Oblivious RAM with $O((\log N)^3)$ Worst-Case Cost

Elaine Shi — PARC/UC Berkeley
T-H. Hubert Chan — HKU
Emil Stefanov — UC Berkeley
Mingfei Li — HKU

Abstract

Oblivious RAM (O-RAM) is a useful primitive that allows a client to hide its data access patterns from an untrusted server in storage outsourcing applications. This paper proposes novel O-RAM constructions that achieves poly-logarithmic worst-case cost, while consuming constant client-side storage. Our techniques for constructing Oblivious RAM are fundamentally different from previous approaches. Specifically, we organize the O-RAM storage into a binary tree over data buckets, while moving data blocks obliviously along tree edges. Our construction (instantiated the trivial bucket O-RAM) has remarkable conceptual simplicity, and eliminates the need to perform expensive oblivious sorting operations. As a result, to the best of our knowledge, our construction is by far the most practical scheme with constant client-side memory, under realistic parameterizations.

1 Introduction

Oblivious RAM (or O-RAM for short) [5–7, 12, 14, 18] is a useful primitive for enabling privacy-preserving outsourced storage, where a client stores its data at a remote untrusted server. While standard encryption techniques allow the client to hide the contents of the data from the server, they do not guard the access patterns. As a result, the server can still learn sensitive information by examining the access patterns. For example, Pinkas and Reinman [14] gave an example in which a sequence of data access operations to specific locations $(\mathsf{u}_1, \mathsf{u}_2, \mathsf{u}_3)$ can indicate a certain stock trading transaction, and such financial information is often considered highly sensitive by organizations and individuals alike.

Oblivious RAM allows the client to completely hide its data access patterns from the untrusted server. It can be used in conjunction with encryption, to enable stronger privacy guarantees in outsourced storage applications. Not surprisingly, the client has to pay a certain cost in order to hide its access patterns from the server. Among all prior work in this space, the seminal constructions recently proposed by Goodrich and Mitzenmacher [7] achieve the best asymptotic performance in terms of amortized cost. Specifically, let N denote the maximum capacity of the O-RAM. Goodrich and Mitzenmacher show that with $O(1)$ client-side storage, one can achieve $O((\log N)^2)$ amortized cost, i.e., each oblivious data request translates into $O((\log N)^2)$ non-oblivious data access operations on average. Goodrich and Mitzenmacher also show that with $O(\sqrt{N})$ client-side storage, one can achieve $O(\log N)$ amortized cost [7].

O-RAM with sublinear worst-case cost. Most prior work on O-RAM focuses on reducing its amortized cost, [6, 7, 14, 18], while not giving much consideration to the worst-case cost. Specifically, while achieving logarithmic or poly-logarithmic amortized cost, these constructions [6, 7, 14, 18] have a worst-case cost of $\Omega(N)$, due to the occasional reshuffling operations which can take up to $\Omega(N)$ time. Such $\Omega(N)$ worst-case behavior renders these schemes impractical in real-world applications; since every now and then, a data request can be blocked waiting for $\Omega(N)$ operations to complete. When this happens, the perceived waiting time for the user would be unacceptable.

Ostrovsky and Shoup were the first to demonstrate in a seminal work [13] how to achieve $O(\log N^3)$ worst-case cost, by spreading the reshuffling operation over time.

1.1 Our Contributions

This paper proposes novel O-RAM constructions that achieve both poly-log amortized and worst-case cost, while consuming $O(1)$ client-side storage, and $O(N \log N)$ server-side storage.

We make the following contributions.

Novel techniques. Most existing constructions [6, 7, 14, 18] inherit the hierarchical solution initially proposed by Goldreich and Ostrovsky [6]. Therefore, these constructions also inherit the periodic reshuffling operations required by the Goldreich and Ostrovsky construction [6], which rely on an expensive primitive called oblivious sorting.

Our techniques are fundamentally different from other O-RAM constructions which build on top of the hierarchical construction by Goldreich and Ostrovsky [6]. In particular, our technique involves the partitioning of an O-RAM into smaller instances called *bucket O-RAMs*, and obliviously evicting data amongst the bucket O-RAMs. This novel technique is crucial for achieving good practical performance.

Efficient pracitical performance. Our construction (based on trival bucket O-RAM) is by far the most practical construction under constant client memory. Simulation results demonstrate a very small constant in our asymptotic bound (see Section 5). Under realistic settings (e.g., 1GB to 1TB of storage capacity, 64KB block size), our scheme is 3 orders of magnitude more efficient (by a conservative estimate) than the construction by Ostrovsky and Shoup [13], which also achieves $O((\log N)^3)$ worst-case performance, but suffers from a huge constant in the asymptotic bound.

Conceptual simplicity. Our novel techniques also enable us to build O-RAM constructions with remarkable conceptual simplicity in comparison with previous schemes. In particular, our construction (based on trivial bucket O-RAM) requires no oblivious sorting or reshuffling, no hashing or Cuckoo hashing (or its oblivious simulation such as in the Goodrich-Mitzenmacher construction [7]).

1.2 Technical Highlights

We propose a novel *binary-tree based construction* (Section 3). Basically, the server-side O-RAM storage is organized into a binary tree over small data buckets. Data blocks are evicted in an oblivious fashion along tree edges from the root bucket to the leaf buckets. While in spirit, the binary-tree based construction is trying to spread the reshuffling cost over time; in reality, its operational mechanisms bear little resemblance to prior schemes [7,14,18] based on Goldreich and Ostrovsky's original hierarchical solution [6]. Therefore, this represents an entirely new technique which has not been previously studied in the O-RAM literature.

While the basic binary-tree based construction achieves poly-logarithmic amortized and worst-case cost, it requires $\frac{N}{c}$ blocks of client-side storage for some constant $c > 1$. To reduce the client-side storage, we recursively apply our O-RAM construction over the index structure. Instead of storing the index structure on the client side, we store it in a separate and smaller O-RAM on the server side. We achieve $O(1)$ client-side storage through recursive application of our O-RAM construction over the index structure (Section 4).

We offer three variants of our construction. The simpler variant (instantiated with the trivial bucket O-RAM) achieves $O((\log N)^3)$ amortized and worst-case cost. The second variant (instantiated with the Square-Root bucket O-RAM) achieves $\widetilde{O}((\log N)^{2.5})$ amortized cost, and $\widetilde{O}((\log N)^3)$ worst-case cost. The third variant utilizes the O-RAM scheme by Damgård, Meldgaard, and Nielsen [4] as the bucket O-RAM, and achieves $\widetilde{O}((\log N)^2)$ amortized cost, and $\widetilde{O}((\log N)^3)$ worst-case cost. We use the $\widetilde{O}$ notation

Scheme	Amortized Cost	Worst-case Cost	Client Storage	Server Storage
GO [6]	$O((\log N)^3)$	$O(N(\log N)^2)$	O(1)	$O(N\log N)$
OS [13]	$O((\log N)^3)$ (const > 6100)	$O((\log N)^3)$ (const > 6100)	$O(1)$	$O(N\log N)$
WS [18]	$O((\log N)^2)$	$O(N\log N)$	$O(\sqrt{N})$	$O(N\log N)$
WSC [19]	$O(\log N\log\log N)$	$O(N\log\log N)$	$O(\sqrt{N})$	$O(N)$
PR [14]	$O((\log N)^2)$	$O(N\log N)$	$O(1)$	$O(N)$
GM [7]	$O((\log N)^2)$ $O(\log N)$	$O(N\log N)$ $O(N)$	$O(1)$ $O(\sqrt{N})$	$O(N)$ $O(N)$
BMP [3]	$O(\sqrt{N})$	$O(\sqrt{N})$	$O(\sqrt{N})$	$O(N)$
SSS [17]	$O((\log N)^2)$	$O(\sqrt{N})$	$O(\sqrt{N})$	$O(N)$
This paper				
Trivial Bucket	$O((\log N)^3)$	$O((\log N)^3)$	$O(1)$	$O(N\log N)$
Square-Root Bucket	$\widetilde{O}((\log N)^{2.5})$	$\widetilde{O}((\log N)^3)$	$O(1)$	$O(N\log N)$
BST Bucket	$\widetilde{O}((\log N)^2)$	$\widetilde{O}((\log N)^3)$	$O(1)$	$\widetilde{O}(N\log N)$

Table 1: Comparison of various schemes. The $\widetilde{O}$ notation hides poly $\log\log N$ terms. The bounds for this paper hold with high probability $1 - \frac{1}{\text{poly}(N)}$, assuming that the total number of data access requests $M = \text{poly}(N)$, and that the block size $B \geq c\log N$ bits, for any constant $c > 1$. For a more precise statement of our bounds, please refer to Section 4. The BST bucket construction is due to an O-RAM construction by Damgård, Meldgaard, and Nielsen [4].

to hide poly $\log\log$ terms from the asymptotic bounds. These afore-mentioned bounds hold with very high probability (i.e., at least $1 - \frac{1}{\text{poly}(N)}$), under realistic assumptions that the number of data requests $M = \text{poly}(N)$, and that the block size $B \geq c\log N$ bits for any constant $c > 1$.

1.3 Related Work

Oblivious RAM was first investigated by Goldreich and Ostrovsky [5,6,12] in the context of protecting software from piracy, and efficient simulation of programs on oblivious RAMs. Apart from proposing a seminal hierarchical solution with $O((\log N)^3)$ amortized cost, Goldreich and Ostrovsky [6] also demonstrate the following lower-bound: for an O-RAM of capacity N, the client has to pay an amortized cost of at least $\Omega(\log N)$. Recently, Beame and Machmouchi [2] improved the lower bound to $\Omega(\log N\log\log N)$.

Since the first investigation of Oblivious RAM by Goldreich and Ostrovsky [5,6,12], several constructions have been proposed subsequently [3,7,14,17,18]. Among these, the seminal constructions recently proposed by Goodrich and Mitzenmacher [7] achieve the best asymptotic performance in terms of amortized cost: with $O(1)$ client-side storage, their construction achieves $O((\log N)^2)$ amortized cost; and with $O(\sqrt{N})$ client-side storage, their construction achieves $O(\log N)$ amortized cost [7]. Pinkas and Reinman [14] also showed a similar result for the $O(1)$ client-side storage case; however, some researchers have pointed out a security flaw in their construction [7], which the authors of [14] have promised to fix in a future journal version.

A few works on O-RAM achieve sublinear worst-case cost. In the seminal work by Ostrovsky and Shoup [13], they show how to spread the reshuffling operations over time, and achieve $O((\log N)^3)$ amortized cost. Boneh,Mazieres, and Popa [3] achieve $O(\sqrt{N})$ worst-case cost, however, at the expense of $O(\sqrt{N})$ amortized cost. Stefanov, Shi, and Song [17] recently proposed a novel O-RAM construction with

$O(\sqrt{N})$ worst-case cost, $O((\log N)^2)$ amortized cost, and $O(\sqrt{N})$ client-side storage. Apart from this, Stefanov, Shi, and Song also offered another construction geared towards practical performance rather than asymptotics. This practical construction uses linear amount of client storage (with a very small constant), and achieves $O(\log N)$ amortized cost and $O(\sqrt{N})$ worst-case cost. Under realistic settings, it achieves $20 - 30X$ amortized cost, while storing $0.01\% - 0.3\%$ amount of total data at the client.

We note that the hierarchical aspect of our binary-tree technique is partially inspired by the hierarchical solution originally proposed by Goldreich and Ostrovsky [6], and later adopted in many constructions [7, 14, 18]; while the eviction aspect is partially inspired by the background eviction idea originally proposed by Stefanov, Shi, and Song [17].

Our binary tree technique may also be superficially reminiscent of a construction by Damgård, Meldgaar, and Nielsen [4]. However, apart from that fact that both schemes rely on a binary tree, the internal mechanisms of our construction and the Damgård-Meldgaar-Nielsen construction are fundamentally different. Specifically, Damgård *et al.* primarily aim to avoid the need of random oracle or pseudo-random function, rather than improve worst-case cost. Their construction uses a binary search tree, and requires periodic reshuffling operations that can take $O(N \log N)$ time. In contrast, we use a binary tree (instead of a binary search tree), and we use a background eviction mechanism to circumvent the need for reshuffling.

Table 1 illustrates the asymptotic performance characteristics of various existing schemes, and positions our work in perspective of related work.

Concurrent/subsequent work. In concurrent/subsequent work, Goodrich *et al.* [8] and Kushilevitz *et al.* [11] also invented novel O-RAM constructions with poly-logarithmic worst-case overhead. Specifically, the construction by Goodrich *et al.* achieves $O((\log N)^2)$ worst-case cost with $O(1)$ memory [8]; and and Kushilevitz *et al.* [11] achieve $O(\frac{(\log N)^2}{\log \log N})$ worst-case cost. Goodrich *et al.* also came up with a stateless Oblivious RAM [9] scheme, with $O(\log N)$ amortized cost and $O(N^a)$ $(0 < a < 1)$ client-side transient (as opposed to permanent) buffers. Due to a larger constant in their asymptotic notations, in realistic scenarios, our scheme with the trivial bucket O-RAM is likely the most practical when the client-side storage is $O(1)$.

2 Preliminaries

Let N denote the O-RAM capacity, i.e., the maximum number of data blocks that an O-RAM can store. We assume that data is fetched and stored in atomic units called *blocks*. Let B denote the block size in terms of the number of bits. We assume that the block size $B \geq c \log N$, for some $c > 1$. Notice that this is true in almost all practical scenarios. We assume that each block has a global identifier $\mathsf{u} \in \mathcal{U}$, where $\mathcal{U}$ denotes the universe of identifiers.

Throughout the paper, we use the asymptotic notation $\widetilde{O}(f(N))$ meaning $O(f(N) \text{poly} \log \log N)$ as a short-hand for hiding poly $\log \log N$ terms.

2.1 Defining O-RAM with Enriched Operations

The standard O-RAM adopted in prior work [5, 7, 14, 18] exports a Read and a Write interfaces. To hide whether the operation is a read or a write, either operation will generate both a read and a write to the O-RAM.

In this paper, we consider O-RAMs that support a few enriched operations. Therefore, we propose a modified O-RAM definition, exporting a ReadAndRemove primitive, and an Add primitive. We later show that given these two primitives, we can easily implement the standard O-RAM Read and Write operations. Moreover, given these two primitives, we can also support an enriched operation called Pop, which will be later needed in our constructions. Therefore, our modified O-RAM definition is more general than the standard O-RAM notion. The same modified O-RAM notion was adopted in the work by Stefanov, Shi, and Song [17].

Definition 1. *An Oblivious RAM (with enriched operations) is a suite of interactive protocols between a client and a server, comprising the following:*

$\mathsf{ReadAndRemove}(\mathsf{u})$: Given a private input $\mathsf{u} \in \mathcal{U}$ which is a block identifier, the client performs an interactive protocol with the server to retrieve a block identified by u, and then remove it from the O-RAM. If u exists in the O-RAM, the content of the block data is returned to the client. Otherwise, $\perp$ is returned.

$\mathsf{Add}(\mathsf{u}, \mathsf{data})$: The client is given private inputs $\mathsf{u} \in \mathcal{U}$ and $\mathsf{data} \in \{0, 1\}^B$, representing a block identifier and some data content respectively. *This operation must be immediately preceded by* $\mathsf{ReadAndRemove}(\mathsf{u})$ *such that block* u *no longer resides in the O-RAM.* The client then performs an interactive protocol with the server to write content data to the block identified by u, which is added to the O-RAM.

Definition 2 (Security definition)**.** *Let* $\vec{y} := ((\mathsf{op}_1, \mathsf{arg}_1), (\mathsf{op}_2, \mathsf{arg}_2), \ldots, (\mathsf{op}_M, \mathsf{arg}_M))$ *denote a data request sequence of length* M. *Each* op_i *denotes a* ReadAndRemove *or an* Add *operation. Moreover, if* op_i *is a* ReadAndRemove *operation, then* $\mathsf{arg}_i = \mathsf{u}_i$*, else if* op_i *is an* Add *operation, then* $\mathsf{arg}_i = (\mathsf{u}_i, \mathsf{data}_i)$*, where* u_i *denotes the identifier of the block being read or added, and* data_i *denotes the data content being written in the second case. Recall that if* op_i *is an* Add *operation with argument* $(\mathsf{u}_i, \mathsf{data}_i)$*, then* op_{i-1} *must be a* ReadAndRemove *operation with argument* $\mathsf{u}_{i-1} = \mathsf{u}_i$.

We use the notation $\mathsf{ops}(\vec{y})$ *to denote the sequence of operations associated with* $\vec{y}$*, i.e.,* $\mathsf{ops}(\vec{y}) := (\mathsf{op}_1, \mathsf{op}_2, \ldots, \mathsf{op}_M)$.

Let $A(\vec{y})$ *denote the (possibly randomized) sequence of accesses to the remote storage given the sequence of data requests* $\vec{y}$. *An O-RAM construction is said to be secure if for any two data request sequences* $\vec{y}$ *and* $\vec{z}$ *such that* $|\vec{y}| = |\vec{z}|$*, and* $\mathsf{ops}(\vec{y}) = \mathsf{ops}(\vec{z})$*, their access patterns* $A(\vec{y})$ *and* $A(\vec{z})$ *are computationally indistinguishable by anyone but the client.*

2.2 Relationship with the Standard O-RAM Definition

As mentioned earlier, our modified O-RAM notion is more general than the standard O-RAM notion, in the sense that given a modified O-RAM exporting ReadAndRemove and Add primitives, we can easily implement a standard O-RAM supporting Read and Write operations, as stated in the following observation.

Observation 1. *Given a modified O-RAM as defined above, we can construct a standard O-RAM, where a standard* $\mathsf{Read}(\mathsf{u})$ *operation is implemented by the operation* $\mathsf{data} \leftarrow \mathsf{ReadAndRemove}(\mathsf{u})$ *followed by* $\mathsf{Add}(\mathsf{u}, \mathsf{data})$*, and a standard* $\mathsf{Write}(\mathsf{u}, \mathsf{data})$ *operation is implemented by the operation* $\mathsf{data}_0 \leftarrow \mathsf{ReadAndRemove}(\mathsf{u})$ *followed by* $\mathsf{Add}(\mathsf{u}, \mathsf{data})$ *operation.*

Most existing constructions [6, 7, 18] based on Goldreich and Ostrovsky's hierarchical solution [6] can be easily modified to support the ReadAndRemove and Add primitives.

2.3 Implementing Enriched Semantics

Implementing the Pop **operation from the** ReadAndRemove **and** Add **primitives.** As mentioned earlier, our O-RAM storage is organized into a binary tree over buckets, where each bucket is a fully functional O-RAM by itself, referred to as a *bucket O-RAM*. For technical reasons which will become clear in Section 3, each bucket O-RAM needs to support not only the ReadAndRemove and Add operations (and hence the standard O-RAM Read and Write operations), but also a special-purpose operation called Pop().

The Pop() operation looks up a real data block and removes it from the O-RAM if one exists. Otherwise, it returns a dummy block $\perp$.

In our online full technical report [16], we present a constructive proof demonstrating that any O-RAM supporting the ReadAndRemove and Add primitives can be modified to support the Pop primitive as well; and the Pop operation costs asymptotically the same as the basic ReadAndRemove and Add primitives. We state this fact in the following lemma.

Lemma 1 (Additional Pop() operation)**.** *Given any O-RAM construction of capacity $3N$ satisfying Definition 1, one can construct a new O-RAM of capacity N that not only provides a* ReadAndRemove(u) *and an* Add(u, data) *primitives (and hence, the standard* Read(u) *and* Write(u, data) *operations), but also provides a* Pop() *operation, where all operation preserve the asymptotic performance of the original O-RAM. Specifically, the* Pop() *operation selects an arbitrary block that currently exists in the O-RAM, reads it back and removes it from the O-RAM. If the O-RAM does not contain any real blocks, the* Pop *operation returns* $\perp$.

2.4 Encryption and Authentication

Similar to prior work in O-RAM [6, 7, 14, 18], we assume that all data blocks are encrypted using a semantically secure encryption scheme, so that two encryptions of the same plaintext cannot be linked. Furthermore, every time a data block is written back it is encrypted again using fresh randomness.

We also assume that the server does not tamper with or modify the data, since authentication and freshness can be achieved using standard techniques such as Message Authentication Codes (MAC), digital signatures, or authenticated data structures.

2.5 Two Simple O-RAM Constructions with Deterministic Guarantees

As mentioned earlier, our O-RAM storage is organized into a binary tree over small data buckets, where each bucket is a fully functional O-RAM by itself, referred to as a bucket O-RAM.

For technical reasons which will become clear in Section 3, we would like each bucket O-RAM to provide *deterministic* (as opposed to high probability) guarantees. Moreover, each bucket O-RAM needs to support *non-contiguous* block identifier space. We consider each block identifier $\mathsf{u} \in \{0,1\}^{\leq B}$, i.e., u can be an arbitrary string, as long as u can be described within one block. Furthermore, the set of block identifiers is unknown in advanced, but rather, determined dynamically during live operations of the bucket O-RAM. As long as the load of the bucket O-RAM never exceeds its capacity, the correct functioning of the bucket O-RAM should be guaranteed.

Below, we present the two candidate bucket O-RAMs constructions, called the trivial O-RAM and the Square-Root O-RAM respectively. They are modifications of the trivial O-RAM and the Square-Root O-RAM constructions originally proposed by Goldreich and Ostrovsky [6].

Trivial O-RAM. We can build a trivial O-RAM supporting non-contiguous block identifier space in the following way. Let N denote the O-RAM capacity. In the trivial O-RAM, the server side has a buffer storing N blocks, where each block is either a real block denoted $(\mathsf{u}, \mathsf{data})$, or a dummy block denoted $\perp$.

To perform a ReadAndRemove(u) operation, a client sequentially scans positions 0 through $N-1$ in the server array: if the current block matches identifier u, the client remembers its content, and overwrites it with $\perp$; if the current block does not match identifier u, the client writes back the original block read.

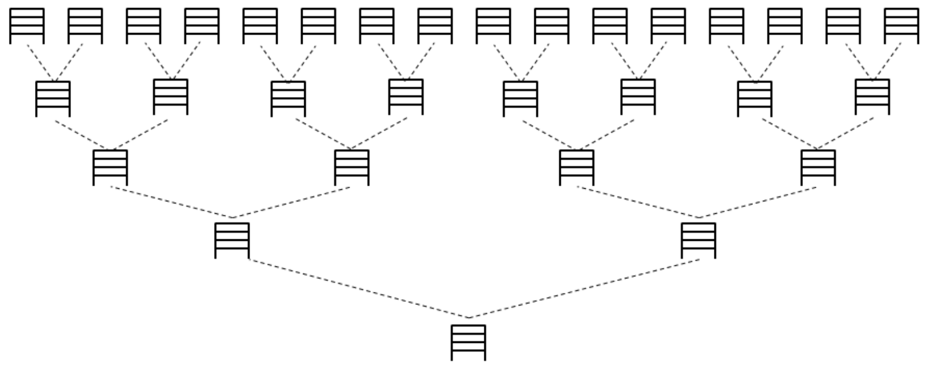

Figure 1: **Server-side storage hierarchy.** The server-side O-RAM storage is organized into a binary tree over data buckets, where each bucket can hold up to $O(\log N)$ data blocks. A data block enters from the root bucket when written to the O-RAM, and then obliviously percolates down towards a random leaf over time, until the same block is accessed again.

To perform an $\mathsf{Add}(\mathsf{u}, \mathsf{data})$ operation, a client sequentially scans positions 0 through $N-1$ in the server buffer: the first time the client sees a dummy block, the client overwrites it with $(\mathsf{u}, \mathsf{data})$; otherwise, the client writes back the original block read.

As mentioned earlier, whenever blocks are written back to the server, they are re-encrypted in order to hide its contents from the server.

Clearly, the trivial O-RAM is secure, requires $O(N)$ amortized and worst-case cost, $O(N)$ server-side storage, and $O(1)$ client-side storage (since the client never downloads the entire array all at once, but performs the reads and updates in a streaming fashion).

Square-Root O-RAM [6]. Goldreich and Ostrovsky present a Square-Root O-RAM [6] which achieves $O(\sqrt{N}\log N)$ amortized cost, $O(N\log N)$ worst-case cost, $O(N)$ server-side storage, and $O(1)$ client-side storage. When using the deterministic AKS sorting network [1] to implement the reshuffling operation, the Square-Root O-RAM achieves deterministic (as opposed to high probability) guarantees. Although the original Square-Root O-RAM construction supports only contiguous block identifier space, it is not too difficult to modify it to support non-contiguous block identifier space, while preserving the same asymptotic performance. We defer the detailed description of this modified Square-Root O-RAM construction to our online full version [16].

3 Basic Construction

3.1 Overview of the Binary Tree Construction

We first describe a binary-tree based construction, which has two variants. The first variant makes use of the trivial bucket O-RAM and has amortized and worst case cost $O((\log N)^2)$; the second variant makes use of the Square-Root bucket O-RAM and has $\widetilde{O}((\log N)^{1.5})$ amortized cost, and $\widetilde{O}((\log N)^2)$ worst-case cost. Both variants require $\frac{N}{c}$ client-side storage, where $c > 1$ and we assume that the failure probability is $\frac{1}{\text{poly}(N)}$ and the number of operations is $M = \text{poly}(N)$, which is reasonable in practice (for instance $N = 10^6$ and $M = N^3 = 10^{18}$). Later, in Section 4, we describe how to apply our O-RAM construction

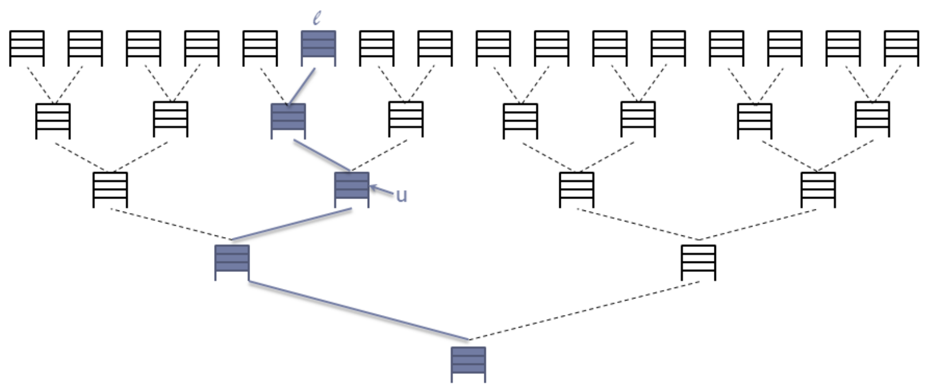

Figure 2: **Searching for a data block.** A block u is logically associated with a leaf node ℓ at a given point time. To look up the block u, it suffices to search every bucket on the path from the leaf bucket ℓ to the root bucket (denoted by the shaded buckets in this figure). Every time a block is accessed, it will be logically assigned to a fresh random leaf node.

recursively for the client-side storage, to achieve $O(1)$ client-side memory, while incurring a multiplicative factor of $O(\log N)$ to the amortized and worst-case costs.

As mentioned in Section 1, the motivation for the binary tree construction is to "in spirit" spread across time the reshuffling operations that commonly appear in existing constructions [5, 7, 14, 18]. However, since there is no trivial way to modify existing schemes to spread the reshuffling operation, we introduce a completely new technique based on the binary tree idea.

Server-side storage organization. In our construction, the server-side storage is organized into a binary tree of depth $D := \lceil \log_2 N \rceil$. For ease of explanation, let us assume that N is a power of 2 for the time being. In this way, there are exactly N leaf nodes in the tree.

Each node in the tree is a data bucket, which is a self-contained O-RAM of capacity $O(\log N)$, henceforth referred to as a *bucket O-RAM*. For technical reasons described later, each bucket O-RAM must have the following properties: (a) support non-contiguous identifier space, (b) support ReadAndRemove and Add primitives – from which we can also implement Read, Write, and Pop primitives as mentioned in Section 2, (c) has zero failure probability.[1]

There are two possible candidates for the bucket O-RAM, both of which are modifications of simple O-RAM constructions initially proposed by Goldreich and Ostrovsky [6], and described in more detail in Section 2.5.

1. **Trivial O-RAM.** Every operation is implemented by a sequential scan of all blocks in the server-side storage. For capacity L, the server-side storage is $O(L)$ and the cost of each operation (both amortized and worst-case) is $O(L)$.
2. **Square-Root O-RAM [6].** For capacity L, the Square-Root O-RAM achieves $O(L)$ server-side storage, $O(1)$ client-side storage, $O(\sqrt{L} \log L)$ amortized cost, and $O(L \log L)$ worst-case cost.

O-RAM operations. When data blocks are being written to the O-RAM, they are first added to the bucket at the root of the tree. As more data blocks are being added to a bucket, the bucket's load will increase. To avoid overflowing the capacity of a bucket O-RAM, data blocks residing in any non-leaf bucket are

[1]It would also be acceptable if a failure probability δ per operation would only incur a multiplicative factor of $O(\log \log \frac{1}{\delta})$ in the cost.

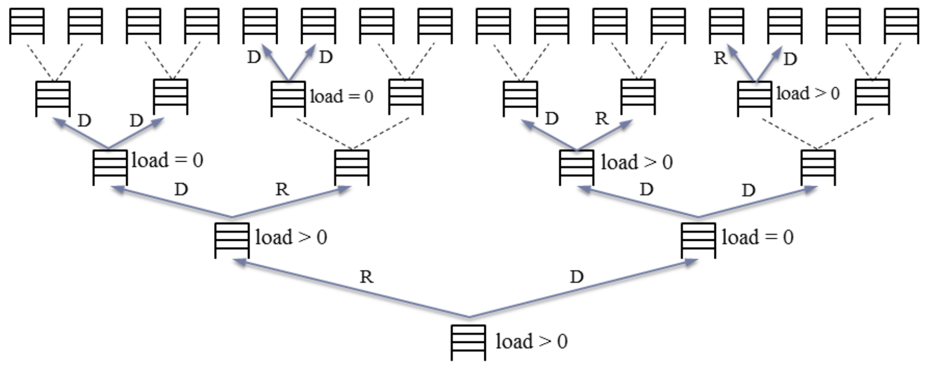

Figure 3: **Background evictions with eviction rate $\nu = 2$.** Upon every data access operation, for each depth in the hierarchy, ν number of buckets are chosen randomly for eviction during which one data block (real or dummy) will be evicted to each of its children. If the bucket is loaded, then one real block and one dummy block are evicted. If the bucket is not loaded, two dummy blocks are evicted. In this figure, D denotes the eviction of a dummy block, and R denotes the eviction of a real block.

periodically evicted to its children buckets. More specifically, eviction is an oblivious protocol between the client and the server in which the client reads data blocks from selected buckets and writes each block to a child bucket.

Over time, each block will gradually percolate down a path in the tree towards a leaf bucket, until the block is read or written again. Whenever a block is being added to the root bucket, it will be logically assigned to a random leaf bucket, indexed by a string in $\{0,1\}^D$. Henceforth, this data block will gradually percolate down towards the designated leaf bucket, until the same data block is read or written again.

Suppose that at some point, a data block is currently logically assigned to leaf node $\ell \in \{0,1\}^D$. This means that a fresh copy of the data block exists somewhere along the path from the leaf node ℓ to the root. To find that data block, it suffices to search the data block in all buckets on the path from the designated leaf node to the root. We assume that when the data block is stored in a bucket, we store the tag ℓ along as well and we denote the block's contents by $(\texttt{data}||\ell)$.

Ensuring security. For security reasons, it is important to ensure the following:

- *Every time a block is accessed, its designated leaf node must be chosen independently at random.* This is necessary to ensure that two operations on the same data block are completely unlinkable.

- *The bucket sequence accessed during eviction process must reveal no information about the load of each bucket, or the data access sequence.* In our construction, the choice of which buckets to evict from is randomly selected, and independent from the load of the bucket, or the data access sequence. Furthermore, whenever a bucket is selected for eviction, we always write to both of its children – depending on whether there are real blocks to evict, we would write a real or a dummy block to each of its children.

Client-side index. As each data block will be logically assigned to a random leaf node every time it is operated on, we need some data structure to remember where each block might be at any point of time. For this reason, the client stores a data structure of size $\frac{N \log N}{B}$ blocks, in which it records which leaf node is currently associated with each block. When $B \geq c \log N$, this index structure's size is a linear fraction of the capacity of the O-RAM. Therefore, in the basic scheme, we require $\frac{N}{c}$ client-side storage, where $c > 1$.

However, later in the recursive construction described in Section 4, we show how to apply our O-RAM construction recursively over the index structure to achieve $O(1)$ client-side storage.

A note about dummy blocks and dummy operations. To ensure the security of the O-RAM, in our construction, we often rely on *dummy blocks* and *dummy operations* to hide certain information from the untrusted server, such as whether a bucket is loaded, and where in the tree a block is headed.

For the purpose of this section, we adopt the following notion of dummy blocks and dummy operations. We will think of the dummy block as a regular but useless data block. We can dedicate a certain block identifier, e.g., $\mathsf{u} = 0$ to serve as the dummy block. In this way, we simply deduct 1 from the O-RAM capacity, which does not affect the asymptotics. In our construction, every bucket may have a dummy block; while each real data block exists in at most one bucket.

Given the above notion of the dummy block, we can define a dummy O-RAM operation as a regular operation on the dedicated dummy block with $\mathsf{u} = 0$. A dummy O-RAM operation serves no purpose other than ensuring the security of the O-RAM. Henceforth, with a slight abuse of notation, we use the symbol $\perp$ to denote a dummy data block or its identifier. We use the notations $\mathsf{ReadAndRemove}(\perp)$, $\mathsf{Add}(\perp)$, $\mathsf{Read}(\perp)$ and $\mathsf{Write}(\perp)$ to denote dummy O-RAM operations.

3.2 Detailed Construction

We define some notations in Table 2 which will be useful in the formal algorithm descriptions.

Table 2: Notations.

D	$\lceil \log_2 N \rceil$
$\mathsf{u} \in \{0, 1, \ldots, N-1\}$	global identifier of a block
index	client's index structure
$\mathsf{index}[\mathsf{u}] \in \{0,1\}^D$	id of leaf node associated with block u, initially random
state	global variable to avoid unnecessary index lookup
root	root bucket of the binary tree
$\mathcal{P}(\ell)$	path from the leaf node ℓ to the root
$\mathsf{Child}_b(\mathsf{bucket})$, for $b \in \{0,1\}$	the left or right child of a bucket
ν	eviction rate
$\mathsf{UniformRandom}(S)$	Samples an element uniformly at random from the set S
$\mathsf{UniformRandom}_\nu(S)$	Samples a subset of size ν uniformly at random from the set S
$\perp$	a dummy block or the identifier of a dummy block

$\mathsf{ReadAndRemove}$ **operation.** The algorithm for performing a $\mathsf{ReadAndRemove}(\mathsf{u})$ operation is described in Figure 4. First, the client looks up its local index structure index to find out which leaf node ℓ the requested block u is associated with. We then generate a fresh random ℓ^* from $\{0,1\}^D$ and overwrite $\mathsf{index}[\mathsf{u}] \leftarrow \ell^*$, i.e., block u is henceforth associated with a fresh random leaf node ℓ^*. Notice that this ensures no linkability between two operations on the same data block. In order to avoid extra index lookup for any following Add operation, ℓ^* is also stored in a global variable state.

Now, given that u is currently associated with leaf node ℓ, it means that a fresh copy of block u must reside in some bucket along the along the path from leaf ℓ to the root, denoted by $\mathcal{P}(\ell)$. If u is found in some bucket, we remove u from that bucket, and remember its the data content. Regardless of whether u has been

```
ReadAndRemove(u):
1: $\ell^* \leftarrow \mathsf{UniformRandom}(\{0,1\}^D)$
2: $\ell \leftarrow \mathsf{index}[\mathsf{u}]$, $\mathsf{index}[\mathsf{u}] \leftarrow \ell^*$
3: $\mathsf{state} \leftarrow \ell^*$   //If an Add operation follows, $\ell^*$ will be used by Add
4: $\mathsf{data} \leftarrow \perp$
5: for each bucket on $\mathcal{P}(\ell)$ do   //path from leaf $\ell$ to root
6:     if $((\mathsf{data}_0||\ell_0) \leftarrow \mathsf{bucket.ReadAndRemove}(\mathsf{u})) \neq \perp$ then
7:         $\mathsf{data} \leftarrow \mathsf{data}_0$   //Notice that $\ell = \ell_0$
8:     end if
9: end for
10: return data
```

```
Add(u, data):
1: $\ell \leftarrow \mathsf{state}$
2: $\mathsf{root.Write}(\mathsf{u}, \mathsf{data}||\ell)$   // Root bucket's O-RAM Write operation
3: Call $\mathsf{Evict}(\nu)$
4: return data
```

Figure 4: **Algorithms for data access**.

found, we always continue our search all the way to the root. Note that to ensure obliviousness, it is important that the search does not abort prematurely even after finding block u. Finally, if the requested block u has been found, the ReadAndRemove algorithm returns its data contents; otherwise, the ReadAndRemove algorithm returns $\perp$.

Add **operation.** Also shown in Figure 4, the $\mathsf{Add}(\mathsf{u}, \mathsf{data})$ operation reads the tag ℓ from state, which was just generated by the preceding $\mathsf{ReadAndRemove}(\mathsf{u})$ operation. The client writes the intended block $(\mathsf{u}, \mathsf{data}||\ell)$ to the root bucket.

Notice that here the client tags the data with ℓ, i.e., the id of the leaf node that block u would be logically associated with until the next operation on block u. The designated leaf node tag will become important when we recursively apply our O-RAM over the client's index structure, as described in Section 4. Specifically, the eviction algorithm will examine this designated leaf node tag to determine to which child node to evict this block. Observe that to preserve the desired asymptotics in the recursive construction, the eviction algorithm cannot afford to (recursively) look up the index structure to find the designated leaf node for a block. By tagging the data with its designated leaf, the eviction algorithm need not perform recursive lookups to the index structure.

Finally, at the end of every Add operation, the client invokes the background eviction process once. We now describe the background eviction algorithm.

Background evictions. Let ν denote the eviction rate. For the purpose of our asymptotic analysis, it suffices to let $\nu = 2$.

Whenever the background eviction algorithm is invoked, the client randomly selects ν buckets to evict at every depth of the tree.

If a bucket is selected for eviction, the client pops a block from the bucket O-RAM by calling the Pop operation (see Section 2.3 for how to implement the Pop operation given an O-RAM that supports

```
Evict($\nu$):
1: for $d = 0$ to $D - 1$ do
2:     Let $S$ denote the set of all buckets at depth $d$.
3:     $A \leftarrow \mathsf{UniformRandom}_\nu(S)$
4:     for each bucket $\in A$ do
5:         $(\mathsf{u}, \mathsf{data}||\ell) \leftarrow \mathsf{bucket.Pop}()$
6:         $b \leftarrow (d{+}1)$-st bit of $\ell$
7:         $\mathsf{block}_b \leftarrow (\mathsf{u}, \mathsf{data}||\ell),\ \mathsf{block}_{1-b} \leftarrow \perp$
8:         $\forall b \in \{0,1\} : \mathsf{Child}_b(\mathsf{bucket}).\mathsf{Write}(\mathsf{block}_b)$
9:     end for
10: end for
```

Figure 5: **Background eviction algorithm with eviction rate ν.**

ReadAndRemove and Write operations). If the bucket selected for eviction is loaded, then the Pop operation returns a real block and removes that block from the bucket O-RAM; otherwise, if the bucket is not loaded, the Pop operation returns a dummy block $\perp$.

Regardless of whether a real block or a dummy block is returned by the Pop operation, the client always performs a write to both children of the selected bucket:

1. If a dummy block is returned by Pop, the client simply performs a dummy write to both children buckets.
2. If a real block is returned, the client examines its designated leaf node tag to figure out the correct child node to evict this block to. Recall that this designated leaf node tag is added when the block is first written to the root bucket. (Note that although in the basic construction, the client can alternatively find out this information by looking up its local index structure; later in the recursive construction, the client will have to obtain this information through the designated leaf node tag.)
   Now, suppose that the block should be evicted to child $b \in \{0,1\}$ of the selected bucket, the client then writes the block to child b, and writes a dummy block to child $1 - b$.

Regardless of which case, to ensure obliviousness, the two writes to the children nodes must proceed in a predetermined order, e.g., first write a real or dummy block to child 0, and then write a real or dummy block to child 1.

3.3 Security Analysis

Theorem 1 (Security of Basic Construction)**.** *Our Basic O-RAM Construction is secure in the sense of Definition 2, assuming that each bucket O-RAM is also secure.*

Proof. Observe that each bucket is itself a secure O-RAM. Hence, it suffices to show that each type of operation induces independently the same distribution on the access patterns of the buckets in the binary tree, regardless of the arguments.

For the ReadAndRemove(u) operation, the buckets along the path $\mathcal{P}(\ell)$ from the root to the leaf indexed by $\ell = \mathsf{index}(\mathsf{u})$ are accessed. Observe that ℓ is generated uniformly at random from $\{0,1\}^D$. Hence, the distribution of buckets accessed is the buckets along the path to a random leaf. Moreover, each time ReadAndRemove(u) is called, a fresh random ℓ^* is generated to be stored in index(u) so that the next invocation of ReadAndRemove(u) will induce an independent random path of buckets.

For the $\mathsf{Add}(\mathsf{u}, \mathsf{data})$ operation, the root bucket is always accessed. More buckets are accessed in the Evict subroutine. However, observe that the access pattern of the buckets are independent of the configuration of the data structure, namely two random buckets at each depth (other than the leaves) are chosen for eviction, followed by accesses to both child buckets. $\square$

3.4 Asymptotic Performance of the Basic Construction

We next analyze the server-side storage and the cost of each operation. If the capacity of each bucket is L, the server-side storage is $O(NL)$, because there are $O(N)$ buckets. If we use the trivial bucket O-RAM, each operation has cost $O(L \log N)$. If we use the Square-Root bucket O-RAM, each operation has amortized cost $O(\sqrt{L} \log L \log N)$ and worst case cost $O(L \log L \log N)$.

We prove the following lemma in Appendix A.

Lemma 2 (Each Bucket Has Small Load). *Let $0 < \delta < \frac{1}{2^{2e}}$. For a fixed time and a fixed bucket, the probability that the bucket has load more than $\log_2 \frac{1}{\delta}$ is at most δ.*

Applying Union Bound on Lemma 2 over all buckets and over all time steps, we have the following result.

Lemma 3 (Bucket Overflow). *Suppose $0 < \delta < 1$ and $N, M \geq 10$. Then, one can use bucket O-RAM with capacity $O(\log \frac{MN}{\delta})$ such that with probability at least $1 - \delta$, the Basic O-RAM Construction can support M operations without any bucket overflow.*

Lemma 3 gives an upper bound on the capacity of each bucket and from the above discussion, we have the following result.

Corollary 1. *The Basic O-RAM Construction can support M operations with failure probability at most δ using $O(N \log \frac{MN}{\delta})$ server-side storage and $O(\frac{N \log N}{B})$ client-side storage. The cost of each operation is as follows:*

Bucket O-RAM	*Amortized*	*Worst-case*
Trivial	$O(\log N \log \frac{MN}{\delta})$	$O(\log N \log \frac{MN}{\delta})$
Square-Root	$O(\log N \sqrt{\log \frac{MN}{\delta}} \log \log \frac{MN}{\delta})$	$O(\log N \log \frac{MN}{\delta} \log \log \frac{MN}{\delta})$

Specifically, if the number of data access requests $M = poly(N)$, then the basic construction with the trivial bucket O-RAM achieves $O((\log N)^2)$ amortized and worst-case cost; and the basic construction with the Square-Root bucket O-RAM achieves $\widetilde{O}((\log N))^{1.5}$ amortized cost, and $\widetilde{O}((\log N)^2)$ worst-case cost. Furthermore, no buckets will overflow with probability $1 - \frac{1}{poly(N)}$.

4 Recursive Construction and How to Achieve the Desired Asymptotics

The basic construction described in Section 3 achieves poly-logarithmic amortized and worst-case cost, but requires $\frac{N}{c}$ client-side storage, where $c = \frac{B}{\log N} > 1$.

In this section, we demonstrate how to recursively apply our O-RAM construction to the client's index structure to achieve $O(1)$ client-side storage, while incurring an $O(\log N)$ multiplicative factor in terms of the amortized and worst-case cost.

4.1 Recursive O-RAM Construction: $O(1)$ Client-side Storage

Storing the index through recursion. In the basic construction, the client's index structure takes up at most $\frac{N \log N}{B} \leq \frac{N}{c}$ space, where $B \geq c \log N$. To achieve $O(1)$ client-side storage, we recursively apply our O-RAM over the index structure. Instead of storing the index structure on the client, we store the index structure in a separate O-RAM on the server side. At each step of the recursion, we effectively compress the O-RAM capacity by a factor of $c > 1$. Therefore, after $\log_c N$ levels of recursion, the index structure will be reduced to constant size.

To see how the recursion can be achieved, notice that Line 2 of the ReadAndRemove algorithm in Figure 4 can be replaced with a recursive O-RAM operation:

$$\mathsf{O\text{-}RAM.Write}(\mathsf{block\_id}(\mathsf{index}[\mathsf{u}]), \ell^*)$$

Here we have a slight abuse of notation, because in reality, the entry index[u] (stored sequentially according to u) resides in a larger block identified by block_id(index[u]), and one would have to first read that block, update the corresponding entry with ℓ^*, and then write the updated block back.

Theorem 2 (Recursive O-RAM Construction)**.** *The Recursive O-RAM Construction can support M operations with failure probability at most δ using $O(N \log \frac{MN}{\delta})$ server-side storage and $O(1)$ client-side storage, and the cost of each operation is as follows:*

Bucket ORAM	*Amortized*	*Worst-case*
Trivial	$O(\log_c N \log N \log \frac{MN}{\delta})$	$O(\log_c N \log N \log \frac{MN}{\delta})$
Square-Root	$O(\log_c N \log N \sqrt{\log \frac{MN}{\delta}} \log\log \frac{MN}{\delta})$	$O(\log_c N \log N \log \frac{MN}{\delta} \log\log \frac{MN}{\delta})$

Specifically, if the number of data access requests $M = poly(N)$, then the recursive construction with the trivial bucket O-RAM achieves $O((\log N)^3)$ amortized and worst-case cost; and the recursive construction with the Square-Root bucket O-RAM achieves $\widetilde{O}((\log N))^{2.5}$ amortized cost, and $\widetilde{O}((\log N)^3)$ worst-case cost. Furthermore, no buckets will overflow with probability $1 - \frac{1}{poly(N)}$.

Proof. The $O(1)$ client-side storage is immediate, due to the fact that all client-side storage (including the state variable in Figure 4, and the shuffling buffer for the Square-Root bucket O-RAM) is transient state rather than persistent state, and therefore, all levels of recursion can share the same $O(1)$ client-side storage.

Observe that for each $j = 0, 1, \ldots, \lceil \log_c N \rceil$, the jth recursion produces a binary tree with $O(\frac{N}{c^j})$ buckets. Hence, there are totally $O(\sum_{j \geq 0} \frac{N}{c^j}) = O(N)$ buckets.

Recall that by Theorem 3, for each bucket and at the end of each operation, with probability at least η, the load of the bucket is at most $\log_2 \frac{1}{\eta}$. Since there are $O(N)$ buckets and M operations, we need to set $\eta = \Theta(\frac{\delta}{NM})$ to apply the Union Bound such that the overall failure probability (due to bucket overflow) is at most δ. It follows that the capacity of each bucket is $L = O(\log \frac{MN}{\delta})$. and hence the server-side storage is $O(NL) = O(N \log \frac{MN}{\delta})$.

Moreover, each operation on the Recursive O-RAM induces $O(\log \frac{N}{c^j})$ operations on the bucket O-RAMs in the jth binary tree. Hence, the total number of bucket O-RAM accesses is $Z = O(\sum_{j \geq 0} \log \frac{N}{c^j}) = O(\log_c N \log N)$.

If we use the trivial bucket O-RAM, each operation has cost $O(ZL)$.

If we use the Square-Root bucket O-RAM, the amortized cost is $O(Z\sqrt{L} \log L)$ and the worst-case cost is $O(ZL \log L)$, as required. □

Remark 1. *Observe that the BST O-RAM construction by Damgård, Meldgaard, and Nielsen [4] for capacity L has client storage $O(1)$, server storage $O(L \log L)$, amortized cost $O((\log L)^a)$ and worst-case cost*

$O((\log L)^b)$, where a and b are small integers. Hence, if we use the BST construction for out bucket O-RAM, the amortized cost of our binary scheme can be improved to $O(\log_c N \log N (\log \frac{MN}{\delta})^a) = \widetilde{O}((\log N)^2)$ and the worst-case cost to $O(\log_c N \log N \log \frac{MN}{\delta} (\log\log \frac{MN}{\delta})^b) = \widetilde{O}((\log N)^3)$, where $M = poly(N)$ and $\delta = \frac{1}{poly(N)}$, while the server storage cost is $\widetilde{O}(N \log N)$.

5 Experiments

We built a simulator of our basic construction. For different values of N, we simulated $2000N$ O-RAM operations and the simulator kept track of each bucket's load at all times. We then calculated the maximum load of a bucket across all levels and all $2000N$ operations. The results are given in Figure 6. The plot confirms our theoretic analysis that the maximum bucket load is upper-bounded by $O(\log N)$.

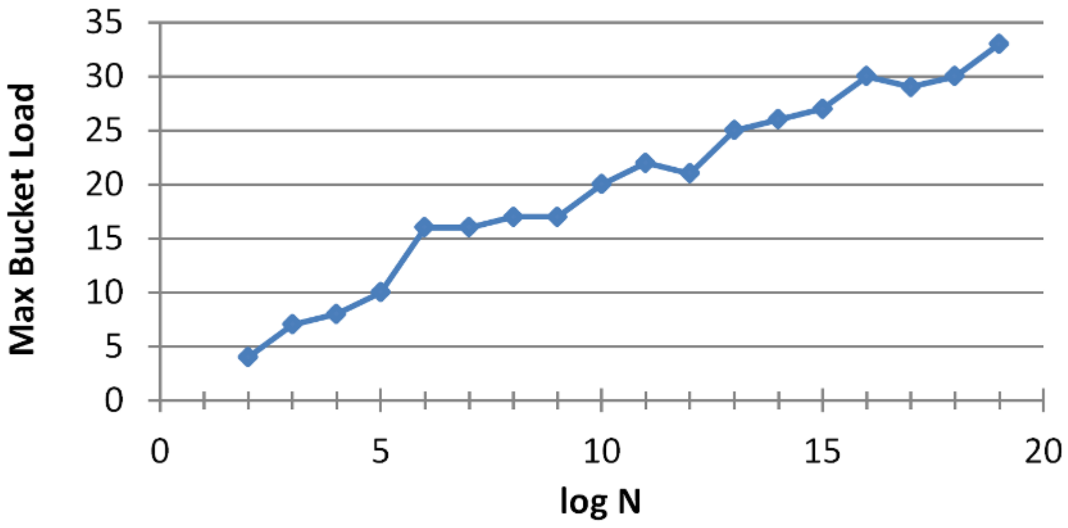

Figure 6: **Max bucket load with $2000N$ operations for our basic construction.**

Acknowledgments

We would like to thank Professor Mor Harchol-Balter for her amazing lectures and lecture notes on queuing theory, which was of essential help in our proofs. We also would like to thank the anonymous reviewers for their insightful feedback.

This material is based upon work partially supported by the Air Force Office of Scientific Research under MURI Grant No. 22178970-4170 and No. FA9550-08-1-0352, and by the National Science Foundation Graduate Research Fellowship under Grant No. DGE-0946797. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the funding agencies.

References

[1] M. Ajtai, J. Komlós, and E. Szemerédi. Sorting in c log n parallel steps. *Combinatorica*, 3:1–19, January 1983.

[2] P. Beame and W. Machmouchi. Making rams oblivious requires superlogarithmic overhead. *Electronic Colloquium on Computational Complexity (ECCC)*, 17:104, 2010.

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

[4] I. Damgård, S. Meldgaard, and J. B. Nielsen. Perfectly secure oblivious ram without random oracles. In *TCC*, pages 144–163, 2011.

[5] O. Goldreich. Towards a theory of software protection and simulation by oblivious rams. In *STOC*, 1987.

[6] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious rams. *J. ACM*, 1996.

[7] M. T. Goodrich and M. Mitzenmacher. Mapreduce parallel cuckoo hashing and oblivious ram simulations. *CoRR*, abs/1007.1259, 2010.

[8] M. T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia. Oblivious ram simulation with efficient worst-case access overhead. In *CCSW*, 2011.

[9] M. T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia. Privacy-preserving group data access via stateless oblivious ram simulation. In *SODA*, 2012.

[10] J. Hsu and P. Burke. Behavior of tandem buffers with geometric input and markovian output. In *IEEE Transactions on Communications. v24*, pages 358–361, 1976.

[11] E. Kushilevitz, S. Lu, and R. Ostrovsky. On the (in)security of hash-based oblivious ram and a new balancing scheme. In *SODA*, 2012.

[12] R. Ostrovsky. Efficient computation on oblivious rams. In *STOC*, 1990.

[13] R. Ostrovsky and V. Shoup. Private information storage (extended abstract). In *STOC*, pages 294–303, 1997.

[14] B. Pinkas and T. Reinman. Oblivious ram revisited. In *CRYPTO*, 2010.

[15] M. Raab and A. Steger. "balls into bins" - a simple and tight analysis. In *Proceedings of the Second International Workshop on Randomization and Approximation Techniques in Computer Science*, RANDOM '98, pages 159–170, London, UK, 1998. Springer-Verlag.

[16] E. Shi, H. Chan, E. Stefanov, and M. Li. Oblivious ram with $o((\log n)^3)$ worst-case cost. Online TR, `eprint.iacr.org/2011/407.pdf`, 2011.

[17] E. Stefanov, E. Shi, and D. Song. Towards practical oblivious ram. Manuscript, 2011.

[18] P. Williams and R. Sion. Usable PIR. In *NDSS*, 2008.

[19] P. Williams, R. Sion, and B. Carbunar. Building castles out of mud: practical access pattern privacy and correctness on untrusted storage. In *CCS*, 2008.

Appendix

A Bounding the Load of Each Bucket

In this section, we prove the following high probability statement for bounding the load in each bucket.

Theorem 3 (Each Bucket Has Small Load)**.** *Let* $0 < \delta < \frac{1}{2^{2e}}$*. For a fixed time and a fixed bucket, the probability that the bucket has load more than* $\log_2 \frac{1}{\delta}$ *is at most* δ*.*

Recall that the number of levels is $L := \lceil \log_2 N \rceil$. We analyze the load according to the depth i of the bucket.

A.1 Bounding the Load for Levels 0 to $L-1$ with Markov Process

Observe that in our scheme, when a block inside some bucket is accessed, the block is removed from the bucket. However, for the purpose of analysis, we assume that a block stays inside its bucket when it is accessed, i.e., a block can leave a bucket only when the bucket is chosen for eviction; moreover, since we are only concerned about the load of a bucket, for simplicity we also assume that the blocks arriving at a bucket are all distinct. The load of a bucket in our scheme is always bounded above by the corresponding load in the modified process, which we analyze using a Markov process. If we assume that a bucket is initially empty, then its load will be stochastically dominated by the load under the stationary distribution.

Defining Markov Process $\mathcal{Q}(\alpha, \beta)$. Given $0 < \alpha \leq \beta \leq 1$, we describe a Markov process $\mathcal{Q}(\alpha, \beta)$ with non-negative integral states as follows. In order to illustrate the relationship between the Markov process and the load of a bucket, we define $\mathcal{Q}(\alpha, \beta)$ using the terminology related to the bucket. The state of the Markov process corresponds to the current load of a bucket. At any time step, the following happens independently of any past events in the specified order:

(a) With probability α, a block arrives at the bucket.

(b) If the load of the bucket is non-zero (maybe because a block has just arrived), then with probability β a block departs from the bucket.

Recall that when a block departs from a depth-i bucket, it arrives at one of the two depth-$(i+1)$ child buckets uniformly at random.

Example. We immediately see that the root bucket is modeled by $\mathcal{Q}(1, 1)$ and a depth-1 bucket is modeled by $\mathcal{Q}(\frac{1}{2}, 1)$. Both cases are trivial because the load at the end of every time step is zero. One can see that at every time step a block arrives at one of the four depth-2 buckets uniformly at random and two out of the four buckets are chosen for eviction every step. Hence, each of the depth-2 buckets can be modeled by $\mathcal{Q}(\frac{1}{4}, \frac{1}{2})$. Using a classic queuing theory result by Hsu and Burke [10] we can show that at further depths, a block leaves a bucket with some fixed probability at every time step, so that independent arrivals are satisfied at the child buckets.

Corollary 2 (Load of an Internal Bucket)**.** *For* $2 \leq i < L$*, under the stationary distribution, the probability that a depth-*i *bucket has load at least* s *is at most* $\rho_i^s \leq \frac{1}{2^s}$*; in particular, for* $0 < \delta < 1$*, with probability at least* $1 - \delta$*, its load is at most* $\log_2 \frac{1}{\delta}$*.*

Proof. The proof builds on top of a classic queuing theory result by Hsu and Burke [10]. Full proof is provide in our online technical report [16]. □

A.2 Bounding the Load of Level L with "Balls into Bins"

Observe that a block residing at a depth-L bucket traversed a random path from the root bucket to a random leaf bucket. Hence, given that a block is at depth L, the block is in one of the leaf buckets uniformly at random. Hence, to give an upper bound on the load of a leaf bucket at any single time step, we can imagine that each of the N blocks is placed independently in one of the leaf buckets uniformly at random. This can be analyzed by the well-known "Balls into Bins" process.

Corollary 3 (Load of a Leaf Bucket)**.** *For each time step, for* $0 < \delta < \frac{1}{2^{2e}}$*, with probability at least* $1 - \delta$*, a leaf bucket has load at most* $\log_2 \frac{1}{\delta}$.

Proof. Using standard balls and bins analysis [15]. Full proof will be supplied in online technical report [16]. $\square$