual accesses of the client to the server storage for every transaction (which is also of the same order as the computational overhead of the server). The goal of improving the overhead has been achieved – in particular by Kushilevitz et al. [6], but the conceptual effort required for understanding and implementing the constructions has increased. Shi et al. [9] have introduced a different kind of tree based ORAM construction, which is much simpler than the hierarchical constructions, and at the time achieved the best worst case complexity, $O(\log^3 N)$. That scheme was later improved by Gentry et al. [2]. Our construction is based on this type of construction.

Lately, Stefanov et al. proposed Path-ORAM [10], a practical ORAM construction that requires only $O(\log^2 N/\log B)$ bandwidth overhead for a block size of $B \log N$ bits. A major difference between our work and Path-ORAM is the client memory size. Whereas our work, as most previous ORAM construction, uses constant client memory, Path-ORAM uses a poly logarithmic client memory (which is equal to $O(\log^2 N/\log B)\omega(1)$ if a stash is stored at the client, or $O(\log N)\omega(1)$ if a stash is stored at the server).

In this section we shortly describe the previous tree based constructions of Shi et al. [9] and Gentry et al. [2] (for the detailed descriptions, the reader is referred to the original papers). Table 1 compares various variants of these schemes and ours.

	client memory	server storage	amortized comp. overhead	worst case comp. overhead
Shi et al [9] using trivial bucket	$O(N)$	$O(N \log N)$	$O(\log^2 N)$	$O(\log^2 N)$
Shi et al [9] using $\sqrt{n}$-ORAM	$O(N)$	$O(N \log N)$	$\tilde{O}(\log^{1.5} N)$	$\tilde{O}(\log^2 N)$
Gentry et al. [2] ($k = \log N$)	$O(N)$	$O(N)$	$O(\frac{\log^2 N}{\log\log N})$	$O(\frac{\log^2 N}{\log\log N})$
Gentry et al. [2] using $\sqrt{n}$-ORAM	$O(N)$	$O(N)$	$O(\log^{1.5} N)$	$O(\log^2 N)$
Our construction	$O(N)$	$O(N)$	$\tilde{O}(\log^{1+\epsilon} N)$	$\tilde{O}(\log^{1+\epsilon} N)$
Shi et al [9] using trivial bucket	$O(1)$	$O(N \log N)$	$O(\log^3 N)$	$O(\log^3 N)$
Shi et al [9] using $\sqrt{n}$-ORAM	$O(1)$	$O(N \log N)$	$\tilde{O}(\log^{2.5} N)$	$\tilde{O}(\log^3 N)$
Gentry et al. [2] ($k = \log N$)	$O(1)$	$O(N)$	$O(\frac{\log^3 N}{\log\log N})$	$O(\frac{\log^3 N}{\log\log N})$
Gentry et al. [2] using $\sqrt{n}$-ORAM	$O(1)$	$O(N)$	$O(\log^{2.5} N)$	$O(\log^3 N)$
Our construction	$O(1)$	$O(N)$	$\tilde{O}(\log^{2+\epsilon} N)$	$\tilde{O}(\log^{2+\epsilon} N)$

Table 1: A comparison between tree ORAM schemes.

A few comments are due with regards to Table 1: (1) The $\tilde{O}$ notation hides poly $\log\log N$ terms (following [9]), so $\tilde{O}(N)$ is equivalent to $O(N \cdot poly \log\log N)$; (2) $\sqrt{n}$-ORAM is a simple ORAM construction, with $\sqrt{n}$ additional server storage and $O(\sqrt{n}\log^2 N)$ amortized computational overhead (see Appendix A for further explanation). Gentry et al. [2] do not refer to using $\sqrt{n}$-ORAM. Rather, this is our analysis of their scheme.

1.1 The Binary Tree ORAM of Shi et al.

Our work is founded on the following construction, which is proposed by Shi et al. [9]:

- **Data Structure.** The server storage is organized as a binary tree: Its height is $\log N$, and each node

in the tree is a bucket whose capacity is $\log N$. The bucket can be implemented trivially (as a buffer or a sequential array), or as a square-root ORAM (which is a simple ORAM construction described in [3], see Appendix A). Each item is assigned an identifier which is associated with a specific leaf of the tree. The item is always located in one of the nodes along the path from the root to that leaf.

The translations of items' addresses to their identifiers are kept in a *position map*, which is stored at the client memory or at another ORAM construction that is stored at the server (the latter is the recursive variant that aims to reduce the client memory size from $O(N)$ to $O(1)$, see Section 1.1.1). Assume that blocks are of constant size B. Then the size of the position map is smaller than the size of the original data by a factor of $O(B/\log(N))$, since instead of storing each block it is only required to store a pointer to it. Therefore even the construction which stores the position map in the client is an improvement over the trivial solution of storing the blocks themselves at the client.

- **Access.** To read or write an item, the client performs a lookup by: (a) identifying the path (using the position map); (b) going over the path from the root to the leaf, and in each node along the path searching for the item (either using a sequential scan, in the case of a trivial bucket, or using the internal-ORAM access interface); (c) writing the reencrypted item back to the root after randomly assigning it a new identifier, that corresponds to a new leaf. In order to avoid an overflow of the root node, for each access, an *eviction* procedure is called.

- **Eviction.** The client randomly chooses two (or, wlog, a larger constant number) buckets at every level of the tree. For each such bucket it pops from the bucket a single arbitrary item. If the bucket is empty, the client pops a dummy item.

  (The authors of [9] define an ORAM *pop* operation, using a linked list and the standard ORAM operations. The idea is that a linked list maintains an order on the items in the bucket, and invoking a pop operation removes the first item and updates the list, obliviously. The overhead of the pop operation is similar to that of an ORAM access operation. See [9] section 2.4 and appendix B for details.)

  After performing the pop operation the client writes items to both descendants of the node (bucket). It writes the popped item to the descendant node which is in the path to the leaf associated with the identifier of the popped item. (If the pop operation is applied to an empty bucket, then a dummy item is written to a random descendant). A dummy item is written to the other descendant. (By writing a dummy item we refer to a process that externally looks like writing an arbitrary value to the bucket, but internally does not store any value. Writing a dummy item to a bucket does not decrease its capacity.)

  Writing to descendants is done in a predetermined order, e.g., always write first to the left descendant and then to the right descendant. A crucial point is that a probabilistic analysis shows that when performing evictions in every round, then with high probability no overflows occur.

Roughly speaking, the eviction cost is proportional to the access cost. The amortized complexity of the scheme is therefore composed of reading the position map (which costs $O(1)$ if stored in the client memory and is implemented as a hash table), traversing the path from the root to the leaf (of length $O(\log N)$), and at each node, performing a number of I/O operations which depends on the bucket's size and the node internal implementation (for example, for a bucket of size $\log N$, the overhead is $O(\log N)$ if the bucket is a implemented as a trivial bucket, or $O(\log^{0.5} N \log\log N)$ if the node is implemented as a square-root ORAM).

In total, we get an amortized overhead of $O(\log^2 N)$ for the trivial buckets implementation or of $O(\log^{1.5} N \log\log N)$ for the square-root ORAM bucket implementation. This complexity is for the basic construction in which the position map is saved at the client using $O(N)$ memory.

1.1.1 Tree Based ORAM Constractions – The Recursive Construction

In a tree based ORAM construction, the associations of identifiers (corresponding to leaves or branches in the tree) to items are saved in a data structure, called a *position map*. In the basic and simple construction, the position map is stored at the client memory. This requires a linear-size client memory. To acheive a constant-size client memory, the position map should be stored at the server storage. Naturally, this must be done obliviously, so another tree based ORAM structure is used for this purpose. Since the size of a position map is a constant fraction of the size of the database it indexes, the size of the ORAM which is used for storing the position map is a constant fraction of the size of the ORAM that stores the data items. The position map of the second ORAM is stored in a third ORAM, and so on. A logarithmic number of steps (ORAM structures) are required, until the size of the last position map is small enough to be stored in the constant client memory.

In our computational overhead analysis we use the term *recursion factor* to denote the cost of storing the position map at the server. The recursion factor is logarithmic in the number of data items, so in our complexity calculation we refer to it as $O(\log N)$.

The recursive structure was developed by [9] and the recursive factor calculation was done by [2]. We follow them and use it in the description of their constructions and also in our construction.

Taking the binary tree ORAM of [9] as an example, and applying to it the recursive construction by storing the position map in another ORAM in the server, yields a client memory of constant size. The cost is another multiplicative logarithmic factor for the complexity. The resulting computational complexity is $O(\log^3 N)$ (for the trivial buckets implementation), or $O(\log^{2.5} N \log\log N)$ (for the square-root ORAM implementation), both with a server storage of $O(N \log N)$.

1.2 The k-ary Tree ORAM with Larger Buckets Utility of Gentry et al.

Gentry et al. [2] have suggested two improvements to the scheme of Shi et. al. [9].

- **Reducing the server storage by enlarging the utility of the buckets.** While the binary tree of Shi et al. [9] has N leaves, Gentry et al. propose a tree with only N/k leaves, where k is the security parameter (such that privacy loss occurs with probability exponentially small in k, and usually, $k = O(\log N)$; note that the privacy loss happens with probability that is polynomially small in N, as is often the case in this research area.). The brilliant observation that enables this improvement is the following: In a tree with $O(N)$ leaves there are N items that are located in $O(N)$ nodes. The expected number of items in each node is $O(1)$, yet the capacity of a node/bucket is $O(k) = O(\log N)$ to support the most occupied node. This is the reason for the requirement for a total server storage of $O(Nk) = O(N \log N)$.

  Gentry et al. suggest to decrease the tree height while enlarging the buckets, such that the expected number of items in each node would be $O(k)$ rather than $O(1)$. This is enabled by increasing the node capacity from k to $2k$, resulting in both the maximum size of a bucket and its expected size being $O(k)$. (They use the Chernoff bound to prove that the probability of bucket overflow remains upper bounded by $e^{-k/3} \approx 2^{-k/2}$.) The tree height is only $\log \frac{N}{k}$. The result is a tree with only N/k

leaves that can store all the N items. I.e. a total storage of only $O(N)$ items at the server. This also improves the complexity: instead of traversing a $\log N$ height tree and going over $O(\log N)$ items in each level (assuming trivial buckets) in the original construction of Shi et al., the improved scheme has a tree height of only $\log \frac{N}{k} = \log \frac{N}{\log N} = \log N - \log \log N$, while the buckets size is still $O(\log N)$.

- **Reducing the complexity by using a k-ary tree.** The basic observation behind this improvement is that increasing the tree degree to k reduces the tree height from $\log \frac{N}{k}$ to $\log_k \frac{N}{k} = \frac{\log N}{\log k} - 1$ and as a result reduces the overhead. A problem arises since using the eviction procedure of Shi et al. on a k-ary tree increases its complexity (since all k descendants of each node are involved in the eviction procedure). Therefore Gentry et al. propose a new eviction procedure. This eviction procedure is deterministic and uses a predefined order of the leaves. As in [9], each item is assigned an identifier which has a one-to-one correspondence with a specific leaf of the tree. The item is always located in one of the nodes along the path from the root to that leaf. The eviction procedure performs the following on each of the leaves, by a pre-defined order: Let L be the leaf whose turn has arrived according to the pre-defined order of the leaves in the eviction process. L also defines a path P from the root to L. The eviction procedure scans all of the buckets on this path P, looking for items that can be pushed down. An item in node s with an identifier $L' \neq L$ can be pushed down if there is a bucket in a lower level than s that: (a) is on the path P, and (b) is on the path P' from the root to L'. Such an item is pushed as down as possible to the lowest bucket that belongs to both path P and P'.

Overall, the computational complexity of this scheme improves that of [9] by a factor of $O(\log \log N)$ and the server storage is decreased by a factor of $O(\log N)$. For a constant client memory, the server storage is $O(N)$ and the computational complexity is $O(\frac{\log^3 N}{\log \log N})$.

2 Our Construction

Before we present our suggestions for improvement, we would like to analyze one aspect of the eviction procedures of the two previous schemes: the number of items that are moved down in each step. The first scheme [9] moves down in each step one item in each of two buckets in each level – a total of $2h$ items are moved one level down in each step (where h is the height of the tree and is equal to $O(\log N)$). The second scheme [2] affects one bucket in each level (the node that is on the path from the root to the leaf), and from that bucket it moves down each item whose path continues down along the path to that leaf. The chance for an arbitrary item to have the same "next path node step" as the accessed item is $1/k$ (since k is the degree of the tree, and the tree is kept as balanced as possible). The expected number of items in a bucket is k, so in expectation, one item is moved down[1]. In summary, in each step we expect to move down one item in each level, a total of h items in each step (here $h = \frac{\log N}{\log k} - 1$).

Our construction combines parts of the two previous schemes with some modifications, and has the following properties: (a) We apply an inner ORAM construction in each node, recursively, i.e., each node is implemented as a tree ORAM (as opposed to [9] that have used either a trivial bucket or a square-root ORAM). (b) The tree degree (fanout) is $d = \log^\epsilon N$ for $0 < \epsilon < 1$. (c) The recursive construction (obliviously storing the position map in another ORAM construction, at the server) is the same as in the previous schemes.

[1] For the item that is moved down, it holds with probability $1/k$ that it is moved down another level (total of two levels down), with probability of $1/k^2$ it is moved down three levels, etc. We ignore this observation, because it does not change the overall calculation.

The construction is as follows:

- **Data Structure.** We use a d-ary tree where $d = \log^{\epsilon} N$ for $0 < \epsilon < 1$, and where each node is a bucket with capacity $M = 2 \log N$. The tree height is $\log_d N = \frac{\log N}{\log(\log^{\epsilon} N)} = O(\frac{\log N}{\log \log N})$. Nodes are implemented as a binary tree ORAM with a linked list (to enable a pop operation, see Section 1.1 and [9] section 2.4 for details).

  As in the previous schemes, each item is assigned an identifier which always stored in one of the nodes along the path from the root to that leaf. A position map is used to translate items' addresses to their identifiers.

- **Access.** The access procedure is similar to the access procedure of the previous schemes, including the execution of the eviction procedure for each access, and writing back to the root after assigning a new random identifier to the item: The client performs a lookup by: (a) identifying the path (using the position map); (b) going over the path from the root to the leaf, and in each node along the path searching for the item; (c) writing the reencrypted item back to the root after randomly assigning it a new identifier that corresponds to a new leaf. In order to avoid an overflow of the root node, for each access, an *eviction* procedure is called.

  Each node bucket is implemented internally as an ORAM. The capacity is $M = 2 \log N$. The internal ORAM construction can be implemented either as our ORAM construction, recursively, or as other known ORAM construction (e.g., [2], or [9])[2]. To simplify the calculation of complexity we assume here that the internal ORAM is implemented as the simple binary tree ORAM construction of [9] with trivial buckets, whose worst and amortized case cost of an access is $O(\log^3(M))$ for a data structure of size M. In our case, the capacity of a node, $M = 2 \log N$. Therefore, the cost of an access for an internal ORAM is $InternalORAM = O(\log^3(2 \log N))$.

  The overhead of an access to an item in the external ORAM scheme that we implement, when the client stores the position map locally (with an overhead of $O(N)$ storage) is therefore $TreeHeight \times InternalORAM = O(\frac{\log N}{\log \log N} \times \log^3(2 \log N)) = O(\log N \log^2 \log N)$. When using constant client memory, the construction has to be recursively called $\log N$ times, and therefore the overhead is $TreeHeight \times InternalORAM \times RecursionFactor = O(\frac{\log N}{\log \log N} \times \log^3(2 \log N) \times \log N) = O(\log^2 N \log^2 \log N)$.

- **Eviction.** The eviction procedure is similar to the eviction procedure of Shi et al. [9] (see Section 1.1). At each level, the client randomly chooses two nodes. From each of these nodes, it pops an item using the pop procedure of [9] and moves it down one level, inserting it to the appropriate descendant (if the original node is empty - a dummy item is inserted to a random descendant). Dummy items are inserted to each of the other descendants. As in [9], there are two items at each level that are successfully evicted. The following explanation describes in more detail the eviction procedure:

  For each level $l \in \{1 \dots h\}$

[2] Using trivial ORAM is too costly. If the internal ORAM is implemented trivially, i.e., a bucket is read to the client memory at once, and the position map is stored in the client memory (the basic, non-recursive, construction), then the complexity of a single access is equal to the tree height, i.e., $O(\frac{\log N}{\log \log N})$. However, this requires a non-constant client memory. For a constant client memory, reading the position map from another ORAM structure (the recursive construction, see Section 1.1.1) costs a multiplicative factor of $O(\log N)$, and implementing a trivial ORAM requires reading the bucket item by item, i.e., another multiplicative factor of $M = O(\log N)$, so together there is a multiplicative factor of $O(\log^2 N)$, resulting in total complexity for a single access of $O(\frac{\log^3 N}{\log \log N})$. (This is the cost of the access operation only, without the eviction.)

1. Randomly choose two nodes.
2. For each such node z:
   (a) Pop an item x from z (using the pop procedure of [9]).
   (b) If x is a real item: Compute the descendant node t that is in the path leading to leaf assigned to the identifier of x.
   (c) Else: Set x to a dummy value and set $t = 1$.
   (d) Insert items *to all* d descendants, by their order, where the value x is inserted to descendant t, and dummy items are inserted to the other descendants.

A few comments about the implementation: (1) Note that dummy items are not counted when the load of a bucket is estimated, i.e., from the server storage point of view, inserting a dummy item is meaningless, and a bucket cannot be overflowed as a consequence of inserting a dummy item. (2) The computation in step 2.b is done using the position map. (3) Inserting an item into a node (Step 2.d) is done using the internal ORAM interface. (4) Each node (internal ORAM) maintains a counter of the number of real items that it contains.

Complexity analysis: The computational overhead of an eviction is composed of the following operations: The whole procedure is performed for each level, in a tree of height $h = O(\frac{\log N}{\log\log N})$. Popping an item (step 2.a) using the pop procedure of [9] is an internal ORAM operation, which costs $O(\log^3 \log N)$. Computing the descendant to which an item should be evicted (steps 2.b, 2.c) takes $O(1)$ operations if the position map is stored at the client. Moving an item to a lower level (step 2.d) includes d insertions that are implemented as internal ORAM operations, each taking $O(\log^3 \log N)$ steps. Overall, if the position map is stored at the client (the basic construction), the overhead of the eviction stage is *TreeHeight*$\times$ *2 nodes in each level* $\times$ *treeDegree* $\times$ *InternalORAM* $= O(\frac{\log N}{\log\log N} \times \log^\epsilon N \times \log^3 \log N) = O(\log^{1+\epsilon} N \log^2 \log N)$. For the recursive construction (see Section 1.1.1), the overhead of the eviction stage is *TreeHeight*$\times$ *2 nodes in each level* $\times$ *treeDegree* $\times$ *InternalORAM* $\times$ *RecursionFactor* $= O(\frac{\log N}{\log\log N} \times \log^\epsilon N \times \log^3 \log N \times \log N) = O(\log^{2+\epsilon} N \log^2 \log N)$ for an $O(N)$ server storage and constant client memory.[3]

To summarize, our construction is a tree based ORAM, where each node is implemented internally as a tree based ORAM with capacity $2 \log N$. The degree of the tree is $d = \log^\epsilon N$, and its height is $O(\frac{\log N}{\log\log N})$. We get the best performance of all tree based ORAM constructions that have a constant client memory and $O(N)$ server storage. Namely, an amortized and worst case computational and communication complexity of $O(\log^{2+\epsilon} N \log^2 \log N)$ per ORAM operation.

3 Security Analysis

Definition 1. *The input y of the client is a sequence of data items, denoted by $((v_1, x_1), \ldots, (v_n, x_n))$ and a corresponding sequence of operations, denoted by $(op_1, \ldots, op_m)$, where each operation is either a read*

[3] We refer here to the recursive construction, in which the client memory is constant and the position map is stored in another ORAM construction. In addition, as before, for the internal ORAM construction, we use [2]. Therefore, $InternalORAM = O(\log^3(\log N))$.

operation, denoted read(v)*, which retrieves the data of the item indexed by* v*, or a write operation, denoted* write(v, x)*, which sets the value of item* v *to be equal to* x*.*

The access pattern $\mathcal{A}(y)$ *is the sequence of accesses to the remote storage system. It contains both the indices accessed in the system and the data items read or written. An oblivious RAM system is considered secure if for any two inputs* y, y' *of the client, of equal length, the access patterns* $\mathcal{A}(y)$ *and* $\mathcal{A}(y')$ *are computationally indistinguishable for anyone but the client.*

Theorem 2. *If no overflows occur than our oblivious RAM construction is secure in the sense of Definition 1.*

Before proving the theorem, let us provide intuition about the proof. Examine the view of the server during the access and eviction operations. We assume that the internal ORAM is secure. The access pattern that the server sees is the ordered list of the nodes that are accessed in the tree. This view is independent of the actual access pattern:

- **Access.** The view of the server is always composed of an access to the position map, and then an access to a single node at each level. The selection of these nodes is according to the item identifier, which is random. Therefore, the list of accessed nodes is random. Inside nodes, accesses are oblivious based on the security of the internal ORAM.
- **Eviction.** The view of the server is always composed of (a) ORAM pop operations on two nodes that are randomly chosen at each level, and (b) ORAM insert operations to each of the descendants of those two nodes, by a pre-defined and fixed order. Due to the semantically secure encryption of the items, the server cannot learn anything about the items that are involved in the pop or insert operation.

Based on these two observations, the server can simulate the access pattern without communicating with the client. Note that this is correct as long as there are no overflows. Therefore, we must ensure, that with high probability no overflows occur at any node. We analyze this property in the following section.

Proof. Let y and y' be any two inputs of the clients of equal length q. Each of them is composed of q operations. For each operation the following is specified: (a) the operation type (whether it is a read or a write); (b) the index that is accessed (item id); and (c) the value that is written (if the operation is a write, or "don't care" if the operation is a read).

Let us define y_0, y_1, y_2 and y_3 such that $y_0 = y$ and $y_3 = y'$. We define y_1 and y_2 as follows: The item ids and values of y_1 are identical to those of y_0, while the operations of y_1 are identical to those of y_n. (Namely, if a write operation is changed to a read, then the original value, rather than the newly written value, is written to the top bucket; if a read is replaced by a write then an arbitrary value is written to the top bucket.) Similarly, y_2 is identical to y_1 in respect of the operations and values, and is identical to y' in respect of the item ids, and y_3 is identical to y_2 in its operations and item ids, and to y' in its values. It is easy to see that $y_3 = y'$. A simple example is presented in Table 2.

We show that for $i = 0, 1, 2$, there is no polynomial adversary that can distinguish between y_i and y_{i+1}. From this it follows that no polynomial adversary can distinguish between y and y', and this proves Theorem 2.

First, each operation is implemented as both a read and a write, so the access patterns of the sequences y_0 and y_1 are computationaly indistinguishable (they differ only in the operations). Second, each item in the ORAM structure is sematically secure encrypted, thus, the access patterms of y_2 and y_3 are computationaly indistinguishable (they differ only in the values). Otherwise, if there is a polynomial adversary distinguishing between them, then we can use it to construct an adversary that distinguishes between encryptions of two values. This will break the semantic security of the encryption.

$y = y_0$	y_1	y_2	$y_3 = y'$
$read\ 15$	$read\ 15$	$read\ 4$	$read\ 4$
$write\ 71, a$	$write\ 71, a$	$write\ 7, a$	$write\ 7, b$
$read\ 18$	$write\ 18, z$	$write\ 5, z$	$write\ 5, c$

Table 2: An example of the sequences in the proof of Theorem 2.

It is left to show that the access patterms of y_1 and y_2 are also computationaly indistinguishable, and this suffices to complete the proof that the access patterns $\mathcal{A}(y)$ and $\mathcal{A}(y')$ are computationaly indistinuishable. The sequences y_1 and y_2 differ only in the item ids, the indices, and this is what we refer to in the sequel of this proof.

1. Lets us assume that the position map is stored at the client memory. For a given query, the buckets that are accessed are along a path which is chosen randomly when the requested item is assigned a leaf. This mapping is random and known only to the client. As a result, the ordered path of buckets that are accessed during a sequence of requests looks random to anyone but the client. Consequently, the paths of buckets accessed in the two access patterns $\mathcal{A}(y_1)$ and $\mathcal{A}(y_2)$ are indistinguishable for anyone but the client.

   Note: At the end of each I/O operation, the item is written back to the root, with a new randomly chosen leaf it is assigned to (independent of the previous leaf this item was assigned to). Thus, the next time this item is requested, the accessed path is random and uncorrelated to the previously accessed path.

2. In the evict operation, the buckets from which items are evicted are chosen at random, the pop operation is oblivious (due to the security of the internal ORAM), and items are inserted (obliviously, due to the security of the internal ORAM) in a fixed order to all descendants of the evicted buckets. Specifically,

   - If the internal ORAM is implemented as a trivial ORAM, i.e., the entire bucket is read into the client memory, the indistinuishability is obvious.
   - If the internal ORAM is implemented as a known and proved ORAM construction, e.g., [9], the indistinuishability is derived from the security of this underline ORAM construction.
   - If the internal ORAM is implemented as our ORAM, recursively, we can prove the above indistinguishability property by induction, where the base case is a small enough ORAM that can be implemented as a trivial ORAM, i.e., can be read into the constant-size client memory.

   Thus, for the eviction operation too, the two access patterns $\mathcal{A}(y_1)$ and $\mathcal{A}(y_2)$ are computationally indistinguishable for anyone but the client.

3. In the recursive construction (see Section 1.1.1) the position map is stored in another ORAM construction at the server, and we can prove the above indistinguishability by induction, where the base case is a small enough position map that can be stored in the constant-size client memory.

3.1 An Analysis of the Probability of an Overflow

Theorem 3. (An upper bound on the overflow probability). *The probability that there is a bucket that contains more than* $2\log N$ *real items is* $o(1)$. *In particular, denote the bucket-load as* b. *For a fixed time and a fixed bucket, it holds that* $Pr[b > 2\log N] \leq 1/N^2$.

We follow a similar path as the analysis of [9] (Appendix A)[4]. In order to prove that the overflow probability is negligible, we model the eviction as a stochastic process and show that there is a steady state in which items are moved down without nodes overflowing.

We define a Markov process with two parameters on a specific node (bucket): λ - the probability that an item is inserted to the node (i.e., the load is increased by 1), and μ - assuming the node is not empty, the probability that an item is removed from the node (i.e., the load is decreased by one). This is a model of a *simple queue* (or a "random walk with a wall"), since the load cannot be decreased beyond zero.

Let us define a simple queue more accurately (following [7] Section 7.3.1; for more details, the reader is referred to [7]):

Definition 4. *A* simple queue *is a length-*n *bounded queue with discrete steps as follows:*

- *If the queue is not full (has fewer than* n *items), then with probability* λ *a new item is added to the queue.*
- *If the queue is not empty, then with probability* μ *an item (the head of the queue) is removed from the queue.*
- *With the remaining probability, the queue is unchanged.*

This yields a finite-state Markov chain with the following transition matrix:

$$\begin{aligned} P_{i,i+1} &= \lambda & if\ i < n; \\ P_{i,i-1} &= \mu & if\ 0 < i; \\ P_{i,i} &= 1-\lambda-\mu & if\ 0 < i < n \\ P_{0,0} &= 1-\lambda \\ P_{n,n} &= 1-\mu \end{aligned}$$

Our scheme is a bit different: at each step, an item can be inserted to a node, an item can be removed from a node, or both events can happen. The transition probabilities for this scheme are:

$$if\ 0 < i < n: \quad P_{i,i+1} = \lambda(1-\mu) = \overline{\lambda} \tag{1}$$

[4] Note, that as in [9], the analysis uses assumptions that makes the probability of overflow highr than the actual overflow probability. The first assumption is that when an item is accessed, it is not removed from the bucket. The second assumption is that all the blocks are distinct (i.e., there are no duplicates blocks that can be removed). Each of these assumptions by itself increases the bucket load, and thus the overflow probability. Consequently, the actual bucket load and overflow probability in the schemes are bounded by the correspomding load and probabilty in the analyzed processes.

$$P_{i,i-1} = \mu(1-\lambda) = \overline{\mu} \tag{2}$$
$$P_{i,i} = 1 - \overline{\lambda} - \overline{\mu} \tag{3}$$
$if\ i = 0:$
$$P_{0,1} = \lambda(1-\mu) = \overline{\lambda} \tag{4}$$
$$P_{0,0} = 1 - \overline{\lambda} \tag{5}$$
$if\ i = n:$
$$P_{n,n-1} = \mu(1-\lambda) = \overline{\mu} \tag{6}$$
$$P_{n,n} = 1 - \overline{\mu} \tag{7}$$

Namely, the system behaves as a simple queue with $\overline{\lambda} = \lambda(1-\mu)$ and $\overline{\mu} = \mu(1-\lambda)$.

A simple queue is an irreducible, finite, and aperiodic Markov chain, so it has a unique stationary distribution $\overline{\pi} = (\pi_0, \pi_1, \ldots, \pi_n)$, such that $\overline{\pi} = P\overline{\pi}$. From this, one can conclude that the probability that a node in our scheme (i.e., a queue) is empty is $\pi_0 = \frac{1}{\sum_{i=0}^{n}(\overline{\lambda}/\overline{\mu})^i} = \frac{1-\frac{\overline{\lambda}}{\overline{\mu}}}{1-(\frac{\overline{\lambda}}{\overline{\mu}})^{n+1}} \to 1 - \frac{\overline{\lambda}}{\overline{\mu}}$ (see [7] Section 7.3.1). Now we apply this to our scheme:

Recall that at the beginning the root node is empty (either we begin with an empty structure or all of the items are randomly located at the leaves).

At the root, the probability that an item is inserted is 1 (due to writing back), and the probability that an item is removed is also 1 (the root is always chosen for eviction). Hence $\lambda = \mu = 1$. As a result, it is easy to see that the root is always empty after the eviction. At level 1, $\lambda = 1/d$ since the item that is evicted from the root is inserted to one of the d descendants of the root, and $\mu = 2/d$ since two nodes out of d nodes in the level are chosen for eviction. Let us continue the calculation:

$$\overline{\lambda} = (1-\mu)\lambda = \frac{d-2}{d} \cdot \frac{1}{d} = \frac{d-2}{d^2}$$
$$\overline{\mu} = (1-\lambda)\mu = \frac{d-1}{d} \cdot \frac{2}{d} = \frac{2d-2}{d^2}$$
$$\pi_0 = 1 - \frac{\overline{\lambda}}{\overline{\mu}} = 1 - \frac{d-2}{2d-2} = \frac{d}{2d-2} > \frac{1}{2}$$

For the second level, there is an additional factor - the probability that an item is inserted to a node is dependent on the probability that its parent is not empty (we denote this by $P_{pne} = 1 - \pi_0$). The calculation of π_0 in the parent level is required in order to find P_{pne}.

$$\mu = \frac{2}{d^2}$$
$$\lambda = P_{pne} \cdot \frac{2}{d} \cdot \frac{1}{d} = P_{pne} \cdot \mu$$
$$\overline{\lambda} = P_{pne} \cdot \mu(1-\mu)$$
$$\overline{\mu} = (1 - P_{pne} \cdot \mu)\mu$$
$$\pi_0 = 1 - \frac{\overline{\lambda}}{\overline{\mu}} = 1 - \frac{P_{pne} \cdot \mu(1-\mu)}{(1 - P_{pne} \cdot \mu)\mu} = 1 - \frac{P_{pne}(1-\mu)}{(1 - P_{pne} \cdot \mu)}$$

Since $P_{pne} < \frac{1}{2}$ we get again $\pi_0 > \frac{1}{2}$, namely that more than half of the nodes in the level are empty.

For the third level:

$$\begin{aligned}\mu &= \frac{2}{d^3}\\ \lambda &= P_{pne}\cdot\mu\\ \overline{\lambda} &= P_{pne}\cdot\mu(1-\mu)\\ \overline{\mu} &= (1-P_{pne}\cdot\mu)\mu\end{aligned}$$

Note that both $\overline{\lambda}$ and $\overline{\mu}$ are the same as in the second level (relatively to P_{pne} and μ), so π_0 would also be the same, and since $P_{pne} < \frac{1}{2}$, we get again $\pi_0 > \frac{1}{2}$.

This ratio, between $\overline{\lambda}$ and $\overline{\mu}$, which is always $< 1/2$, is kept for all following levels (relatively to P_{pne} and μ), such that the probability of removing an item from a node is always more than twice larger than the probability of inserting an item to the same node. Due to this invariant, the load of each node has a stationary distribution, which is required for using the following proposition.

A queuing theory result of Hsu and Burke [5] states the following on a Markov process with a stationary distribution.

Proposition 5. *Let $0 < \lambda < \mu \leq 1$. Then, the Markov process $Q(\lambda, \mu)$ satisfies the following properties.*

1. *A unique stationary distribution exists, and is given by $\pi_j = \rho^j(1-\rho)$, where $\rho = \frac{\overline{\lambda}}{\overline{\mu}}$.*
2. *Under stationary distribution, a block departs from the bucket at every time step with probability λ, independent of past departure history.*
3. *Under stationary distribution, the current load of a bucket is independent of past departure history.*

For level i, we model a bucket load by $Q(\lambda_i, \mu_i)$, such that $\lambda_i = \frac{\lambda_{i-1}}{d} = \frac{1}{d^i}$ and $\mu_i = \frac{2}{d^i}$. Therefore the load has a stationary distribution with $\rho_i = \frac{\overline{\lambda}}{\overline{\mu}} = \frac{\frac{1}{d^i}(1-\frac{2}{d^i})}{\frac{2}{d^i}(1-\frac{1}{d^i})} = \frac{d^i-2}{2d^i-2} < \frac{1}{2}$. Using Proposition 5, we can conclude that the probability that a bucket in level i has a load of at least s is bounded by $\rho_i^s < \frac{1}{2^s}$, and in particular, for maximal load $s = 2\log N$ the probability of overflowing is bounded by $\frac{1}{N^2}$.

Note: The above analysis does not apply to the lowest level, since the removal of items from nodes at this level is different. In the analysis of [9] it is a simple "Balls into Bins" (N items into N leaves), but in our work the number of leaves is $N/k = N/\log N$ so the analysis is more complex. Intuitively, the overflow probability is still negligible. What is the probability of more than $2\log N$ balls (out of a total of N balls) falling into a single bin (out of a total of $N/\log N$ bins)? It is a well known fact that with m balls thrown uniformly at random into n bins, where $m \geq n\log n$, the maximum load of any bin is $\Theta(m/n)$, i.e., of the order of the mean. In our case, there are $m = N$ balls, $n = \frac{N}{\log N}$ bins, so $n\log n = \frac{N}{\log N}\log(\frac{N}{\log N}) < \frac{N}{\log N}\log(N) = N = m$, and we get a maximum load of $\Theta(m/n) = \Theta(N/\frac{N}{\log N}) = \Theta(\log N)$ as required.

This proves Theorem 3 and consequently Theorem 2.

References

[1] Ivan Damgård, Sigurd Meldgaard, and Jesper Buus Nielsen. Perfectly secure oblivious RAM without random oracles. Cryptology ePrint Archive, Report 2010/108, 2010. http://eprint.iacr.org/2010/108.

[2] Craig Gentry, Kenny Goldman, Shai Halevi, Charanjit S. Julta, Mariana Raykova, and Daniel Wichs. Optimizing ORAM and Using it Efficiently for Secure Computation. *IACR Cryptology ePrint Archive*, 2013:239, 2013.

[3] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious RAMs. *Journal of the ACM*, 43(3):431–473, 1996.

[4] Michael T. Goodrich and Michael Mitzenmacher. MapReduce Parallel Cuckoo Hashing and Oblivious RAM Simulations. *CoRR*, abs/1007.1259, 2010.

[5] J. Hsu and P. Burke. Behavior of Tandem Buffers with Geometric Input and Markovian Output. *IEEE Transactions on Communications*, 24:358–361, 1976.

[6] Eyal Kushilevitz, Steve Lu, and Rafail Ostrovsky. On the (in)security of hash-based oblivious ram and a new balancing scheme. Cryptology ePrint Archive, Report 2011/327, 2011. http://eprint.iacr.org/.

[7] Michael Mitzenmacher and Eli Upfal. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, 2005.

[8] Benny Pinkas and Tzachy Reinman. Oblivious RAM Revisited. In *CRYPTO 2010*, volume 6223, pages 502–519, 2010.

[9] Elaine Shi, T.-H. Hubert Chan, Emil Stefanov, and Mingfei Li. Oblivious RAM with $O((\log N)^3)$ Worst-Case Cost. In Dong Hoon Lee and Xiaoyun Wang, editors, *ASIACRYPT*, volume 7073 of *Lecture Notes in Computer Science*, pages 197–214. Springer, 2011.

[10] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: An Extremely Simple Oblivious RAM Protocol. *CoRR*, abs/1202.5150v2, 2012.

[11] Peter Williams and Radu Sion. Usable PIR. In *NDSS*, 2008.

[12] 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, 2008.

A Square Root ORAM

A square-root ORAM is a predecessor of the hierarchical ORAM. From a practical point of view, in many realistic scenarios, the square-root ORAM is more efficient than the hierarchical construction. The basic idea of the square-root ORAM is to maintain at the server a main memory (AKA "permuted memory") and a cache (AKA "shelter"). To store N items, the main memory should be of size $N + \sqrt{N}$ and the cache should be of size $\sqrt{N}$. The N items and additional $\sqrt{N}$ dummy items are stored in the main memory, ordered (permuted) by a secret randomly chosen hash function. Accessing an item is composed of scanning the entire cache and then probing the main memory according to the hash function (if the item has been found in the cache, a dummy item is probed in the main memory). Whenever an item is accessed, it is written back to the next available slot in the cache. After each $\sqrt{N}$ accesses a new hash function is chosen and the content

of the cache is moved into the main memory - the entire content of the main memory is permuted according to the new hash function. For a constant client memory, the server storage is $O(N + \sqrt{N})$, the amortized complexity of a single access is $O(\sqrt{N} \log N)$, and the worst case complexity is $O(N \log^2 N)$. For more details and analysis, the reader is referred to [3].