An Oblivious Parallel RAM with $O(\log^2 N)$ Parallel Runtime

Kartik Nayak[*] Jonathan Katz[*]

Abstract

Oblivious RAM (ORAM) is a cryptographic primitive that allows a client to access memory locations from a server without revealing its access patterns. Oblivious Parallel RAM (OPRAM) is a PRAM counterpart of Oblivious RAM, i.e., it allows m clients that trust each other to simultaneously access data from a server without revealing their access patterns. The best known OPRAM scheme [1, 2] achieves amortized client-server bandwidth of $O(\log^2 N)$ per lookup, but they do not achieve perfectly linear access time speedup with clients. In fact, for each access, the blowup for the slowest client (also known as parallel runtime blowup) is $O(f(m) \log m \log^2 N), f(m) = \omega(1)$. This implies that, for most accesses, some clients remain idle while others are accessing data. In this work, we show an OPRAM scheme that has parallel runtime blowup of $O(\log^2 N)$ while maintaining $O(\log^2 N)$ client-server bandwidth blowup for each client.

1 Introduction

Oblivious RAM is a cryptographic primitive that allows a client to store and retrieve blocks of data on an untrusted server in such a way that no information about the access pattern of the client's accesses is revealed to the server beyond the total number of accesses. This primitive was introduced by Goldreich and Ostrovsky [8,9,15] in a model where a trusted processor is accessing data from main memory during execution of some protected software. Since this initial work, there has been a lot of work on designing ORAM schemes in the secure data-outsourcing scenario [4, 10–14, 16, 18–20, 23], secure processor scenario [5, 17, 20], and for applications to secure computation [21, 22].

ORAM schemes inherently make multiple physical accesses to portions of server memory for each logical memory access. ORAM schemes mainly try to optimize the bandwidth, i.e., the (amortized) client-server communication for each logical memory access, as a function of N, the total number of memory locations. Assuming no computation on the server, the best known ORAM scheme achieves a bandwidth of $O(\log^2 N/ \log\log N)$ when the block size of memory is $\Omega(\log N)$ [12] and $O(\log N)$ if the block size is $\Omega(\log^2 N)$ [20, 21].

A fundamental limitation of ORAM is that it allows only one client to accesses memory. Sometimes—for example, in a multicore processor—it may be necessary to allow multiple independent clients to access the same memory at the same time. Oblivious *Parallel* RAM (OPRAM) [1] addresses this, and allows m clients to access data obliviously. A key challenge in realizing OPRAM is to ensure that when multiple clients access the same data block, no additional information is revealed. Also, the communication between clients may also be visible to the server, and hence this communication pattern needs to be oblivious as well. Boyle et al. introduced an OPRAM scheme with client-server bandwidth blowup of $O(\log^3 N)$ [1], which was later improved by Chen, Lin, and Tessaro [2] to an amortized blowup of $O(\log^2 N)$ per client.

A key metric for any parallel system is the parallel runtime of the computation, i.e., the number of rounds of communication with the server until all clients have obtained the result. The OPRAM scheme by Chen *et al.* has parallel runtime blowup of $O(f(m) \log m \log^2 N), f(m) = \omega(1)$, compared to an insecure PRAM

[*]Department of Computer Science, University of Maryland. Email: `{kartik,jkatz}@cs.umd.edu`.

Table 1: **Complexity of OPRAM protocols.** Here, $f(m) = \omega(1)$ and κ is the computational security parameter for an additive homomorphic encryption scheme.

	Client-server bandwidth blowup (amortized)	Client-server bandwidth blowup (worst case)	Parallel runtime blowup
Trivial	$O(\log^2 N)$	$O(\log^2 N)$	$O(m \log^2 N)$
Boyle *et al.* [1]	$O(\log^3 N)$	$O(\log^3 N)$	$O(\log^3 N)$
Chen *et al.* [2]	$O(\log^2 N)$	$\omega(\log N \log \lambda) + O(\log^2 N)$	$O(f(m) \log m \log^2 N)$
Our scheme (with AHE)	$O(\log N(\log N + \kappa))$	$O(\log N(\log N + \kappa))$	$O(\log N(\log N + \kappa))$
Our scheme (w/o AHE)	$O(\log^2 N)$	$O(\log^2 N)$	$O(\log^2 N)$

that does not hide access patterns. In this work we improve upon this and show an OPRAM construction that achieves parallel runtime blowup of $O(\log^2 N)$, while maintaining bandwidth blowup of $O(\log^2 N)$. We first show this by allowing the server to perform additive homomorphic encryption (Section 3.2). We then improve upon this construction to avoid any computation on the server (Section 3.5).

1.1 Overview of our Construction

In Path ORAM [20], memory is stored as a tree and each block is mapped to a random leaf in this tree. Given a (block b, leaf l) mapping, the following invariant holds: the block b would be stored in some tree node from root to the leaf l. In OPRAM, there are m clients and for each access, each of the m clients request one logical block from memory. The OPRAM scheme by Chen *et al.* accesses data from memory using Path ORAM scheme. The physical memory is partitioned into m parts, forming m Path ORAM subtrees. Each client is allowed to read and write only to its assigned part. With this, they ensure that no two clients read or write to the same physical location at the same time. Given blocks in Path ORAM are mapped to a leaf uniformly at random, for each access, it can be shown by a simple balls and bins analysis that $O(f(m) \log m), f(m) = \omega(1)$ leaves may be mapped to one subtree. Thus, for each access, some client performs k Path ORAM accesses. Hence, the span of computation (or the parallel runtime) is a factor k larger than the ideal span in which each client performs one ORAM access.

Our protocol is geared towards making each client perform exactly one access, thereby obtaining ideal span. A key observation we make is that although *writing* to a particular physical location cannot be performed by multiple clients in parallel, the constraint does not hold for *read* operations. A Path ORAM access involves the following three steps: (1) Reading a path from the tree (2) Path write-back after deleting the block read (3) Path eviction to percolate blocks towards the leaves of the tree. In our protocol, each client can read a single path from the servers in parallel. We suggest two different ways for performing path write-back after deletion. The first method involves server computation by using a homomorphically encrypted integer valid, which maintains whether a block is *real.* Each of the clients perform write-back in parallel by updating only the value valid and the server performs additive homomorphic computation on this value. The second method involves no server computation. Each node is written back to the tree by a unique client, which is selected through some inter-client communication. Finally, the eviction procedure is performed using reverse lexicographic ordering of paths introduced by Gentry *et al.* This ordering ensures that clients evict paths that do not overlap with each other.

```
function Access(addr, op, $v'$)
    $l \leftarrow$PosMap [addr]
    stash$\leftarrow$ stash$\cup$ReadPath($l$)
    $l' \leftarrow$ UniformRandom($0, N-1$)
    PosMap [addr] $\leftarrow l'$
    $v \leftarrow$ ReadBlockFromStash(addr, stash)
    if op = Read then
        return $v$ to client
    else
        $v \leftarrow v'$
    evict($l$, stash)
```

Figure 1: **Path ORAM data access algorithm.** Here, PosMap is a map from an address addr to a leaf l of the tree. ReadPath(l, addr) retrieves a path from the root until leaf l. ReadBlockFromStash(addr, stash) retrieves a valid block with address addr from the stash.

2 Preliminaries

2.1 Oblivious RAM

The basic oblivious RAM setting consists of two parties, a client and a server. The client wants to outsource storage of N data blocks, each of size B bits, on the server, and can also store a small amount of data locally. The client outsources (encrypted) data blocks to the server and can arbitrarily access the blocks using $\mathsf{Access}(\mathsf{op}, \mathsf{addr}, v')$ commands. Here, $\mathsf{op} \in \{\mathsf{Read}, \mathsf{Write}\}$, addr is the logical address of the block, and v' is the data to be written to the block when $\mathsf{op} = \mathsf{Write}$. An ORAM scheme hides a sequence of *logical* accesses $\mathsf{Access}(\mathsf{op}, \mathsf{addr}, v')$ made by the client from the untrusted server who observes the corresponding *physical* accesses output by the ORAM. The input to the ORAM algorithm can be viewed as a sequence of logical accesses $\mathsf{Access}(\mathsf{op}_i, \mathsf{addr}_i, v'_i)$. To "hide" the logical accesses, they are converted into multiple physical accesses per request $(\mathsf{op}_{i,1}, \mathsf{addr}_{i,1}, v'_{i,1}), \ldots, (\mathsf{op}_{i,q}, \mathsf{addr}_{i,q}, v'_{i,q})$. For the ith logical access, at the end of the qth request, the client learns contents of block with address addr_i. Typical ORAM constructions have $q = \mathsf{polylog}(N)$ physical accesses for every logical access.

2.1.1 Path ORAM

We provide a high-level overview of Path ORAM [20].

Server Storage. In Path ORAM, the data blocks are stored in a binary tree. Each node of the binary tree is called a bucket and contains $Z = O(1)$ blocks. Each block may contain B bits of information. Some of these blocks may be dummy, i.e., contain no information. All the buckets are encrypted using a symmetric key randomized encryption scheme and hence the server does not learn whether a block is dummy.

Metadata. For each block on the tree, along with the blocks, some metadata is also stored. In particular, metadata consists of an identifier to denote the logical address addr and a validity bit valid. In addition, the client stores a map PosMap from each addr to a path to a leaf l of the tree.

Main invariant. A key invariant of Path ORAM is that, each block is mapped to a leaf $l \leftarrow [N]$ in the binary tree. This block is stored in some bucket from the root to the leaf l.

Stash. The client maintains a small amount of local memory, called the *stash.* This is used to hold overflowing blocks from the tree.

Accessing a block. A pseudocode for the algorithm to access a block is shown in Figure 1. Each access $\mathsf{Access}(\mathsf{op}, \mathsf{addr}, v')$ of a block consists of two steps:

1. **Retrieve data.** The client looks up the position map to determine the leaf l corresponding to addr. It then reads the path from root to leaf l and adds them to the stash. The block with address addr is then read from the stash. The position map for addr is updated to a randomly selected leaf l'. If $\mathsf{op} = \mathsf{Write}$, it updates the block content to v'.

2. **Path write-back and eviction.** The eviction operation is used to upload blocks from the stash to the tree. The client iterates over every block b in stash and inserts it along the path l on the lowest possible bucket that is also along the path assigned to b. If there are no positions available, b remains in the stash.

Obliviousness and size of stash. For every access, a single path is read from the tree. This location of this path is statistically independent of all the previous paths accessed. Hence, intuitively, the access algorithm is oblivious.

In [20], the authors analyze size of stash for polynomial number of accesses. They show that for a bucket size $Z = 5$, tree height of $\lceil \log N \rceil$ and stash size R, the probability of failure for s accesses is $\leq s.14.(0.6002)^R$. Thus, for $s = poly(N)$, for a stash size of $O(f(N) \log N)$, $f(N) = \omega(1)$, the probability of failure is negligible in N.

Recursion. The position map consists of $O(N \log N)$ bits. When $B > \log N$, the size of this position map is smaller than the original data. This data can be stored recursively as another ORAM tree with an even smaller position map. As was shown in [20], if this is done $O(\log N)$ times, the client storage reduces to $O(1)$ blocks. We assume such recursion is done in what follows.

Improved eviction procedure. Gentry et al. [7] introduced an alternative eviction procedure for Path ORAM. In this procedure, instead of evicting along the same path chosen for reading the block, after t-th access, a leaf $l = \mathsf{DigitReverse}(t)$ is chosen for eviction ($\mathsf{DigitReverse(t)}$ represents the bits of t in the reverse-bit order). This reduces the required bucket size of Path ORAM from $Z = 4$ to $Z = 2$.

Bandwidth blowup. Path ORAM achieves a bandwidth overhead of $O(\log^2 N)$ when $B = \Omega(\log N)$, and $O(\log N)$ when $B = \Omega(\log^2 N)$.

2.2 Oblivious Parallel RAM (OPRAM)

An oblivious parallel RAM (OPRAM) setting consists of (1) m stateful interactive clients and (2) a server S. The m clients together outsource N (encrypted) data blocks, each of size B, to the server S. The server stores $M(N)$ data blocks, where $M(\cdot)$ is the storage overhead of the OPRAM scheme. Each of the clients access logical data blocks in parallel. At the beginning of an access, each client i inputs $\mathsf{Access}(\mathsf{op}_i, \mathsf{addr}_i, v'_i)$ where, $\mathsf{op_i} \in \{\mathsf{Read}, \mathsf{Write}\}$, addr_i is the logical address of the block, and v'_i is the data to be written to the block when $\mathsf{op}_i = \mathsf{Write}$. Subsequently, the clients perform an interactive protocol where the clients either (1) read/write data from server S or (2) communicate with other clients. We assume that all the clients hold a shared symmetric secret key. Communication between clients as well as the data stored on the server are encrypted using this secret key.

Write-conflict resolution. We note that in a PRAM, multiple Read operations to the same address can be executed concurrently. However, when multiple clients perform a Write to the same address, only one of the operations can be applied. Without loss of generality, we assume that write request of the client with smallest identifier takes effect. Moreover, when multiple clients read and write to the same address during one access, all clients read the contents of the block before the write operation is performed.

Correctness and obliviousness. During each access, the server S observes the physical accesses by each client on S as well as the communication pattern between the clients. Let $OPRAM(\mathbf{y})$ denote this sequence of interactions between the clients and the server as well as communication between clients. Let $\mathbf{y} = (\mathbf{y}_1, \ldots, \mathbf{y}_t)$ denote t OPRAM accesses where each access $\mathbf{y}_j$ is given by $\mathbf{y}_j = (\mathsf{Access}^j(\mathsf{op}_1, \mathsf{addr}_1, v'_1), \ldots, \mathsf{Access}^j(\mathsf{op}_m, \mathsf{addr}_m, v'_m))$.

We say that an OPRAM algorithm is correct if for each access $j \in [t]$, $OPRAM(\mathbf{y}_j)$ returns data that is consistent with $\mathbf{y}_j$ except with negligible probability. In other words, for each client i, for each access t, the following holds: client i receives block v as output, where v is the most recently written value to address addr_i or $\perp$ if addr_i has not yet been written.

We say that an OPRAM algorithm is secure if for two access patterns $\mathbf{y}$ and $\mathbf{z}$ with $|\mathbf{y}| = |\mathbf{z}|$, their access patterns $OPRAM(\mathbf{y})$ and $OPRAM(\mathbf{y})$ are computationally or statistically indistinguishable. Respectively, the OPRAM algorithms are called computationally or statistically secure.

Efficiency metrics. The following metrics have been used to determine the efficiency of OPRAM schemes.

- **Client-server bandwidth blowup.** The client-server bandwidth blowup is the ratio of number of bits accessed by the slowest client in the oblivious PRAM for each logical access to the number of bits accessed by a non-oblivious PRAM for each logical access.
- **Inter-client communication.** Different clients may access the same memory block from the server during the same access and hence need to communicate to ensure obliviousness. Inter-client communication is the amount of data communicated by each client (in bits) with other clients for each logical access.

2.3 Oblivious Inter-client Communication

In the following subsections, we discuss oblivious inter-client communication protocols from previous work [1, 2] that we use in our construction.

2.3.1 Oblivious Aggregation Protocol (OblivAgg)

An oblivious aggregation protocol is used to aggregate data held by multiple clients holding the same key. Each client begins holding as input a key and some data. At the end of the protocol, the following holds: for each key, a unique client (say, the client with the smallest id holding that key initially) outputs a function Agg of all data associated with that key. We assume that the aggregation function Agg is associative, i.e., $\mathsf{Agg}(\mathsf{Agg}(\mathsf{data}_1, \mathsf{data}_2), \mathsf{data}_3) = \mathsf{Agg}(\mathsf{data}_1, \mathsf{Agg}(\mathsf{data}_2, \mathsf{data}_3))$.

OblivAgg

Input: Each client i has input $(\mathsf{key}_i, \mathsf{data}_i)$.
Output: Each client i receives either $(\mathsf{key}, \mathsf{data})$ or $(\perp, \perp)$. For each key, a unique client outputs $(\mathsf{key}, \mathsf{agg}_{\mathsf{key}})$ where $\mathsf{agg}_{\mathsf{key}} = \mathsf{Agg}(\{\mathsf{data}_j \mid j : \mathsf{key}_j = \mathsf{key}\})$.

There exists a protocol for this functionality that can be implemented in $O(\log m)$ rounds, where in each round, each client sends $O(1)$ messages of size $O(\log m + |\mathsf{data}|)$.

2.3.2 Oblivious Election Protocol (OblivElect)

An oblivious election protocol selects a unique representative for each address that appears in m client requests. If there is a client that writes to an address, then the client with the smallest id writing to that address is chosen as the representative. Otherwise, the party with the smallest id reading from that address is chosen as the representative.

OblivElect

Input: Each party i has input $(\mathsf{op}_i, \mathsf{key}_i)$ where $\mathsf{op}_i \in \{\mathsf{Read}, \mathsf{Write}\}$.
Output: Each party i receives either rep or $\perp$. For each unique $\mathsf{key} = \mathsf{key}_i$, the party with smallest i having input $(\mathsf{Write}, \mathsf{key})$ will output rep. If no party issues a Write request, then the party with lowest i with input $\mathsf{Read}, \mathsf{key}$ will output rep. All other parties output $\perp$.

$\mathsf{OblivElect}$ can be implemented using $\mathsf{OblivAgg}$ with $|\mathsf{data}| = \log N$. Thus, the functionality can be implemented in $O(\log m)$ rounds, where in each round, each client sends $O(1)$ messages of size $O(\log m + \log N)$.

2.3.3 Oblivious Routing Protocol (OblivRoute)

This protocol allows each party to send a message to another party in such a way that the recipient of the message is hidden.

> OblivRoute
>
> **Input:** Each client i has input (id_i, data_i), where $1 \le id_i \le m$.
> **Output**: Each client i outputs $\{(id_j, \mathsf{data}_j) \mid id_j = i\}$, i.e., the set of messages for which it is the receiver.

We use $\mathsf{OblivRoute}$ only in the scenario where the recipient (id_i) is chosen at random by each client. Boyle *et al.* [1] implement $\mathsf{OblivRoute}$ in $O(\log m)$ rounds, where in each round clients send messages of size $O(K(|\mathsf{data}_i| + \log m))$; the protocol aborts with probability $O(m \log m 2^{-K})$. This can be made negligible by setting $K = \omega(\log \lambda)$, where λ is a statistical security parameter.

2.3.4 Oblivious Multicast (OblivMCast)

The oblivious multicast protocol allows a subset of clients to multicast a message to other clients such that each client can either act as a sender or receiver. Each client can send and/or receive exactly one message.

> OblivMCast
>
> **Input:** Each sender i has input $(\mathsf{send}, id_i, v_i)$, where $1 \le id_i \le m$. Each receiver i has input $(\mathsf{recv}, id_i, \perp)$. For every possible id, there is at most one sender with $id_i = id$.
> **Output:** Each sender i receives (id_i, v_i) as its output; each receiver i receives output (id_j, v_j), where $id_j = i$.

The $\mathsf{OblivMCast}$ protocol is similar to the $\mathsf{OblivElect}$ protocol and can be implemented in $O(\log m)$ rounds, where in each round each client sends $O(1)$ messages of size $O(\log m + \log N + |v|)$ where v is the message sent/received by a client.

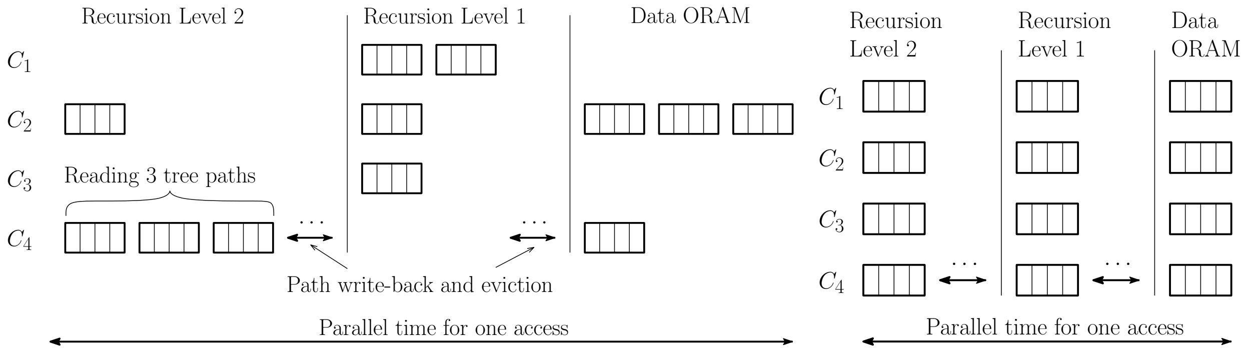
3 Our OPRAM Construction

We describe our construction in this section. We first define parallel runtime blowup for an OPRAM access in Section 3.1. Section 3.2 describes our construction for one recursion level assuming a shared local position map between clients and assuming the server performs computation. Section 3.3 explains how this position map can be offloaded to the server and accessed recursively. We discuss the security and asymptotics of our construction in Section 3.4. Finally, we discuss how the construction can be modified to achieve the same parallel runtime blowup without server computation in Section 3.5.

3.1 Parallel Runtime Blowup for an OPRAM Access

Definition. We assume a fixed bandwidth D between each client i and server S. In one round of communication, a subset of clients read/write D bits of information from/to the server. The parallel runtime is the total number of rounds of client-server communication for each OPRAM access. Thus, parallel runtime blowup is the ratio of rounds of client-server communication required for an OPRAM access to the rounds of client-server communication required for a PRAM access.

Parallel runtime blowup for Chen *et al.* Figure 2a shows an example for parallel time for one access



(a) Scheme by Chen *et al.* [2] (b) Our scheme

Figure 2: **Parallel runtime for a single OPRAM access.** The figure shows parallel runtime an OPRAM access performed for $m = 4$ clients and two Path ORAM recursion levels. For ease of exposition, only paths read by each client is shown.

in the scheme by Chen *et al.* for $m = 4$ clients and two Path ORAM recursion levels in addition to the data ORAM. For simplicity, only paths read by clients are shown in the figure. Communication for path write-back and eviction are not shown.

For each ORAM tree access, clients access one random path. The scheme by Chen *et al.* partitions memory into m parts and one client is responsible for each part. As there are m clients, one of the clients may access multiple paths. For example, in Figure 2a, when accessing the main ORAM tree, client C_2 accesses three paths whereas C_1 and C_2 do not access a path. Similarly, in the first recursion level, client C_1 accesses two paths while C_4 accesses none. Accessing a path at recursion level j depends on the position map value obtained through recursion level $j+1$. Accesses between two recursion levels are thus dependent. Hence, although client C_4 does not access a block at the first recursion level, it cannot proceed to accessing data ORAM.

For a fixed bandwidth of D bits, accessing one block of B bits requires $\lceil B/D \rceil$ rounds of communication. The slowest client accesses up to $O(\lambda)$ paths with all but negligible probability. For Path ORAM, each path consists of $O(\log N)$ blocks and there are $O(\log N)$ recursion levels. The parallel runtime is thus $O(\lceil (Bf(m) \log m \log N)/D \rceil \log N)$ rounds of communication. The parallel runtime blowup is $\frac{O(\lceil (Bf(m) \log m \log N)/D \rceil \log N)}{\lceil B/D \rceil} = O(f(m) \log m \log^2 N)$.

Parallel runtime blowup for our construction. The parallel runtime blowup for our construction is shown in Figure 2b. In contrast to Chen *et al.*, in our construction, each client accesses exactly one path for each ORAM tree access. By a similar analysis as above, it can be shown that our construction obtains parallel runtime blowup of $O(\log^2 N)$.

3.2 Non-recursive Construction

At a very high level, our construction follows a structure similar to the OPRAM protocol of Chen *et al.* However, compared to their work, we ensure that each client performs an equal amount of work in parallel for every access.

Server storage. For simplicity, we assume that both the number of data items N and the number of clients m are powers of 2. The server storage is organized as a complete tree with N leaves numbered 0 to $N-1$, but with the top $\log m$ levels of the tree removed. In other words, the data is organized as a forest containing m trees $\mathcal{T}_1, \ldots, \mathcal{T}_m$, each of height $\log N - \log m$. Each node in the tree is a bucket of size $Z = O(1)$ blocks, each of size B bits. As in Path ORAM, blocks are encrypted (see more below) and some of them may be dummy blocks.

Each block logically stores some data, the logical address `addr` to which that data corresponds, and an integer valid with $\mathsf{valid} = 1$ indicating that the block is valid. The data and address are encrypted using a symmetric-key encryption scheme, whereas valid is encrypted using an additively homomorphic (public-key) encryption scheme.

Managing a tree. Client i is responsible for managing the i-th tree $\mathcal{T}_i$ and all leaves therein. That is, client 1 is responsible for leaves $0, \dots, N/m - 1$; client 2 is responsible for leaves $N/m, \dots, 2N/m - 1$, and so on. The client responsible for a leaf l, denoted $\mathsf{Resp}(l)$, is in charge of:

1. Maintaining a stash for all blocks assigned to leaf l. As in [2], the required stash size is $\omega(\log \lambda) + O(\log m)$.
2. Performing an eviction operation for the path to leaf l.

Access. Consider a logical access where each client begins holding input $(\mathsf{op}_i,\ \mathsf{addr}_i,\ v'_i)$ with $\mathsf{op}_i \in \{\mathsf{Read}, \mathsf{Write}\}, 1 \le \mathsf{addr}_i \le N$, and $v'_i = \perp$ in case of a read operation. We assume the clients have access to a shared position map.

1. **Select block representatives.** The m clients run sub-protocol $\mathsf{OblivElect}$ (Section 2.3.2) where each client i has input $(\mathsf{op}_i,\ \mathsf{addr}_i)$. Each client outputs rep or $\perp$. We call any client who does not output $\perp$ a *representative*.
2. **Lookup position map.** Each client i does:
    - **Representative clients.** If client i is a representative for some address addr_i, then it uses the shared position map PosMap to obtain the leaf l_i corresponding to addr_i. It then generates a fresh leaf $l'_i \leftarrow [N]$ and sets PosMap $(\mathsf{addr}_i) = l'_i$.
    - **Other clients.** If client i is not a representative, then it performs a random access to PosMap. It generates uniform leaves $l_i, l'_i \leftarrow [N]$.
3. **Search for block in stash.** Suppose client $j = \mathsf{Resp}(l_i)$. Each client i does:
    - **Representative clients.** Client i sends addr_i to client j.
    - **Other clients.** Client i sends $\perp$ to client j.

    Client j looks up addr_i in its stash. If the block is found in stash, it is deleted and returned to client i. Otherwise, a dummy block is returned to client i.
4. **Read paths.** Suppose $j = \mathsf{Resp}(l_i)$. Each client i reads a path $\mathcal{P}(\mathcal{T}_j, l_i)$ from root of $\mathcal{T}_j$ until leaf l_i. Note that overlapping paths can be read by two different clients.
5. **Delete block with address addr_i.** In this step, the block that was read by each client is deleted on the server.
    - **Representative clients.** Suppose k-th block on $\mathcal{P}(\mathcal{T}_j, l_i)$ has address addr_i and $\mathsf{valid} = 1$. The client generates a vector Y of length $|\mathcal{P}(\mathcal{T}_j, l_i)| = Z \log N$. The k-th element of this vector is $Enc(1)$ and all other elements are $Enc(0)$, where Enc is an additively homomorphic encryption scheme. The client sends Y to the server. For the k-th block in the path, server computes $Enc(\mathsf{valid}_k) \times Y_k$.
        - For the block that is to be deleted, $Y_k = Enc(1)$. Hence, $Enc(\mathsf{valid}_k) \times Y_k = Enc(1) \times Enc(1) = Enc(2)$. $\mathsf{valid}_k = 2$ implies an invalid block.
        - For all other blocks, $Y_k = Enc(0)$. Hence, $Enc(\mathsf{valid}_k) \times Y_k = Enc(\mathsf{valid}_k) \times Enc(0) = Enc(\mathsf{valid}_k)$. Thus, these blocks are not invalidated.
    - **Other clients.** The client generates a vector Y of length $|\mathcal{P}(\mathcal{T}_j, l_i)|$ where each element is $Enc(0)$. The client sends Y to the server. The server performs $Enc(\mathsf{valid}_k) \times Enc(0) = Enc(\mathsf{valid}_k)$. Thus, none of these blocks are invalidated.

6. **Answer client requests and update.** At the end of the previous step, every representative client holds v_i. The clients now run a protocol $\mathsf{OblivMCast}$ where:

    - **Representative clients.** Representative clients act as senders and multicast the block to all clients that requested it. Representative clients use input $(\mathsf{send}, \mathsf{addr}_i, v_i)$. If the representative client requests a write operation, it updates value of the block to v_i' locally.
    - **Other clients.** Non-representative clients do not hold valid blocks and act as receivers in the protocol. Hence, client i uses input $(\mathsf{recv}, \mathsf{addr}_i, \perp)$. Client i receives output (addr_i, v_i'') where addr_i is the block that client i requested and v_i'' is the value of the block.

7. **Route blocks to clients responsible for the block.** Recall that l_i' is the new leaf generated uniformly at random by client i in step 2. Suppose $j = \mathsf{Resp}(l_i')$. The clients run an $\mathsf{OblivRoute}$ protocol where each representative i uses input $(j, (l_i', \mathsf{addr}_i, b_i))$ and other clients use $(j, (l_i', \perp, \perp))$. Each client k receives a set of blocks $\{(l_p', \mathsf{a}_p, b_p')\}$ where $k = \mathsf{Resp}(l_p')$. The client adds valid (non-dummy) blocks to local stash.

8. **Evict using reverse lexicographical paths.** Each client uses a reverse lexicographical path (as suggested by Gentry *et al.* [6]) to evict as in Path ORAM. The reverse lexicographical ordering ensures that a client i responsible for managing a tree $\mathcal{T}_i$ evicts exactly one path from the tree.

3.3 Recursive Construction

Our solution for the recursive construction uses the same idea as Chen *et al.* [2]. Our non-recursive construction stores N blocks of size B bits and requires the existence of a shared position map PosMap of size $N \log N$ bits. Assuming $B \geq c \log N, c > 1$, one can recursively store the position map in $O(\log N)$ trees to achieve a position map of size $O(1)$ blocks. This can be stored by one of the clients and other clients can access this information using $\mathsf{OblivAgg}$ and $\mathsf{OblivMCast}$ protocols.

In our OPRAM construction, as m clients access m blocks, if they request to write to the same block one of the clients is chosen as a representative and this client performs the write. However, when position map is accessed, position values for different data blocks may be stored in the same position map block. Thus, for correctness, updates to all these distinct position map values in the same position map block should be incorporated. This can be achieved by replacing the $\mathsf{OblivElect}$ protocol in step 1 by an $\mathsf{OblivAgg}$ protocol during recursion. Here, each client inputs the new leaf values and the representative client aggregates these values by performing a union of the values. More precisely, each client i uses input $(\mathsf{key}_i = \mathsf{addr}_i, \mathsf{data}_i = l_i)$, and the aggregate function is $\mathsf{Agg}(\mathsf{data}_1, \mathsf{data}_2, \ldots) = \mathsf{data}_1 || \mathsf{data}_2 || \ldots$. The representative client outputs $(\mathsf{rep}, \mathsf{Agg}(\mathsf{data}_1, \mathsf{data}_2, \ldots))$ while other clients output $(\perp, \perp)$.

Given that a position map block stores a maximum of a constant c position map values, the size of the aggregated data is bounded by $c \log N$. Thus, this step of the protocol proceeds in $O(\log m)$ rounds, where in each round, every client sends $O(1)$ messages of size $O(\log N + \log m)$ bits.

3.4 Obliviousness and Asymptotics of our Construction

Obliviousness. We first note that all the inter-client protocols discussed in Section 2.3 are oblivious. Thus, steps 1, 6 and 7 are oblivious. Step 2 is a recursive access to position map. For each access, steps 3, 4 and 5 reveal a new leaf and hence a new random path that is statistically indpendent of all the paths accessed before. In step 8, each client a path is chosen deterministically. As each of these steps are oblivious, the access algorithm is oblivious.

Asymptotics. Let us first find the client-server bandwidth required. Let $|\mathsf{ct}|$ denote size of the AHE ciphertext for valid. Due to reverse lexicographical ordering of paths for eviction, each path is evicted every N/m accesses. Thus, the maximum value that valid can assume is N/m. Assuming Paillier cryptosystem, $|\mathsf{ct}| = O(\kappa + \log(N/m))$ bits, where κ is the number of bits used in the modulus of Paillier cryptosystem.

Steps 1, 3, 6 and 7 do not involve client-server communication. Step 2 involves accessing $O(\log N)$ position map blocks (one from each recursion tree) of size $O(\log N)$ bits. For each position map block, a path of $Z \log N = O(\log N)$ blocks is read. The write-back operation involves writing back $|\mathsf{ct}| \log N$ bits, where This results in $O(\log^2 N(\log N + |\mathsf{ct}|))$ bits for position map data.

For data ORAM access, in steps 4 and 8, each client accesses exactly one path which is $O(\log N)$ blocks and $O(\log N)$ ciphertexts. Step 5 requires uploading $O(\log N)$ ciphertexts of an AHE scheme. This results in accessing $O((B + |\mathsf{ct}|) \log N)$ bits by each client. Thus, the per client client-server communication is $O((B + \kappa \log N + \log^2 N) \log N)$ bits. For a sufficiently large block with $B = \Omega(\kappa \log N + \log^2 N)$, the blowup is $O(\log N)$. For $B = \Omega(\kappa + \log N)$, the blowup is $O(\log^2 N)$.

For each of the steps, when clients communicate with the server, each client accesses a single path on the server independently of each other. Thus, the parallel client-server runtime blowup is $O((\log N + \kappa) \log N)$ blowup.

3.5 Avoiding Additive Homomorphic Encryption

Step 5 of our construction in Section 3.2 requires the server to perform additive homomorphic encryption (AHE). In this section, we describe the protocol without using AHE. In this step, all clients write back blocks in the read path except the block b of interest. b is replaced by a dummy block. This was done by setting the value of $\mathsf{valid} \neq 1$ using AHE.

In this modified construction, we use valid', which is a bit to indicate validity of the block. This bit is stored encrypted using symmetric key encryption on the server. As two intersecting paths can be read by two clients and the required blocks can reside in the intersecting portions (buckets) of the paths, it is necessary that only one client deletes and writes back this intersecting portion.

Let $\mathsf{path}_{i,k}$ denote the length k bit prefix of path l_i. Each client performs the following - for each level $\mathsf{lev} = \log m + 1, \ldots, \log N$ of the tree -

- The clients run $\mathsf{OblivAgg}$ procedure with $\mathsf{key}_i = \mathsf{path}_{i,\mathsf{lev}}$, $\mathsf{data}_i = b_i$ if b_i is in the current level of the path l_i, otherwise $\mathsf{data}_i = \perp$. The aggregation function is the union function $\mathsf{Agg}(\mathsf{data}_1, \mathsf{data}_2, \ldots) = \mathsf{data}_1 || \mathsf{data}_2 || \ldots$. As the output of the protocol, for each block with bucket prefix $\mathsf{path}_{i,\mathsf{lev}}$ held by clients, a representative client j is chosen for deletion of bucket $\mathsf{path}_{i,\mathsf{lev}}$.
- The representative client j performs the deletion of bucket $\mathsf{path}_{i,\mathsf{lev}}$ in the ORAM tree $\mathcal{T}_{\mathsf{path}_{i,\mathsf{lev}}}$.

The above step ensures that a unique client is selected for deleting (possibly zero) blocks from every bucket that was read. The other steps from Section 3.2 remain unchanged.

Client-server bandwidth blowup and parallel runtime blowup. The only modification from analysis in the previous section is that, AHE ciphertexts are not read/written by clients. Hence, the per client client-server communication is $O((B + \log^2 N) \log N)$ bits. For $B = \Omega(\log^2 N)$, the per client bandwidth blowup is $O(\log N)$ while for $B = \Omega(\log N)$, the blowup is $O(\log^2 N)$. Similarly, the client-server parallel runtime blowup is $O(\log^2 N)$ for all block sizes.

4 Conclusion

In this work, we show a scheme that obtains $O(\log^2 N)$ client-server bandwidth blowup with $O(\log^2 N)$ parallel runtime.

5 Acknowledgements

This work is funded in part by NSF award #1563722 and a Google Ph.D. Fellowship award.

References

[1] E. Boyle, K.-M. Chung, and R. Pass. Oblivious parallel ram and applications. In *Theory of Cryptography*, pages 175–204. Springer, 2016.

[2] B. Chen, H. Lin, and S. Tessaro. Oblivious parallel ram: improved efficiency and generic constructions. In *Theory of Cryptography*, pages 205–234. Springer, 2016.

[3] K.-M. Chung and R. Pass. A simple oram. Technical report, DTIC Document, 2013.

[4] S. Devadas, M. van Dijk, C. W. Fletcher, L. Ren, E. Shi, and D. Wichs. Onion oram: A constant bandwidth blowup oblivious ram. In *Theory of Cryptography Conference*, pages 145–174. Springer, 2016.

[5] C. W. Fletcher, M. v. Dijk, and S. Devadas. A secure processor architecture for encrypted computation on untrusted programs. In *Proceedings of the seventh ACM workshop on Scalable trusted computing*, pages 3–8. ACM, 2012.

[6] C. Gentry, K. A. Goldman, S. Halevi, C. S. Jutla, M. Raykova, and D. Wichs. Optimizing ORAM and using it efficiently for secure computation. In *Privacy Enhancing Technologies Symposium (PETS)*, 2013.

[7] C. Gentry, S. Halevi, C. Jutla, and M. Raykova. Private database access with he-over-oram architecture. In *Applied Cryptography and Network Security*, pages 172–191. Springer, 2015.

[8] O. Goldreich. Towards a theory of software protection and simulation by oblivious rams. In *ACM symposium on Theory of computing (STOC)*, 1987.

[9] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious rams. *J. ACM*, 43(3), May 1996.

[10] M. T. Goodrich and M. Mitzenmacher. Privacy-preserving access of outsourced data via oblivious ram simulation. In *Proceedings of the 38th international conference on Automata, languages and programming - Volume Part II*, ICALP'11, pages 576–587, Berlin, Heidelberg, 2011. Springer-Verlag.

[11] M. T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia. Privacy-preserving group data access via stateless oblivious ram simulation. In *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '12, pages 157–167. SIAM, 2012.

[12] E. Kushilevitz, S. Lu, and R. Ostrovsky. On the (in)security of hash-based oblivious ram and a new balancing scheme. In *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '12, pages 143–156. SIAM, 2012.

[13] S. Lu and R. Ostrovsky. Distributed oblivious ram for secure two-party computation. In *Theory of Cryptography*, pages 377–396. Springer, 2013.

[14] K. Nayak, L. Ren, C. W. Fletcher, I. Abraham, and B. Pinkas. Asymptotically tight bounds for composing oram with pir.

[15] R. Ostrovsky. Efficient computation on oblivious rams. In *Proceedings of the twenty-second annual ACM symposium on Theory of computing*, STOC '90, pages 514–523, New York, NY, USA, 1990. ACM.

[16] L. Ren, C. W. Fletcher, A. Kwon, E. Stefanov, E. Shi, M. van Dijk, and S. Devadas. Ring oram: Closing the gap between small and large client storage oblivious ram. *IACR Cryptology ePrint Archive*, 2014:997, 2014.

[17] L. Ren, X. Yu, C. W. Fletcher, M. Van Dijk, and S. Devadas. Design space exploration and optimization of path oblivious ram in secure processors. In *ACM SIGARCH Computer Architecture News*, volume 41, pages 571–582. ACM, 2013.

[18] E. Shi, T.-H. H. Chan, E. Stefanov, and M. Li. Oblivious ram with o((logn)3) worst-case cost. In *ASIACRYPT*, pages 197–214, 2011.

[19] E. Stefanov and E. Shi. Oblivistore: High performance oblivious cloud storage. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 253–267. IEEE, 2013.

[20] E. Stefanov, M. van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas. Path ORAM – an extremely simple oblivious ram protocol. In *CCS*, 2013.

[21] X. Wang, H. Chan, and E. Shi. Circuit oram: On tightness of the goldreich-ostrovsky lower bound. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 850–861. ACM, 2015.

[22] X. S. Wang, Y. Huang, T. H. Chan, A. Shelat, and E. Shi. Scoram: oblivious ram for secure computation. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 191–202. ACM, 2014.

[23] X. S. Wang, K. Nayak, C. Liu, T. Chan, E. Shi, E. Stefanov, and Y. Huang. Oblivious data structures. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 215–226. ACM, 2014.