Oblivious Parallel RAM: Improved Efficiency and Generic Constructions

Binyi Chen(✉), Huijia Lin, and Stefano Tessaro

Department of Computer Science, University of California, Santa Barbara, USA
{binyichen,rachel.lin,tessaro}@cs.ucsb.edu

Abstract. Oblivious RAM (ORAM) garbles read/write operations by a client (to access a remote storage server or a random-access memory) so that an adversary observing the garbled access sequence cannot infer any information about the original operations, other than their overall number. This paper considers the natural setting of Oblivious *Parallel* RAM (OPRAM) recently introduced by Boyle, Chung, and Pass (TCC 2016A), where m clients simultaneously access in *parallel* the storage server. The clients are additionally connected via point-to-point links to coordinate their accesses. However, this additional inter-client communication must also remain oblivious.

The main contribution of this paper is twofold: We construct the first OPRAM scheme that (nearly) matches the storage and server-client communication complexities of the most efficient single-client ORAM schemes. Our scheme is based on an extension of Path-ORAM by Stefanov et al. [18]. Moreover, we present a *generic* transformation turning any (single-client) ORAM scheme into an OPRAM scheme.

1 Introduction

This paper considers the problem of hiding *access patterns* when reading from and writing to an untrusted memory or storage server. This is a fundamental problem in both in the context of software protection, as well as for secure outsourcing to a third-party storage provider.

The basic cryptographic method to hide access patterns is *Oblivious RAM* (ORAM) [8,9]. It compiles logical access sequences (from a client) into garbled ones (to a storage space, or *server*) so that a curious observer seeing the latter only (as well as the server contents) cannot infer anything *other than the overall number of logical accesses*—we say that such garbled access sequences are *oblivious*. Since its proposal, ORAM and its applications have been extensively studied (cf e.g. [1,3–7,9,11–21,24–28,30]). The state-of-the-art constructions [16,26] have a $\tilde{O}(\log^2 N)$ computation (and communication) overhead (per logical access),[1] where N is the size of the storage, i.e., the number of *data* bocks (of a certain bit size) it can store.

[1] The ORAM scheme of [16] has only $O(\log^2 N/\log\log N)$ overhead, while that of [26] has $O(\log^2 N)$ overhead. However, the latter construction is simpler and achieves better practical efficiency [26].

E. Kushilevitz and T. Malkin (Eds.): TCC 2016-A, Part II, LNCS 9563, pp. 205–234, 2016.
DOI: 10.1007/978-3-662-49099-0_8

Parallel Oblivious Accesses. Existing ORAM schemes only support a *single* client, and in particular do not deal with *parallel accesses from multiple clients*. However, enabling such parallelism is important, e.g., to achieve scalable cloud storage services for multiple users, or to secure multi-processor architectures. To overcome this barrier, a few systems-oriented works [15,24,30] suggested to either use a trusted proxy shared by multiple clients to act as the "sole client" of ORAM, or to adapt known ORAM schemes (such as [7,9,29]) to support a limited, $O(\log N)$, number of parallel accesses.

Recently, Boyle, Chung, and Pass (BCP) [2] proposed the notion of *Oblivious Parallel RAM (OPRAM)*, which compiles *synchronous parallel* logical access sequences by m clients into, *parallel*, garbled sequences and inter-client messages, which together still reveal no information other than the total number of logical accesses. They also provided the first – and so far, the only – OPRAM scheme. Their construction is simple and elegant, but, has a server-client communication overhead of $\omega(\log^3 N)$—a factor of $\tilde{\Omega}(\log N)$ higher than state-of-the-art ORAM schemes [16,26]. Their approach seems not to extend directly to use the techniques behind existing communication-efficient ORAM schemes.

Hence, the natural question that arises is: *"Can we design an OPRAM scheme with the same per-client efficiency as the state-of-the-art ORAM schemes?"*

Our Contributions, in a Nutshell. Our first contribution answers this question affirmatively. In particular, we prove:

> **Theorem 1 (Informal):** *There is an OPRAM scheme with $O(\log^2 N)$ (amortized) server-client communication overhead, and constant storage overhead.*

Going beyond, an even more fundamental question concerns the basic relation between ORAM and OPRAM. We show that the two problems are related at a far more generic level:

> **Theorem 2 (Informal):** *There is a generic transformation that turns any ORAM scheme into an OPRAM scheme, with additional $O(\log N)$ (amortized) server-client communication overhead with respect to the original ORAM scheme.*

While the above results are in the amortized case, we note that in the worst case, the above complexity statements are true with O replaced by ω. Moreover, our OPRAM schemes all require client-to-client communication. Their inter-client communication is $\omega(\log N) \log m(\log m + \log N)B$ bits. We note that this also is an improvement by a factor $O(\log N)$ over BCP.

We stress that our approach is substantially different from that of BCP: One key idea is the use of partitioning, i.e., the fact that each client is responsible for a designated portion of the server storage. This eliminates much of the coordination necessary in BCP. Next, we move to explaining the high-level ideas behind our constructions in greater detail.

1.1 Subtree-OPRAM

We provide an overview of our scheme Subtree-OPRAM. Our construction of an m-client OPRAM scheme can be seen as consisting of two steps.

(1) First, we construct an ORAM scheme, called *Subtree-ORAM*, that enables a *single* client to batch-process m logical accesses at a time *in parallel*. Our Subtree-ORAM scheme is a generalization of Path-ORAM [26] to the setting with large client memory and parallel processing. We believe that this generalization is of independent interest.

(2) In a second step, we exploit the batch-processing structure of Subtree-ORAM to adapt it to the multiple-client setting, and derive our Subtree-OPRAM scheme by distributing its computation across multiple clients.

In the following, we explain all of this in more detail.

Review of Path-ORAM. Let us first give an overview of the tree-based ORAM approach by Shi et al. [23]. In particular, we review Path-ORAM [26], as it will serve as our starting point. (A more detailed review is given in Appendix B.)

To implement a storage space for N *data blocks*, basic (i.e., non-recursive) Path-ORAM organizes the storage space (virtually) as a complete binary tree with depth $O(\log N)$, where each node is a "bucket" that contains a fixed number $Z = O(1)$ of encrypted blocks (some of which may be dummies). To hide access patterns, each data block is assigned to a random path ℓ (from a leaf ℓ to the root, and we use ℓ to identify both the leaf and the associated path interchangeably) and stored in *some* bucket on path ℓ; after each access, the assignment is "refreshed" to a new random path ℓ'. The client keeps track of the current path assigned to each block using a *position map*. The client also keeps an additional (small) memory for overflowing blocks, called the *stash*. For each logical access to a certain block with address $\mathsf{a} \in [N]$, Path-ORAM takes the two following steps:

(1) Fetching a path. Retrieve the path ℓ currently associated with block a in the position map, and find block a on the path or in the local *stash*. Then, assign the block a to a new random path ℓ' and update the position map accordingly.

(2) Flushing along a path. Iterate over every block a' in the fetched path ℓ and in the stash (this includes the block a we just retrieved and possibly updated, and which was assigned to the new path ℓ'), and re-insert each block a' into the lowest possible bucket on ℓ that is also on the path assigned to a' according to the position map. If no suitable place is found (as each bucket can only contain at most Z blocks), the block is placed into the stash. The contents of the path are re-encrypted when being written back to the server (including dummy blocks).

The analysis of Path-ORAM [26] shows that the stash size is bounded by $\omega(\log N)$ with probability roughly $poly(\lambda)2^{-\omega(\log N)}$. To avoid keeping a large

position map, Path-ORAM *recursively* stores the position map at the server. The final scheme has a recursion depth of $O(\log N)$—each logical access is translated to $O(\log N)$ actual accesses, each consisting of retrieving a path. Overall, the communication overhead is $O(\log^2 N)$. Also, the overall storage complexity at the server can be kept to $O(N)$ despite the recursion.

Subtree-ORAM. As our first contribution, we generalize Path-ORAM to process $m \geq 1$ logical accesses at a time. As the recursion step in Path-ORAM is rather generic, we focus on the non-recursive scheme, ignoring the costs of storing the position map.

The natural approach to achieve this is to retrieve a *subtree* of m paths, i.e., for every m logical accesses to blocks $\mathsf{a}_1, \ldots, \mathsf{a}_m$, we can do the following:

(1) Fetching subtree. Retrieve the subtree ST composed of the paths $\ell_1, \ldots, \ell_m$ assigned to the m blocks and find the blocks of interest in the subtree or in the stash, and possibly update their values.

(2) Path-by-path flushing. Execute the flushing procedure from Path-ORAM on the m paths in ST sequentially as in Path-ORAM, with each a_i assigned to a new random path ℓ'_i.

Unfortunately, there are *two* problems with this approach. First, if $\mathsf{a}_1, \ldots, \mathsf{a}_m$ are not all distinct, the accesses are not oblivious, as the same path would be retrieved multiple times. To avoid this, the final Subtree-ORAM scheme perform some pre-processing: For accesses to the same block, replace all but the first one with $\perp$ in the logical sequence to obtain $\mathsf{a}'_1, \ldots, \mathsf{a}'_m$, and for each repetition $\mathsf{a}'_i = \perp$, assign random path to be retrieved from the server — this is called a *fake read*.[2]

The second drawback is that repeating the flushing procedure of Path-ORAM m times in Step 2 is inherently sequential. To use Subtree-ORAM within Subtree-OPRAM below, we instead target a parallelizable flushing procedure. To this end, we introduce the following new flushing procedure, which we refer to as *subtree flushing*:

(2) Subtree flushing: Iterate over every block in ST and in the stash and place each block into the lowest node in the *entire subtree* ST that is still on its assigned path, and not yet full. The order in which blocks are processed can be arbitrary, and the process can be parallelized (subject to maintaining the size constraint of each node).

Security and correctness of Subtree-ORAM follow similar arguments as Path-ORAM. Furthermore, we bound the stash size of Subtree-ORAM by generalizing aspects of the analysis of Path-ORAM – we believe this to be of independent interest.

Subtree-OPRAM. Our end goal is to design an interactive protocol that enables m clients to access (and possibly alter) blocks $\mathsf{a}_1, \ldots, \mathsf{a}_m$ in parallel,

[2] Note that this random path may well collide with one of the other paths. Still, the key point is that it is chosen *independently* of the actual blocks. The use of such fake read has appeared in many previous works, such as, [2,24].

where client C_i is requesting in particular block a_i; both the access patterns to the server, as well as inter-client communication, must be oblivious.

We can think of our Subtree-OPRAM protocol as having the m clients collectively emulate the single Subtree-ORAM client. To this end, we use inter-client oblivious communication protocols based on tools developed in [2] to let clients interact with each other. Here, we focus our description on how to "distribute" Step 1 and Step 2 of Subtree-ORAM for the special cases that the requested blocks $\mathsf{a}_1, \ldots, \mathsf{a}_m$ are distinct. (Handling colliding requests in an oblivious way will require extra work.) For simplicity, we assume that all clients have access to the position map and all messages are implicitly encrypted (with a key shared by all clients). In particular, everything is re-encrypted before being written to the server.

Assume for simplicity that $m = 2^l$. We can think of the server storage in Subtree-OPRAM in terms of a tree of buckets, as in Path-ORAM and Subtree-ORAM. However, we remove the top l levels, effectively turning the tree into a forest of m trees $T_1, \ldots, T_m$; client C_i manages all read/write from/to T_i, and all blocks assigned to (a path in) T_i that do not fit in one of the buckets on the server remain in a local stash managed locally by C_i. More precisely:

(1) In parallel, each C_i finds the path ℓ_i assigned to a_i (using the position map) and delegates the job of reading path ℓ_i to the client C_j responsible for the tree T_j containing ℓ_i, to which it sends a request. Each C_j retrieves all paths for which it has received a request (again in parallel), which form a subtree ST_j of T_j; it then finds the blocks of interest in ST_j and its local stash, and sends them back to the respective clients who requested them.

(2) Each C_i assigns a_i a new path ℓ'_i, and delegates the job of writing back (B_i, ℓ'_i) to the client C_j responsible for the tree T_j containing ℓ'_i. To ensure obliviousness, the clients achieve this by running collectively the oblivious routing protocol of [2], which hides the destination of messages. Next, each C_i runs the subtree-flushing procedure locally on the retrieved subtree ST_i and its own stash, and finally writes the entire subtree ST_i back.

We will show that the m clients indeed collectively emulate the execution of the single client of Subtree-ORAM. In particular, parallel flushing on the individual subtrees emulates the effect of a global flushing over the union of these subtrees, but keeping the top l levels of the tree locally at the clients; also, the *union* of the stashes of all clients contains exactly the contents of the stash of the Subtree-ORAM client, as well as the contents of the top of the tree. This gives a bound on the overall sizes of the stashes.

In expectation, each client reads and writes one path per round, and thus the amortized client-server communication overhead is $O(\log N)$, and the final recursive Subtree-OPRAM has amortized overhead of $O(\log^2 N)$, with overwhelming probability. In fact, we prove that the worst-case overhead is not much higher, and is of the order of $\omega(\log^2 N)$, e.g., $O((\log^2 N) \cdot \log\log N)$, much smaller than BCP's $\omega(\log^3 N)$. We improve over BCP also in terms of inter-client communication complexity by a factor of $\log N$.

1.2 The Generic Transformation

Subtree-OPRAM is tailored at achieving the same overhead as Path-ORAM, and not surprisingly, the former heavily relies on the latter. Our second contribution is a generic transformation that converts *any* ORAM scheme into an OPRAM protocol. When applied to Path-ORAM, the resulting scheme is less efficient than Subtree-OPRAM – still, the main benefit here is *generality*.

Our approach generalizes ideas from partition-based ORAM [25]. Specifically, we split the server storage into m partitions each storing (roughly) N/m blocks, and let the m clients run each a copy of the basic ORAM algorithm (call them $\mathcal{O}_1, \ldots, \mathcal{O}_m$). Each client C_i thus manages the i-th partition independently using $\mathcal{O}_i$. Every block a is randomly assigned to one of the m partitions $P \in [m]$, and it is re-assigned to a new random partition after each access. The current assignment of blocks to the m partitions is recorded in a *partition map*, which we assume (for now) to be accessible by all clients. (In the end, it will be shared using recursion techniques.) Then, when m clients request the m blocks $\mathsf{a}_1, \ldots, \mathsf{a}_m$ in parallel, the clients simply find the respective partitions $P_1, \ldots, P_m$ containing these blocks, and let the corresponding clients retrieve the desired blocks and delete them from their partitions (if a block is accessed for multiple times, then "fake reads" are performed to a random partition). The actual access pattern *so far* is oblivious since all P_i's are random, and the basic ORAM scheme ensures that retrieving blocks from each partition is done obliviously.

However, writing these blocks back to new random partitions without revealing their destinations turns out to be non-trivial, *even if we can deliver the blocks obliviously to the clients responsible for the new partitions.* Indeed, naively invoking the corresponding ORAM copies to insert would reveal how many blocks are assigned to each partition. To hide this information, in our protocol each client inserts the *same* number κ of blocks to its partition, and keeps a queue of blocks to be inserted. We use a stochastic analysis to show that for any $R = \omega(\log \lambda)$, it is sufficient to insert *only* $\kappa = 2$ *blocks to each partition each time (and in particular, perform fake "insertions" if less than 2 blocks need to be inserted)*, and at most R "overflowing" blocks ever remain in the queue (except with negligible probability).

A challenge we have not addressed is how to use an ORAM for a partition of size $O(N/m)$ to store the blocks associated with it in an efficient way, i.e., without using the whole space of $[N]$ addresses. We will solve this by using an appropriate ORAM-based oblivious dictionary data structure.

As the expected number of read and write operations each client performs is 3 (one read and two writes), the non-recursive version has the same (amortized) computation and communication overhead as the underlying ORAM scheme. To obtain the final OPRAM scheme, we apply recursive techniques to outsource the partition map to the server.

Notation. Throughout this paper, we let $[n]$ denote the set $\{1, 2, ..., n\}$. We denote by $\Delta(X, Y)$ the statistical distance between distributions (or random variables) X and Y, i.e. $\Delta(X, Y) = \sum_x |\Pr[X = x] - \Pr[Y = x]|$. Also, we say

that a function μ is *negligible* if for every polynomial p there exists a sufficiently large integer n_0, such that $\mu(n) \leq 1/p(n)$ for all $n > n_0$.

2 Oblivious (Parallel) RAM

We start by reviewing the notion of Oblivious RAM and its parallel extensions. We present definitions different from (yet essentially equivalent to) the ones by Goldreich and Ostrovsky [8,9] and BCP [2], considering clients and servers, instead of RAM compilers, which we consider to lead to more compact and natural descriptions, and are more in line with the applied ORAM literature.

Basic ORAM Setting. The basic ORAM setting considers two parties, a *client* and a *server*: The server $\mathcal{S}(M, B)$ has a large storage space consisting of M cells, each of size B bits, whereas the client has a much smaller memory. The client can access the storage space at the server using read and write commands, denoted as $\mathsf{Acc}(\mathsf{read}, \mathsf{a}, \bot)$ and $\mathsf{Acc}(\mathsf{write}, \mathsf{a}, v)$, where $\mathsf{a} \in [M]$ and $v \in \{0,1\}^B$. (We assume that all cells on the server are initialized to some fixed string, i.e., 0^B.) Both operations return the current value stored in cell a, in particular for the latter operation this is the value before the cell is overwritten with v.

An oblivious RAM (ORAM) scheme consists of an *ORAM client* $\mathcal{O}$ (or simply, an ORAM $\mathcal{O}$), which is a stateful interactive PPT machine which on initial input the security parameter λ, block size B, and storage size N, processes *logical* commands $\mathsf{Acc}(\mathrm{op}_i, \mathsf{a}_i, v_i)$, $\mathrm{op}_i \in \{\mathsf{read}, \mathsf{write}\}$, $\mathsf{a}_i \in [N]$, $v_i \in \{0,1\}^B \cup \{\bot\}$, by interacting with a server $\mathcal{S}(M, B)$ (for values $M = M(N)$ and $B = B(B, \lambda)$ explicitly defined by the scheme), via sequence of *actual* (read/write) accesses $\mathsf{Acc}(\overline{\mathrm{op}}_{i,1}, \overline{a}_{i,1}, \overline{v}_{i,1}), \ldots, \mathsf{Acc}(\overline{\mathrm{op}}_{i,q_i}, \overline{a}_{i,q_i}, \overline{v}_{i,q_i})$, and finally outputs a value val_i and updates its local state depending on the answers of these accesses.

An ORAM scheme hides the sequence of logical commands from an untrusted (honest-but-curious) server, who observes the actual sequence of accesses. The actual values written to the server can be hidden using semantically-secure encryption. Indeed, all known ORAM solutions have server cells hold each the encryption of a block, i.e., in general one has $B = B + O(\lambda)$. For this reason, we abstract away from the usage of encryption by dealing only with access-pattern security and tacitly assuming that all cells are going to be stored encrypted in the final scheme with a semantically secure encryption scheme, and that every write access to the server will be in form of a fresh re-encryption of the value. In this case, it makes sense to think of $B = B$, and an adversary who cannot see the value written to/read from the server.

We defer a definition of security and correctness for single-client ORAM in Appendix A, and here rather focus on generalizing above to the multi-client setting.

Multi-Client Setting. We now consider the setting of *oblivious parallel ORAM* (or OPRAM for short) with m clients. An m-client OPRAM is a set[3] of stateful

[3] For notational simplicity, we give definitions for the case where the number of clients m is fixed and independent of the security parameter. However, one can easily extend these definitions to the case where $m = m(\lambda)$ with some (straightforward) notational effort.

interactive PPT machines $\mathcal{PO} = \{\mathcal{O}_i\}_{i\in[m]}$ which all on initial input the security parameter λ, the storage size parameter N, and the block size B, proceed in rounds, interacting with the server $\mathcal{S}(M(N), B)$ (where M is a parameter of the scheme[4]) and with each other through point-to-point connections. At each round r the following steps happen: First, every client $\mathcal{O}_i$ receives as input a logical operation $\mathsf{Acc}(\mathrm{op}_{i,r}, \mathsf{a}_{i,r}, v_{i,r})$ where $\mathrm{op}_{i,r} \in \{\mathsf{read}, \mathsf{write}\}$, $\mathsf{a}_{i,r} \in [N]$ and $v_{i,r} \in \{0,1\}^B \cup \{\perp\}$. Then, the clients engage in an interactive protocol where at any time each client $\mathcal{O}_i$ can (1) Send messages to other clients, and (2) Perform one or more accesses to the server $\mathcal{S}(M, B)$. Finally, every $\mathcal{O}_i$ outputs some value $\mathrm{val}_{i,r}$.

Correctness and Obliviousness. We assume without loss of generality than the honest-but-curious adversary learns only the *access and communication patterns*. To this end, let us fix a sequence of logical access operations that are issued to the m clients in T successive rounds. First off, for all $i \in [m]$, we denote by $\boldsymbol{y}_i = \left(\mathsf{Acc}(\mathrm{op}_{i,r}, \mathsf{a}_{i,r}, v_{i,r})\right)_{r\in[T]}$ the sequence of logical operations issued to $\mathcal{O}_i$ in the T rounds, and let $\boldsymbol{y} = (\boldsymbol{y}_1, \ldots, \boldsymbol{y}_m)$.

Now, for an execution of an OPRAM scheme $\mathcal{PO}$ for logical sequence of accesses $\boldsymbol{y}$ as above, we let ACP_i be the round-i communication pattern, i.e., the transcript of the communication among clients and between each client and the server in round $i \in [T]$, except that actual contents of the messages sent among clients, as well as the values v_i in server accesses by the clients, are removed. We define

$$\mathsf{ACP}_{\mathcal{PO}}(\lambda, N, B, \boldsymbol{y}) = (\mathsf{ACP}_1, \ldots, \mathsf{ACP}_T).$$

Finally, we also denote the outputs client i as $\boldsymbol{val}_i = (val_{i,1}, \ldots, val_{i,T})$ and

$$\mathsf{Out}_{\mathcal{PO}}(\lambda, N, B, \boldsymbol{y}) = (\boldsymbol{val}_1, \ldots, \boldsymbol{val}_m).$$

The outputs $\boldsymbol{z} = \mathsf{Out}_{\mathcal{PO}}(\lambda, N, B, \boldsymbol{y})$ of $\mathcal{PO}$ are correct w.r.t. the parallel accesses sequence $\boldsymbol{y}$, if it satisfies that for each command $\mathsf{Acc}(\mathrm{op}_{i,t}, \mathsf{a}_{i,t}, v_{i,t})$ in $\boldsymbol{y}$, the corresponding output $val_{i,t}$ in $\boldsymbol{z}$ is either the most recently written value on address a_i, or $\perp$ if a_i has not yet been written. Moreover, we assume that if two write operations occur in the same round for the same address, issued by clients $\mathcal{O}_i$ and $\mathcal{O}_j$, for $i < j$, then the value written by $\mathcal{O}_i$ is the one that takes effect. Let $\mathsf{Correct}$ be the predicate that on input $(\boldsymbol{y}, \boldsymbol{z})$ returns whether $\boldsymbol{z}$ is correct w.r.t. $\boldsymbol{y}$.

Definition 1 (Correctness and Security). *An OPRAM scheme $\mathcal{PO}$ achieves correctness and obliviousness if or all $N, B, T = poly(\lambda)$, there exists a negligible function μ such that, for every λ, every two parallel sequences $\boldsymbol{y}$ and $\boldsymbol{y'}$ of the same length $T(\lambda)$, the following are satisfied:*

(i) ***Correctness.*** $\Pr\left[\mathsf{Correct}\left(\boldsymbol{y}, \mathsf{Out}_{\mathcal{PO}}(\lambda, N, B, \boldsymbol{y})\right) = 1\right] \geq 1 - \mu(\lambda)$.
(ii) ***Obliviousness.*** $\Delta\left(\mathsf{ACP}_{\mathcal{PO}}(\lambda, N, B, \boldsymbol{y}), \mathsf{ACP}_{\mathcal{PO}}(\lambda, N, B, \boldsymbol{y'})\right) \leq \mu(\lambda)$.

[4] As in the single-client case above, we simply assume that server blocks and logical blocks have the same size for simplicity, as we only consider the unencrypted case.

Usually, the values λ, N, B are understood from the context, and we thus often use $\mathsf{ACP}(\boldsymbol{y}) = \mathsf{ACP}_{\mathcal{PO}}(\lambda, N, B, \boldsymbol{y})$ for notational simplicity.

OPRAM Complexity. The *server-communication* overhead and *inter-client communication* overhead of an OPRAM scheme $\mathcal{PO}$ are respectively the number of bits sent/received per client to/from the server, and to/from other clients, per logical access command, divided by the block size B. Finally, the server storage overhead of $\mathcal{PO}$ is the number of blocks stored at the server divided by N, and client storage overhead is the number of blocks stored at each client after each parallel access.

3 OPRAM with $O(\log^2 N)$ Server Communication Overhead

In this section, we present our first OPRAM scheme, called Subtree-OPRAM.

Theorem 1 (Subtree-OPRAM). *For every m, there is a m-client OPRAM scheme with the following properties: Let λ, N, and B denote the security parameter, the size of the logical space, and block size satisfying $B \geq 2 \log N$.*

- ***Client Storage Overhead.*** *Every client keeps a local stash consisting of $R = (\omega(\log \lambda) + O(\log m)) \log N$ blocks.*
- ***Server Storage Overhead.*** *$O(1)$.*
- ***Server Communication Overhead.*** *The amortized overhead is $O(\log^2 N)$ and the worst case overhead is $\omega(\log \lambda \log N) + O(\log^2 N)$ with overwhelming probability.*
- ***Inter-Client Communication Overhead.*** *The amortized and worst-case overheads are both $\omega(\log \lambda) \log m(\log m + \log N)$ with overwhelming probability.*

In particular, when the security parameter λ is set to N, the server communication complexity is $\omega(\log^2 N)$ in the worst case, and $O(\log^2 N)$ amortized.

To prove the theorem, as discussed in the introduction, we first present a single-client ORAM scheme, Subtree-ORAM, that supports parallel accesses in Sect. 3.1, and then adapt it to the multiple-client setting to obtain Subtree-OPRAM in Sect. 3.3. We analyze these two schemes in Appendixes C and D. Additional helper protocols needed by Subtree-OPRAM are given in Sect. 3.2.

3.1 Subtree-ORAM

In this section, we describe the non-recursive version of Subtree-ORAM, where the client keeps a large position map of size $O(N \log N)$; the same recursive technique as in Path-ORAM can be applied to reduce the client memory size.

The Subtree-ORAM client, $\mathsf{ST}\text{-}\mathcal{O}$, keeps a logical space of N blocks of size B using $M(N) = O(N)$ blocks on the server. The server storage space is organized (virtually) as a complete binary tree $\mathcal{T}$ of depth $D = \log N$ (we assume for

simplicity that N is a power of two), where each node is a *bucket* capable of storing Z blocks. In particular, we associate leaves (and paths leading to them from the root) with elements of $[2^D] = [N]$. Additionally, $\mathsf{ST}\text{-}\mathcal{O}$ locally maintains a position map $\mathsf{pos.map}$ and a stash stash of size respectively $O(N \log N)$ bits and $R(\lambda) \in \omega(\log \lambda)$ blocks.

In each iteration r, the Subtree-ORAM client $\mathsf{ST}\text{-}\mathcal{O}$ processes a batch of m logical access operations $\{\mathsf{Acc}(\mathrm{op}_i, \mathsf{a}_i, v_i)\}_{i\in[m]}$ as follows:

1. **Pre-process.** Remove repetitive block accesses by producing a new m-component vector Q as follows: The i-th entry is set to $Q_i = (\mathrm{op}_i, \mathsf{a}_i)$ if the following condition holds, otherwise $Q_i = \perp$.
   - *Either*, there are (one or many) write requests to block a_i, and the i-th operation $\mathsf{Acc}(\mathrm{op}_i, \mathsf{a}_i, v_i)$ is the one with the minimal index among them.
   - *Or*, there are only read requests to block a_i, and the i-th operation $\mathsf{Acc}(\mathrm{op}_i, \mathsf{a}_i, v_i)$ is the one with the minimal index among them.
2. **Read paths in parallel.** Determine a set $S = \{\ell_1, \ldots, \ell_m\}$ of m paths to read, where each path is of one of the following two types:
   - **Real-read.** For each $Q_i = (\mathrm{op}_i, \mathsf{a}_i) \neq \perp$, set $\ell_i = \mathsf{pos.map}(\mathsf{a}_i)$ and immediately refresh $\mathsf{pos.map}(\mathsf{a}_i)$ to $\ell_i' \xleftarrow{\$} [N]$.
   - **Fake-read.** For each entry $Q_i = \perp$, sample a random path $\ell_i \xleftarrow{\$} [N]$.

   Then, retrieve all paths in S from the server, forming a subtree $\mathcal{T}_S$ of buckets with (at most) Z decrypted blocks in them.
3. **Post-process.** Answer each logical access $\mathsf{Acc}(\mathrm{op}_i, \mathsf{a}_i, v_i)$ as follows: Find block a_i in subtree $\mathcal{T}_S$ or stash, and returns the value of the block. Next, for each $Q_i \neq \perp$ if the corresponding logical access is a write operation $\mathsf{Acc}(\mathsf{write}, \mathsf{a}_i, v_i \neq \perp)$, update block a_i to value v_i.
4. **Flush subtree and write-back.** Let $\mathcal{T}_{\mathrm{real}}$ be the subtree consisting of only real-read paths in $\mathcal{T}_S$. Before (re-encrypting and) writing $\mathcal{T}_S$ back to the server, re-arrange the contents of $\mathcal{T}_{\mathrm{real}}$ and stash to fit as many blocks from stash into the subtree as follows:

   > **Subtree-flushing.** Move all blocks in $\mathcal{T}_{\mathrm{real}}$ and stash to a temporary set Λ. Traverse through all blocks in Λ in an *arbitrary* order: Insert each block with address a from Λ, either into the lowest non-full bucket in $\mathcal{T}_{\mathrm{real}}$ that lies on the path $\mathsf{pos.map}(\mathsf{a})$ (if such bucket exists), or into stash. If at any point, the stash contains more than R blocks, output $\mathsf{overflow}$ and abort.

In Appendix C, we briefly discuss the analysis of Subtree-ORAM, noting the bulk of it (proving that the overflow probability is small) is deferred to the full version for lack of space.

3.2 Oblivious Inter-client Communication Protocols

Subtree-OPRAM, which we introduce in the next section, will use as components a few oblivious inter-client communication sub-protocols which will allow

to emulate Subtree-ORAM in a distributed fashion. These are variants of similar protocols proposed in [2]. Their communication patterns are *statically fixed*, independent of inputs (and thus are oblivious in a very strong sense), and the communication and computation complexities of each protocol participant is small, i.e., roughly polylog(m) where m is the number of participants. We only describe the interfaces of these protocols; their implementations are based on $\log(m)$-depth sorting networks, and we refer the reader to [2] for further low-level details.

Oblivious Aggregation. Our first component protocol is used to aggregate data held by multiple users, and is parameterized by an *aggregation function* agg which can combine an arbitrary number of data items $\mathsf{d}_1, \mathsf{d}_2, \ldots$ (from a given data set) into an element $\mathsf{agg}(\mathsf{d}_1, \mathsf{d}_2, \ldots)$. The function agg is associative, i.e., $\mathsf{agg}(\mathsf{agg}(\mathsf{d}_1, \mathsf{d}_2, \ldots, \mathsf{d}_k), \mathsf{d}_{k+1}, \ldots \mathsf{d}_{k+r})$ and $\mathsf{agg}(\mathsf{d}_1, \mathsf{d}_2, \ldots, \mathsf{d}_k, \mathsf{agg}(\mathsf{d}_{k+1}, \ldots \mathsf{d}_{k+r}))$ both give us the same value as $\mathsf{agg}(\mathsf{d}_1, \ldots, \mathsf{d}_{k+r})$. Each party $i \in [m]$ starts the protocol with an input pair consisting of a pair $(\mathsf{key}_i, \mathsf{d}_i)$. At the end of the execution, each party i obtains an output with one of two forms: (1) $(\mathsf{rep}, \mathsf{d}^*)$, where d^* is the output of the aggregation function applied to $\{\mathsf{d}_j : \mathsf{key}_j = \mathsf{key}_i\}$, or (2) $(\perp, \perp)$. Moreover, for every key which appears among the $\{\mathsf{key}_i\}_{i \in [m]}$, there exists exactly one party i with $\mathsf{key}_i = \mathsf{key}$ receiving an output of type (1). We refer to each such party as the *representative* for key_i.

An aggregation protocol with fixed communication patterns, called $\mathsf{OblivAgg}$, is given in [2]. When the bit length of the data items and of the key values is at most ℓ bits, the protocol from [2] proceeds in $O(\log m)$ rounds, and in each round, every client sends $O(1)$ messages of size $O(\log m + \ell)$ bits.

Oblivious Routing. Another protocol we will use is the Oblivious Routing protocol $\mathsf{OblivRoute}$ from [2]. This m-party sub-protocol allows each party to send a message to another party; since the communication patterns are fixed, the recipients of the messages are hidden from an observer.

Protocol $\mathsf{OblivRoute}$:

- Input of party i: (id_i, m_i) where m_i is the message of client i and id_i is the index of the recipient of the messages.
- Output of party i: $\{(\mathrm{id}_j, m_j) \mid \mathrm{id}_j = i\}$ the set of messages sent to party i.

We note that the implementation of $\mathsf{OblivRoute}$ is tailored at the case where each id_i is drawn independently and uniformly at random from $[m]$. (And this will be the case of our application below.) For a parameter $K \geq 0$, their protocol proceeds in $O(\log m)$ rounds, and in every round, a client sends a message of size $O(K \cdot (\ell + \log m))$ bits to another client, where ℓ is the size of the inputs. Then, the probability that the protocol aborts is roughly $O(m \log m 2^{-K})$, and thus one can set $K = \omega(\log \lambda)$ for this probability to be negligible in λ, or $K = \omega(\log N)$ in our ORAM applications where N becomes the security parameter.

Oblivious Election. We will need a variant of the above $\mathsf{OblivAgg}$ protocol with stronger guarantees. In particular, we need a protocol $\mathsf{OblivElect}$ that allows

m parties with requests $\{(\mathrm{op}_i, \mathsf{a}_i)\}_{i\in[m]}$ to elect a unique representative party for each unique address that appears among the m requests. This representative will be the party with the smallest identity $i \in [m]$ wanting to write to that address (if it exists), or otherwise the one with the smallest identity wanting to read from it. Formally, the protocol provides the following interface.

Protocol OblivElect:

- Input of party i: $(\mathrm{op}_i, \mathsf{a}_i)$, where $\mathrm{op}_i \in \{\mathsf{read}, \mathsf{write}\}$ and $\mathsf{a}_i \in [N]$.
- Output of party i: a value $o_i = \{\mathsf{rep}, \perp\}$, which is defined as follows. For each address a, define $S_a = \{i \mid \mathsf{a}_i = a\}$ and $W_a = \{i \mid \mathsf{a}_i = a \ \wedge \ \mathrm{op}_i = \mathsf{write}\}$, and let $i^*(a) = \min(W_a)$ if W_a is non-empty, or $i^*(a) = \min(S_a)$ otherwise. Then, we let $o_i = \mathsf{rep}$ if and only if $i = i^*(\mathsf{a}_i)$, and $o_i = \perp$ otherwise.

OblivElect can be implemented by modifying OblivAgg. At the high level, OblivAgg proceeds as follows (we refer to [2] for further details):

- Initially, every client i inputs a pair $(\mathsf{key}_i, \mathsf{d}_i)$, and these inputs are reshuffled across clients and sorted according to the first component. That is, at the end of the first phase, any two clients $j < j'$ are going to hold a triple $(i(j), \mathsf{key}_{i(j)}, \mathsf{d}_{i(j)})$ and $(i(j'), \mathsf{key}_{i(j')}, \mathsf{d}_{i(j')})$, respectively, such that $\mathsf{key}_{i(j)} \leq \mathsf{key}_{i(j')}$ and $i(j) \neq i(j')$. This is achieved via a sorting network, where each client i initially holds $(i, \mathsf{key}_i, \mathsf{d}_i)$, and then such triples are swapped between pairs of clients (defined by the sorting network), according the key values.
- This guarantees that for every key which was initially input by $m' \geq 1$ clients, at the end of the first phase there exist m' consecutive clients $j, j+1, \ldots, j+m'-1$ (for some j) holding triples with $\mathsf{key}_{i(j)} = \cdots = \mathsf{key}_{i(j+m'-1)} = \mathsf{key}$. Then, client j is going to aggregate $\mathsf{d}_{i(j)}, \ldots, \mathsf{d}_{i(j+m'-1)}$, and the final representative for key is client $i(j)$. The aggregate information is sent back to the representatives by using once again a sorting network, sorting with respect to the $i(j)$'s.

We can easily modify OblivAgg to achieve OblivElect as follows. We run OblivAgg with client i inputting $\mathsf{key}_i = \mathsf{a}_i$ and $\mathsf{d}_i = (\mathrm{op}_i, i)$. However, the sorting network is not going to sort *solely* according to the key value, but *also* according to the associated d entry. In particular, we say that $(\mathsf{a}, \mathrm{op}, i) < (\mathsf{a}', \mathrm{op}', i')$ iff (1) $\mathsf{a} < \mathsf{a}'$, or (2) $\mathsf{a} = \mathsf{a}'$, $\mathrm{op} = \mathsf{write}$ and $\mathrm{op}' = \mathsf{read}$, or (3) $\mathsf{a} = \mathsf{a}'$, $\mathrm{op} = \mathrm{op}'$, and $i < i'$. The sorting now will ensure that the left-most client j holding a value for some $\mathsf{key} = \mathsf{a}$ will be such that $i(j)$ is our intended representative.

The complexity of OblivElect is the same as that of OblivAgg, setting $\ell = O(\log m + \log N)$. Thus we have $O(\log m)$ rounds, where each client sends $O(1)$ messages of size $O(\log m + \log N)$ bits.

Oblivious Multicasting. The oblivious multicast protocol OblivMCast is a m-party subprotocol that allows a subset of the parties, called the senders, to multicast values to others, called the receivers. More precisely:

Protocol OblivMCast:

- Input of party i: Input is either $(\mathsf{a}_i, v_i \neq \perp)$ (where $\mathsf{a}_i \in [N]$) indicating that party i is a sender with value v_i indexed by address a_i, or $(\mathsf{a}_i, \perp)$ indicating that it is a receiver fetching the value indexed by a_i. For every possible a, there is at most one party with $\mathsf{a}_i = \mathsf{a}$ and $v_i \neq \perp$.
- Output of party i: If party i is a sender, its output is v_i. If party i is a receiver, its output is v_j, the value sent by party j with index $\mathsf{a}_j = \mathsf{a}_i$.

The protocol is in essence the reversal of our OblivElect protocol above. It can be built using similar techniques, achieving round complexity $O(\log m)$, and every client sends in each round $O(1)$ messages of size $O(B + \log N + \log m)$ bits, where B is the bit size of the values v_i.

3.3 Subtree-OPRAM

Non-Recursive Subtree-OPRAM. We first describe the non-recursive version of Subtree-OPRAM, where multiple clients share access to a global position map, which can be eliminated using recursive techniques as we explain further below. (Due to the constraints of coordinating access to the same items in OPRAM, our recursive techniques are somewhat more involved than in the basic ORAM case.)

Let m be the number of clients; assume for simplicity that it is a power of 2, i.e., $\log(m)$ is an integer. The Subtree-OPRAM protocol $\mathsf{ST}\text{-}\mathcal{PO} = \{\mathcal{O}_i\}_{i\in[m]}$, on common input (λ, N, B, m), organizes the server storage as a forest of m complete binary trees $\mathcal{T}_1, \ldots, \mathcal{T}_m$, each of depth $\log N - \log(m)$, where every node in each tree is a bucket of $Z = O(1)$ blocks of B bits. In other words, the union of $\mathcal{T}_i$ is the complete tree $\mathcal{T}$ in Subtree-ORAM, but with the top $\log(m)$ levels removed. Again, we identify paths with leaves in the tree, and we say that a path ℓ "belongs to" $\mathcal{T}_i$, if the leaf ℓ is in $\mathcal{T}_i$. Each client $\mathcal{O}_i$ is responsible for managing the portion of the storage space corresponding to $\mathcal{T}_i$, meaning that it reads/writes all paths belonging to $\mathcal{T}_i$, and maintains a local stash stash_i for storing all "overflowing" blocks whose assigned path belongs to $\mathcal{T}_i$. The Subtree-ORAM analysis will carry over, and imply that the size of each local stash is bounded by any function $R(\lambda, m) \in \omega(\log \lambda) + O(\log m)$, where the extra $O(\log m)$ is to store blocks that in the original Subtree-ORAM scheme would have belonged to the upper $\log(m)$ levels. The clients also share a global size-N position map $\mathsf{pos.map}$. (Recall that we are looking at the non-recursive version here.)

Recall that the m clients share a secret key for a semantically secure encryption scheme. In each iteration, each client i processes a logical access request $\mathsf{Acc}(op_i, addr_i, v_i)$. The m clients then proceed in parallel to process jointly the m logical requests from this iteration:

1. **Select block representatives.** The m clients run sub-protocol $\mathsf{OblivElect}$, where client i uses input $(\mathrm{op}_i, \mathsf{a}_i)$ and receives either output rep or $\perp$; in the former case client i knows it is the *representative* for accessing block a_i.[5]
2. **Forward read-path requests.** Each client i determines the path ℓ_i it wants to fetch, and there are two possibilities:
   - **Real read.** If it is a representative, set path $\ell_i = \mathsf{pos.map}(\mathsf{a}_i)$ and $\mathsf{a}'_i = \mathsf{a}_i$, and immediately refresh $\mathsf{pos.map}(\mathsf{a}_i)$ to $\ell'_i \xleftarrow{\$} [N]$;
   - **Fake read.** If it is not a representative for a_i choose a random path $\ell_i \xleftarrow{\$} [N]$ and set $\mathsf{a}'_i = \perp$.

   If path ℓ_i belongs to tree $\mathcal{T}_j$, client i sends an encrypted message $(i, \mathsf{a}'_i, \ell_i)$ to client j.
3. **Read paths.** Each client $j \in [m]$ retrieves collects a set S_j of all paths contained in the messages $\{(i, \mathsf{a}'_i, \ell_i)\}$ received in the previous step, and then proceeds as follows:
   (1) Retrieve all paths in S_j, which form a subtree denoted $\mathcal{T}_{S_j}$.
   (2) For each $i \in [m]$ such that a request $(i, \perp, \ell_i)$ was received, send the encryption of a dummy block $\perp$ to client i
   (3) For each $i \in [m]$ such that a request $(i, \mathsf{a}'_i \neq \perp, \ell_i)$ was received, find block a'_i in $\mathcal{T}_{S_j}$ or in stash, *delete it*, and send the encryption of the value $\overline{v}_i$ of the block to client i.
4. **Answer client requests and update.** At the end of the previous step, each client holds a value $\overline{v}_i$ which is $\neq \perp$ if and only if i is the representative for a_i. Next, the m clients run sub-protocol $\mathsf{OblivMCast}$ to allow each representative to multicast the value it holds to other clients requesting the same block: Concretely, each client i uses input $(\mathsf{a}_i, \overline{v}_i)$ (recall a non-representative has $\overline{v}_i = \perp$) and receives output $\overline{v}'_i$, which is guaranteed to be the value of block a_i it requests. Each client i answers its logical request with $\overline{v}'_i$.
   Next, each representative i that has a write operation $\mathsf{Acc}(\mathsf{write}, \mathsf{a}_i, v_i)$ locally updates the value of block a_i to $\overline{v}_i = v_i$.
5. **Re-route blocks with newly assigned paths.** Each representative i send its block $(\mathsf{a}_i, \overline{v}_i)$ to the appropriate client for insertion according to the newly assigned path ℓ'_i (Step 1) as follows: Let j_i be the tree that path ℓ'_i belongs to; the m clients run sub-protocol $\mathsf{OblivRoute}$ where each representative i uses input $(j_i, (\ell'_i, \mathsf{a}_i, \overline{v}_i))$, and other clients use input $(j_i, \perp)$ for a randomly drawn $j_i \xleftarrow{\$} [m]$. [6]
   As the output of $\mathsf{OblivRoute}$, each client j receives a set of blocks $\{(\ell'_i, \mathsf{a}_i, \overline{v}_i)\}$ whose path ℓ'_i belong to $\mathcal{T}_j$; it stores each $(\mathsf{a}_i, \overline{v}_i)$ in its local stash stash_j.
6. **Flush subtree and write-back.** For each client j, let $\mathcal{T}_{\mathrm{real}_j}$ be the subtree consisting of only real-read paths in $\mathcal{T}_{S_j}$. Before writing subtree $\mathcal{T}_{S_j}$ back to the server (re-encrypting all of its contents), client j runs the Subtree

[5] Note that the representatives are chosen consistently with how repetition is removed in Subtree-ORAM.

[6] Note that the destination addresses of $\mathsf{OblivRoute}$ here are all uniformly chosen, and thus we can use the implementation from [2].

Flushing Procedure on $\mathcal{T}_{\text{real}_j}$ and stash_j (recall that if at any point, stash_j contains more than R blocks, the procedure output $\mathsf{overflow}$).

Recursive Version. We can apply recursion to eliminate the use of the shared global position map in the above scheme. Observe that in each iteration, each client read/write the position map at most once in Step 2. In other words, the m clients, in order to answer a batch of m accesses, one per client, to a logical space of size $N \times B$ bits, clients need to first make a batch of at most m accesses, one per client, to the position map of size $N \times \log N$ bits. Since $B \geq \alpha \log N$ for some constant $\alpha > 1$ (for simplicity, N is a power of two), by recursively storing the shared position map to the server in $O(\log N)$ trees, the clients no longer need to share any position map. At the end of recursion, the size of the position map decreases to $O(1)$ and can be stored in the local memory of say, the first client. Other clients can access and update this position map using oblivious sub-protocols $\mathsf{OblivAgg}$ and $\mathsf{OblivMCast}$.

This high-level strategy goes through almost identically as in Path-ORAM, except from the following caveat. Recall that in Step 2 of Subtree-OPRAM, if a client i is a representative, then it reads entry $\ell_i = \mathsf{pos.map}(\mathsf{a}_i)$ of the position map and updates it to a new random address ℓ'_i, and otherwise, it does not access the position map. Since $B \geq \alpha \log N$, the entire position map fits into a logical space of N/α blocks, where the block with address $\widetilde{\mathsf{a}}$ contains α position map entries, $\mathsf{pos.map}(\alpha\widetilde{\mathsf{a}}+1)||\cdots||\mathsf{pos.map}(\alpha(\widetilde{\mathsf{a}}+1))$. This means, when applying recursion and storing the position map at the server, client i needs to make the following logical access:

$$\mathsf{Acc}(\widetilde{\mathrm{op}}_i, \widetilde{\mathsf{a}}_i, \widetilde{v}_i) = \begin{cases} \mathsf{Acc}(\mathsf{write}, \lfloor \mathsf{a}_i/\alpha \rfloor, \ell'_i) & \text{if } i \text{ is a representative} \\ \mathsf{Acc}(\mathsf{read}, 0, \perp) & \text{otherwise} \end{cases}$$

We assume without loss of generality above that clients who are not representatives simply make a read access to the block with address 0. By construction, different representatives i and j access different entries in the position map $\mathsf{a}_i \neq \mathsf{a}_j$. However, it is possible that two representatives i and j need to access the same logical address $\widetilde{\mathsf{a}} = \widetilde{\mathsf{a}}_i = \widetilde{\mathsf{a}}_j$, in order to update different entries of position map located in the same block $\widetilde{\mathsf{a}}$—call this a *write-collision*; since each block contains at most α position map entries, there are at most α write collisions for the same logical address. Recall that in Subtree-OPRAM, when multiple clients write to the same logical address, only the write operation with the smallest index is executed. Hence, naively applying recursion on Subtree-OPRAM means when write-collision occurs, only one position map entry would be updated.

This problem can be addressed by slightly modifying the interface of Subtree-OPRAM, so that, under the constraint that there are at most α writes to different parts of the same block, all writes are executed. In recursion, the modified scheme is invoked, to ensure that position maps are updated correctly, whereas at the top level, the original Subtree-OPRAM is used. To accommodate α write collisions, the only change appears in Step 1: In Subtree-OPRAM, the sub-protocol $\mathsf{OblivElect}$ is used, which ensures that for each address a, only the

minimal indexed write is executed. We now modify this step to run the sub-protocol $\mathsf{OblivAgg}$ (with appropriate key, data and aggregate function specified shortly), so that, a unique representative is elected for each a, who receives all the write requests to that a, and executing all of them (note that while the write request are for the same block, they will concern different portions of the block corresponding to distinct position map entries, and thus "executing all of them" has a well-defined meaning):

1. **Select block representatives, modified.** The m clients run sub-protocol $\mathsf{OblivAgg}$, where client i uses input $(\mathsf{key}_i = \mathsf{a}_i, \mathsf{d}_i = v_i)$, and aggregate function $\mathsf{agg}(\mathsf{d}_1, \mathsf{d}_2, \cdots) = \mathsf{d}_1 || \mathsf{d}_2, \cdots = V$. $\mathsf{OblivAgg}$ ensures that for each address a_i, a unique client j accessing that address a_i receives output (rep, V_i), and all other clients receive output $(\perp, \perp)$. In the former case, client j knows it is the *representative* for accessing block a_i, and V_i determines the new value of the block v_i.

The rest of the protocol proceeds identically as before. Since there are at most α write collision for each address, the length of the output of agg is bounded by $\ell = \alpha B$. Thus the protocol proceeds in $O(\log m)$ rounds, where in each round every client sends $O(1)$ messages of size $O(\log N + \log m + B)$ bits.

4 Generic OPRAM Scheme

In this section, we generalize the ideas from Subtree-OPRAM to obtain a generic transformation transforming an arbitrary single-client ORAM to an OPRAM scheme, incurring only in a $O(\log N)$ factor of efficiency loss. Overall, we are going to prove the following general theorem.

Theorem 2 (Generic-OPRAM). *There exists a generic transformation that turns any ORAM scheme $\mathcal{O}$ into an m-client OPRAM scheme Generic-OPRAM such that, for any $R = \omega(\log \lambda)$, the following are satisfied, as long as the block length satisfied $B \geq 2 \log m$, and moreover $N/m \geq R$:*

- ***Server Communication Overhead.*** *The amortized communication overhead is $O(\log N \cdot \alpha(N/m))$ and the worst-case communication overhead is $O((\log N + \omega(\log \lambda)) \cdot \alpha(N/m))$, where $\alpha(N')$ is the communication overhead of ORAM scheme $\mathcal{O}$ with logical address space $[N']$.*
- ***Inter-Client Communication Overhead.*** *The amortized and worst-case overheads are both $\omega(\log \lambda) \log m (\log m + \log N)$ with overwhelming probability.*
- ***Server and Client Storage.*** *The sever stores $O(m \cdot M(N/m))$ blocks, where $M(N')$ is the number of blocks stored by $\mathcal{O}$ for logical address space N'. Moreover, the client's local storage overhead is $R + \mathrm{polylog}(N)$.*

Our presentation will avoid taking the detour of introducing a single-client ORAM scheme allowing for parallel processing of batches of m access operations, as we have done above with Subtree-OPRAM. A direct description of Generic-OPRAM is conceptually simpler. Before we turn to discussing Generic-OPRAM, however, we discuss a basic building block behind our protocol.

4.1 Oblivious Dictionaries

In our construction below, every client will be responsible for a partition holding roughly N/m blocks. One of the challenges is to store these blocks obliviously using space which is roughly equivalent to that of storing N/m blocks. Ideally, we want to implement this using an ORAM with logical address space for N/m blocks, as this would result in constant storage overhead when the ORAM has also constant overhead. In particular, the elements assigned to a certain partition have addresses spread around the whole of $[N]$, and we have to map them efficiently to be stored into some block in $[N/m]$ in a way which is (a) storage efficient for the client, and (b) only requires accessing a small (i.e., constant) number of blocks to fetch or insert a new block. We going to solve this via an oblivious data structure implementing a dictionary interface and able to store roughly N/m blocks into a not-much-larger amount of memory.

The Data Structure. We want an oblivious implementation $\mathcal{OD}$ of a dictionary data structure holding at most n pairs (a, v), where v corresponds to a data block in our ORAM scheme, and $\mathsf{a} \in [N]$. (For our purposes, think of $n \approx N/m$.) At any point in time, $\mathcal{OD}$ stores at most one pair (a, v) for every a. It allows us to perform two operations:

- $\mathcal{OD}(\mathcal{I}, \mathsf{a}, v)$ inserts an item (a, v), where $\mathsf{a} \in [N]$, if the data structure contains less than n elements. Otherwise, if n elements are stored, it does not add an element, and returns an error symbol $\perp$.
- $\mathcal{OD}(\mathcal{R}\&\mathcal{D}, \mathsf{a})$ retrieves and deletes an item (a, v) stored in the data structure (if it exists), returning v, and otherwise returns an error $\perp$ if the element is not contained.

Moreover, $\mathcal{OD}$ enables two additional "dummy" operations $\mathcal{OD}(\mathcal{R}\&\mathcal{D}, \perp)$ and $\mathcal{OD}(\mathcal{I}, \perp, \perp)$ which are meant to have no effect on the data structure. Informally, for security, we demant that the access patterns resulting from any two equally long sequences of operations of type $\mathcal{OD}(\mathcal{I}, *, *)$ and $\mathcal{OD}(\mathcal{R}\&\mathcal{D}, *)$ are (statistically) indistinguishable.[7]

The Implementation. We can easily obtain the above $\mathcal{OD}$ data structure using for instance any Cuckoo-hashing based dictionary data structure with constant worst-case access complexity.[8]

Theorem 3 (Efficient Cuckoo-Hashing Based Dictionary [10]). *There exists an implementation of a dictionary data structure holding at most n blocks with the following properties: (1) It stores $n' = O(n)$ blocks in the memory. (2) Every insert, delete, and lookup operation, requires $c = O(1)$ accesses to blocks in memory. (3) The client stores* $\mathrm{polylog}(n)$ *blocks in local memory.*

[7] In fact, for our purposes, we could leak which operations are of which type, but it will be easy enough to achieve this even stronger notion.

[8] We think of a data structure as being in a simliar model as our ORAM scheme, namely consisting of a client interface, using a small amount of local memory, and the actual data being stored externally on the server.

(4) The failure probability is negligible (in n) for any poly(n)-long sequence of lookups, insertions, and deletions which guarantees that at most n elements are ever stored in the data structure.

From any ORAM scheme $\mathcal{O}$ with address space n', it is easy to implement the oblivious data-structure $\mathcal{OD}$: The client simply implements the dictionary data structure from Theorem 3 on top of the ORAM's logical address space, and uses additional polylog(n) local memory for managing this data structure. Dummy accesses can be performed by simply issuing c arbitrary read requests to the ORAM storage. We omit a formal analysis of this construction, which is immediate.

4.2 The Generic OPRAM Protocol

We finally show how to obtain our main generic construction of an oblivious parallel RAM: The server storage consists of m partitions, and the i-th client manages the i-th partition. In particular, client i runs the oblivious dictionary scheme $\mathcal{OD}$ presented above (we refer to its interface as $\mathcal{OD}_i$) on the i-th partition. Here, we assume that the clients have access to the partition map, mapping each address $\mathsf{a} \in [N]$ to some partition $\mathsf{partition}[\mathsf{a}]$. (We will discuss in the analysis how to eliminate this sharing using recursion.) Besides, Generic-OPRAM further takes care of the communication among clients using the algorithms $\mathsf{OblivElect}$, $\mathsf{OblivMCast}$, $\mathsf{OblivRoute}$ from Sect. 3.2.

We postpone a complexity, correctness, and security analysis to Appendix E, as well as a discussion of the recursion version.

Data Structures. The non-recursive version of Generic-OPRAM keeps a *partition map* with N entries that maps block addresses a to their currently assigned partition $\mathsf{partition}[\mathsf{a}]$, and that can be accessed by all clients obviously (i.e., access to the partition map are secret). Every client additionally keeps a *stash* SS_i which contains at most R items to be inserted into $\mathcal{OD}_i$. For our analysis to work out, we need $R = \omega(\log \lambda)$. Also let $\kappa \geq 2$ be a constant.

Generic OPRAM Protocol. In each iteration, given the logical access requests $(\mathsf{Acc}(op_i, \mathrm{addr}_i, v_i))_{i\in[m]}$ input to the client, the m clients go through the following steps (all messages are tacitly encrypted with fresh random coins):

1. **Select block representatives.** Run $\mathsf{OblivElect}$ between clients with inputs $(\mathsf{a}_i, \mathrm{op}_i)_{i\in[m]}$. In the end, each client i knows whether it has been selected as the representative to get the block value a_i, or not.
2. **Query blocks.** Clients do one of two things:
   - **Real requests.** Each representative client i gets the partition index $p_i = \mathsf{partition}[\mathsf{a}_i]$, and sends a request a_i to client p_i. Moreover, it reassigns $\mathsf{partition}[\mathsf{a}_i] \xleftarrow{\$} [m]$.
   - **Fake requests.** Every non-representative client i generates a random $q_i \xleftarrow{\$} [m]$ and sends a request $\perp$ to client q_i.

3. **Retrieve the blocks.** Each client $p \in [m]$ processes the received requests according to some random ordering: For each request $\mathsf{a}_i \neq \perp$ received from client i, client p executes $\mathcal{OD}_p(\mathcal{R}\&\mathcal{D}, \mathsf{a}_i)$ and denote the retrieved block value $\overline{v}_i$. If $\overline{v}_i = \perp$, then there must be some entry $(\mathsf{a}_i, \overline{v}'_i)$ in the SS_p. Then, client p deletes this entry, and sets $\overline{v}_i = \overline{v}'_i$. Finally, it sends $\overline{v}_i$ back to i. For every $\perp$ request received from some client i, client p executes the fake read access $\mathcal{OD}_p(\mathcal{R}\&\mathcal{D}, \perp)$, and returns $\overline{v}_i = \perp$ to i.
4. **Representatives inform.** At the end of the previous step, each client holds a value $\overline{v}_i$ which is $\neq \perp$ if and only if i is the representative for a_i. Next, the m clients run sub-protocol $\mathsf{OblivMCast}$ to allow each representative to multicast the value it holds to other clients requesting the same block: Concretely, each client i uses input $(\mathsf{a}_i, \overline{v}_i)$ (recall a non-representative has $\overline{v}_i = \perp$) and receives output $\overline{v}'_i$, which is guaranteed to be the value of block a_i it requests. Each client i answers its logical request with $\overline{v}'_i$.
5. **Send updated values.** For each representative i such that $\mathsf{Acc}(op_i, \mathsf{a}_i, v_i)$ is a *write* command, let $\mathrm{id}_i = \mathsf{partition}[\mathsf{a}_i]$ and $\mathrm{msg}_i = (\mathsf{a}_i, v_i)$. Otherwise, if it is *not* a write command (but still, i a representative), it sets $\mathrm{msg}_i = (\mathsf{a}_i, \overline{v}_i)$ instead. Non-representative clients set $\mathrm{msg}_i = \perp$ and $\mathrm{id}_i \xleftarrow{\$} [m]$. Then, the clients run $\mathsf{OblivRoute}$ with respective inputs $(\mathrm{id}_i, \mathrm{msg}_i)$.
6. **Write back.** Each client $p \in [m]$ adds all pairs (a, v) received through $\mathsf{OblivRoute}$ to SS_p. Then, client p picks the first κ elements from SS_p, and for each such element (a, v), executes $\mathcal{OD}_i(\mathcal{I}, \mathsf{a}_i, v)$. If $\kappa' < \kappa$ elements are in SS_i, then the last $\kappa - \kappa'$ insertions are dummy insertions $\mathcal{OD}_i(\mathcal{I}, \perp, \perp)$. Anytime when stash SS_i needs to store more than R blocks or the partition holds more then $2N/m + R$ blocks, output "$\mathsf{overflow}$" and halt.

Acknowledgments. The authors wish to thank Elette Boyle, Kai-Min Chung, and Mariana Raykova for insightful discussions.

Binyi Chen was partially supported by NSF grants CNS-1423566 and CNS-1514526, and a gift from the Gareatis Foundation. Huijia Lin was partially supported by NSF grants CNS-1528178 and CNS-1514526. Stefano Tessaro was partially supported by NSF grants CNS-1423566, CNS-1528178, and the Glen and Susanne Culler Chair. This work was done in part while the authors were visiting the Simons Institute for the Theory of Computing, supported by the Simons Foundation and by the DIMACS/Simons Collaboration in Cryptography through NSF grant CNS-1523467.

A Correctness and Obliviousness of ORAM

For an access sequence $\boldsymbol{y}$ we let $\mathsf{AP}_i = \mathsf{AP}_i(\boldsymbol{y})$ be the access pattern of its i-th operation – i.e., the sequence of pairs $(\overline{\mathrm{op}}_{i,1}, \overline{a}_{i,1}), \ldots, (\overline{\mathrm{op}}_{i,q_i}, \overline{a}_{i,q_i})$ describing the client's server accesses (*without* the actual values) when processing the i-th operation – and denote by val_i the answer of this operation. Then, we let

$$\mathsf{Out}_\mathcal{O}(\lambda, N, B, \boldsymbol{y}) = (\mathrm{val}_1, \mathrm{val}_2, ..., \mathrm{val}_T)\ , \quad \mathsf{AP}_\mathcal{O}(\lambda, N, B, \boldsymbol{y}) = (\mathsf{AP}_1, ..., \mathsf{AP}_T).$$

We say that the sequence of outputs $\boldsymbol{z} = \mathsf{Out}_\mathcal{O}(\lambda, N, B, \boldsymbol{y})$ of $\mathcal{O}$ is correct w.r.t. the sequence of logical accesses $\boldsymbol{y}$, if for each logical command $\mathsf{Acc}(\mathrm{op}_i, \mathsf{a}_i, v_i)$ in

$\boldsymbol{y}$, the corresponding output val_i in $\boldsymbol{z}$ is *either* the most recently written value on address a_i, *or* $\perp$ if a_i has not yet been written to. Let $\mathsf{Correct}$ be the predicate that on input $(\boldsymbol{y}, \boldsymbol{z})$ returns whether $\boldsymbol{z}$ is correct w.r.t. $\boldsymbol{y}$.

Definition 2 (ORAM Correctness and Security). *An ORAM $\mathcal{O}$ achieves correctness and obliviousness if for all $N, T, B = poly(\lambda)$, there exists a negligible function μ, such that, for every λ, every two sequences $\boldsymbol{y}$ and $\boldsymbol{y'}$ of $T(\lambda)$ access operations, the following are satisfied:*

1. ***Correctness:*** $\Pr\left[\mathsf{Correct}\left(\boldsymbol{y}, \mathsf{Out}_{\mathcal{O}}(\lambda, N, B, \boldsymbol{y})\right) = 1\right] \geq 1 - \mu(\lambda)$.
2. ***Obliviousness:*** $\Delta\left(\mathsf{AP}_{\mathcal{O}}(\lambda, N, B, \boldsymbol{y}), \mathsf{AP}_{\mathcal{O}}(\lambda, N, B, \boldsymbol{y'})\right) \leq \mu(\lambda)$.

We note that the above definition considers *statistical* obliviousness. This is generally achieved by tree-based ORAM schemes, but it can be relaxed to computational obliviousness, where the statistical distance is replaced by the best distinguishing advantage of a PPT distinguisher.

B Review of Path-ORAM

In this section, we review the Path-ORAM scheme in detail, as it is used as a starting point for Subtree-ORAM and Subtree-OPRAM.

Overview. Path-ORAM is a tree-based ORAM that works for the single client setting. To implement a logical storage space for N data blocks, Path-ORAM organizes the storage space (virtually) as a complete binary tree with depth $D = \lceil \log N \rceil$. Each node of the tree is a bucket capable of storing Z blocks of size B (bits). Here Z is a constant, thus the server storage overhead is $O(1)$. To hide the logical access patterns, each data block a is assigned to a random path ℓ from root to leaf in the tree and stored at some node of the path; in order to hide the repetitive accesses to the same block, the assignment is updated to a new independent random path after each access.

In [26], Path-ORAM is constructed in two steps; first, a non-recursive version is proposed and analyzed, in which the client keeps a local position map with $N \log N$ bits; then the position map is recursively outsourced to the server, reducing the client storage to only polylog(N) bits. Below we describe the non-recursive version first, and then show how to apply the recursive transformation.

Non-Recursive Version. The client maintains a *position map* that maps each block a to a path $\mathsf{pos.map}(\mathsf{a})$. Since each path can be specified using D (the depth of the tree) bits, the size of the position map is $ND = N\lceil \log N \rceil$ bits. Additionally, the client keeps a small local storage *stash* used for storing blocks that do not fit in the assigned path (due to limited space at each tree node). The capacity of the stash is bounded by $R = R(\lambda)$ for any function $R(\lambda) = \omega(\log \lambda)$, except with negligible probability in λ.

Given the i-th logical access $\mathsf{Acc}(op_i, \mathsf{a}_i, v_i)$, Path-ORAM proceeds in two phases:

- **Phase 1: Processing the query.** Path-ORAM retrieves the path $\ell_i = \mathsf{pos.map}(\mathsf{a}_i)$ assigned to block a_i, and finds the block a_i on the path or in the stash. After returning the block value and potentially updating the block, Path-ORAM re-assigns block a to a new independent random path $\ell' \xleftarrow{\$} [N]$ and updates $\mathsf{pos.map}(\mathsf{a}_i) = \ell'_i$. It then moves the block to the stash.
- **Phase 2: Flushing and write-back.** Before re-encrypting and writing the path back to the server, in order to avoid the stash from "overflowing", Path-ORAM re-arranges path ℓ_i, to fit as many blocks from the stash into the path. More specifically, for each block a_j in the stash and on the path, Path-ORAM places it at the lowest non-full node p_j that intersects with its assigned path $\ell_j = \mathsf{pos.map}(\mathsf{a}_j)$. If no such node is found, the block remains in the stash. If at any point, the stash contains more than R blocks, Path-ORAM outputs "$\mathsf{overflow}$" and aborts.

Recursive Version. In the above non-recursive version, the client keeps a large $N \log N$-bit position map. To reduce the client storage, Path-ORAM recursively outsources the position map to the server by adding extra $O(\log N)$ trees. More specifically, if the cell size $B \geq \alpha \lceil \log N \rceil$ for some integer $\alpha > 1$, the position map can be stored in $\lceil \frac{N}{\alpha} \rceil$ cells. This means, to answer an access to a logical storage space of size N, the non-recursive version only needs to make a query to another logical storage space (i.e. the position map) of size $\lceil \frac{N}{\alpha} \rceil$. Therefore, if the client further outsources the position map to the server, its local storage would be reduced to $\lceil \frac{N}{\alpha^2} \rceil$. This idea can be applied recursively until the client storage becomes polylog(N). In the final scheme, at the server, besides tree $\mathcal{T}_0$ that stores data blocks, there are additional trees $r\mathcal{T}_1, r\mathcal{T}_2, ..., r\mathcal{T}_l$ for position map queries, where $l = \lceil \log_\alpha N \rceil$. Tree $r\mathcal{T}_i$ has size $\tilde{O}(\lceil \frac{N}{\alpha^i} \rceil B)$ bits, and maintains the position map corresponding to tree $r\mathcal{T}_{i-1}$ which contains $\lceil \frac{N}{\alpha^{i-1}} \rceil$ cells. The position map corresponding to tree $r\mathcal{T}_l$ is stored in local storage. Now, to access a block a, the client needs to query $\mathsf{pos.map}(\mathsf{a})$ in $\mathcal{T}_0$ by looking up the position in tree $r\mathcal{T}_1$. In order to query the position map corresponding to tree $r\mathcal{T}_1$, similarly, the client looks up in $r\mathcal{T}_2$, so on and so forth. Finally the position map value of tree $r\mathcal{T}_l$ is stored in local storage.

Complexity. The storage overhead is $O(1)$ both in the non-recursive version and the recursive version. In the non-recursive version, for each logical access, Path-ORAM reads and writes a path with $\log N$ nodes, each of which contains Z cells, therefore the communication overhead is $O(\log N)$ per access. The computation overhead is $O(\log^2 N)$, since the flushing procedure takes time $O(\log^2 N)$ per access. After the recursive transformation is applied, to answer each logical access, the client needs to query $l = \lceil \log_\alpha N \rceil$ number of trees, and hence the communication/computation overhead blow by a factor of $\log N$.

C Analysis of Subtree-ORAM

In the full version, we show that the overflow probability of Subtree-ORAM is negligible given any sequence of logical access requests. In particular, we prove the following proposition, which generalizes the analysis of Path-ORAM.

Proposition 1. *Fix the stash size to any $R(\lambda) \in \omega(\log \lambda)$. For every polynomial m, N, T, B, there exists a negligible function μ, such that, for every λ, and sequence $\boldsymbol{y}$ of T batches of m access requests, the probability that Subtree-ORAM outputs* overflow *is at most $\mu(\lambda)$.*

From Proposition 1, it is easy to show that Subtree-ORAM satisfies correctness and obliviousness.

- Correctness: Since the stash overflows with negligible probability, and Subtree-ORAM maintains the *block-path invariance* (as in Path-ORAM) – at any moment, each block can be found either on the path currently assigned to it or in the stash; by construction, Subtree-ORAM answers logical accesses correctly according to the correctness condition of ORAM.
- Obliviousness: Conditioned on no overflowing: (1) In each iteration, Subtree-ORAM always reads m independent and random paths from the server. (2) After each iteration, every requested block is assigned to a new random path, which is hidden from the adversary (as in Path-ORAM). Thus the construction is oblivious.

D Analysis of Subtree-OPRAM

In this section, we give a high-level overview of why Subtree-OPRAM is correct and satisfies obliviousness. Also we discuss below the complexity of the protocol.

We discuss correctness and obliviousness for the non-recursive version only. The same properties are then also easily shown to be true for the recursive version. The first key observation is that the m clients of Subtree-OPRAM can be seen as collectively emulating the operations of the single client of Subtree-ORAM. Compare an execution of Subtree-OPRAM with a sequence $\boldsymbol{y}$ of T batches of m parallel logical accesses with the execution of Subtree-ORAM with the same sequence.[9] Then, we observe the following:

1. The ORAM tree $\mathcal{T}$ of Subtree-ORAM is stored in parts in Subtree-OPRAM: All but the top $\log m$ levels is stored as the m-tree forest $\mathcal{T}_1, \cdots, \mathcal{T}_m$ at the server, while the top $\log m$ levels are stored in a distributed way by the individual clients in their respective stashes — namely, if a block is stored at client j, its assigned path belongs to $\mathcal{T}_j$. Therefore, the union of $\{\mathsf{stash}_i\}$ contains the same blocks as the stash of Subtree-ORAM, as well as all blocks in the top of the tree $\mathcal{T}$ of Subtree-ORAM.

[9] We are being somewhat informal here – one would have to define precisely what it means to "compare" in terms of executing both protocols with the same random choices. As it is somewhat tedious, we keep this on a more informal high level, hoping to convey the main ideas.

2. Subtree-OPRAM answers a batch of m requests (in each iteration) as Subtree-ORAM does: The m clients of Subtree-OPRAM choose a representative for each requested block (in Step 1) with the exactly same rule Subtree-ORAM uses to remove repetitive accesses, and to only keep one access per block. Later (in Step 4), Subtree-OPRAM first answers requests using the most recently written value from previous iterations and then executes the write operations; in particular, due to the way representatives are chosen, the write operation with the minimal index always takes effect.
3. Subtree-OPRAM maintains the *block-path invariant* as Subtree-ORAM. This is because each time a block is assigned to a new path ℓ', it is sent (using the $\mathsf{OblivRoute}$ sub-protocol) to client j managing the tree $\mathcal{T}_j$ the path ℓ' belongs to. Therefore, at any moment, a block is either on its assigned path or in the local stash of the client responsible for the tree its assigned path belongs to.
4. Subtree-OPRAM emulates the flushing procedure of Subtree-ORAM: Recall that Subtree-ORAM flushes along the subtree $\mathcal{T}_{\text{real}}$ of paths assigned to all requested blocks. Removing the top $\log m$ levels of $\mathcal{T}_{\text{real}}$ gives a set of m subtrees $\mathcal{T}_{\text{real}_1}, \cdots, \mathcal{T}_{\text{real}_m}$. Note that $\mathcal{T}_{\text{real}_i}$ is exactly the subtree that client i in Subtree-OPRAM performs flushing on (in Step 6). Indeed, by the design of the subtree flushing procedure, blocks that land in different subtrees $\mathcal{T}_{\text{real}_i} \neq \mathcal{T}_{\text{real}_j}$ can be operated on independently. Moreover, blocks that would land in the top $\log m$ levels of $\mathcal{T}_{\text{real}}$ or stash in Subtree-ORAM are naturally divided into the m local stashes according to which tree $\mathcal{T}_j$ their assigned path belongs to.

Correctness and Stash Analysis. By the above, if we fix any sequence $\boldsymbol{y}$ of parallel accesses, and consider the executions of (non-recursive) Subtree-OPRAM and (non-recursive) Subtree-ORAM with the same input sequence $\boldsymbol{y}$, since Subtree-ORAM answers every request correctly as long as it does not overflow, so does Subtree-OPRAM.

To argue that Subtree-OPRAM only overflows with negligible probability, recall that by Proposition 1, when the stash size of Subtree-ORAM is set to any $R'(\lambda) \in \omega(\log(\lambda))$, the probability of overflowing is negligible. We can thus bound the size of each local stash stash_i in Subtree-OPRAM, using the bound on the stash size of Subtree-ORAM. As noted above, after each iteration, the local stash stash_i of client i stores two types of blocks:

1. Blocks in the stash of Subtree-ORAM with an assigned path belonging to $\mathcal{T}_i$, and
2. Blocks in the top $\log m$ levels of the ORAM tree $\mathcal{T}$ of Subtree-ORAM, again with an assigned path belonging to $\mathcal{T}_i$.

By Proposition 1, the number of blocks of the first type is bounded by $\omega(\log \lambda)$ with overwhelming probability. Moreover, it is easy to see that the number of blocks of the second type is bounded by $O(\log m)$. Therefore, the size of stash_i is bounded by any $R(\lambda, m) \in \omega(\log \lambda) + O(\log m)$ with overwhelming probability. This is summarized by the following lemma.

Lemma 1. *Fix the stash size to any* $R(\lambda, m) \in \omega(\log \lambda) + O(\log m)$. *For every polynomial* m, N, T, B, *there is a negligible function* μ, *such that, for every* λ, *and sequence* $\boldsymbol{y}$ *of* $T(\lambda)$ *accesses, the probability that any client of the non-recursive Subtree-OPRAM outputs* overflow *is at most* $\mu(\lambda)$.

Complexity. The storage overhead of Subtree-OPRAM is the same as that of Path-ORAM, which is $O(1)$. The only contents stored at each client are the stashes, one per recursion level. Since each stash is of size $R(\lambda, m)B$, and the recursion depth is bounded by $O(\log N)$, the total client storage overhead is $O(\log N)R(\lambda, m) \in \omega(\log N \log \lambda) + O(\log N \log m)$.

Next, we analyze the communication and computation overheads (per client per access) of the recursive Subtree-OPRAM. In each iteration, to process m logical accesses (one per client), the m clients first recursively look up the position maps for $O(\log N)$ times using the non-recursive Subtree-OPRAM, and then process their requests using again the non-recursive Subtree-OPRAM. Fix any client i, we analyze its communication and computation complexities as follows:

- Server communication overhead: In each invocation of non-recursive Subtree-OPRAM, client i reads/writes a subtree of paths delegated to it by other clients. Since these paths are all chosen at random, in expectation client i read/write only 1 path in each invocation. Furthermore, across all $O(\log N)$ invocations of non-recursive Subtree-OPRAM, the probability that client i is delegated to read/write $\omega(\log \lambda) + O(\log N)$ paths is negligible. (Consider tossing $O(\log N) \times m$ balls (read/write path requests) randomly into m bins (clients); the probability that any bin has more than $O(\log N) + \omega(\log \lambda)$ balls is negligible in λ.) Since each path contains $O(\log N)$ blocks, the server communication overhead is bounded by $\omega(\log \lambda \log N) + O(\log^2 N)$ in the worst case, with overwhelming probability.
- Inter-client communication overhead: In each invocation of the non-recursive Subtree-OPRAM protocol, client i communicates with other clients in two ways: (1) using the oblivious sub-protocols (Steps 1, 4 and 5) and (2) sending the requests for reading certain block and path $(i, \mathsf{a}'_i, \ell_i)$ (Step 2) and sending back the retrieved block (Step 3). The maximum communicating complexity of the oblivious sub-protocols is $O(K \log m(\log m + \log N + B))$ bits, where K is in $\omega(\log \lambda)$. Therefore, across $O(\log N)$ invocations of non-recursive Subtree-OPRAM, the first type of inter-client communication involves sending/receiving at most $O(\log NK \log m(\log m + \log N + B))$ bits. On the other hand, by a similar argument as above, across $O(\log N)$ recursive invocations, with overwhelming probability, each client receives at most $O(\log N) + \omega(\log \lambda)$ requests of form $(i, \mathsf{a}'_i, \ell_i)$, and hence the second type of communication involves sending/receiving $(\omega(\log \lambda) + O(\log N)) \times O(\log m + \log N + B)$ bits with overwhelming probability. Thus, in total, the inter-client communication is $\omega(\log \lambda) \log m \log N(\log m + \log N + B)$ bits. Since $B \geq \alpha \log N$ for an $\alpha > 1$, the inter-client communication overhead is $\omega(\log \lambda) \log m(\log m + \log N)$.

Finally, we observe that when considering the communication overhead averaged over a sufficiently large number T of parallel accesses, the server communication

overhead is bounded by $O(\log^2 N)$ with overwhelming probability. The inter-client communication complexity stays the same.

Obliviousness. The obliviousness of recursive Subtree-OPRAM follows from that of the non-recursive version. Conditioned on that the stash does not overflow, the latter follows from three observations: (i) In each iteration, the paths $\{\ell_i\}$ read/write from/to the server (in Steps 3 and 6) are all independent and random, (ii) the communication between different clients is either through one of the oblivious sub-protocols (in Steps 1, 4, and 5), which has fixed communication pattern, or depends on the random paths $\{\ell_i\}$ (in Steps 2 and 3), and (iii) the new assignment of paths $\{\ell'_i\}$ to blocks accessed are hidden using $\mathsf{OblivRoute}$ (in Step 5). Combining these observations, we conclude that the access and communication patterns of Subtree-OPRAM is oblivious of the logical access pattern.

E Analysis of Generic-OPRAM

Recursive Version. The above protocol assumes that every client has (private) access to the partition map to be accessed and updated throughout the execution of the protocol. This is of course not realistic. But similar to the case of the position map in Subtree-OPRAM, we can use $O(\log N)$-deep recursion. For this to work, we need block size to be at least, say, $B = 2 \log m$, since each entry in the partition map can be represented by $\log m$ bits.

Complexity Analysis. We now analyze the complexity of the Generic-OPRAM. We assume that $\mathcal{OD}_i$ is implemented from some ORAM scheme $\mathcal{O}_i$ which has communication overhead $\alpha(N')$ when using address space N', and that the same scheme stores $M(N')$ blocks on the server for the same address space. We make some assumptions in the following that appear reasonable, namely that $\alpha(O(N')) = O(\alpha(N'))$ and $M(O(N')) = O(M(N'))$ (this is true because these functions are polynomial). Moreover, we can also assume also that $M(N'/c) \leq M(N')/c$ for any constant c. (Note that the scheme is meaningful without these assumptions, but the resulting complexity statement would be somewhat more cumbersome.)

- **Server and Client Storage.** Let us start with the non-recursive case. Note that each partition needs enough blocks to implement a dynamic data structure to store $2N/m + R$ blocks. This will require an ORAM for $N' = O(N/m + R)$ blocks, which thus requires $O(M(N/m + R))$ blocks. Thus, the overall server storage complexity is of $O(m \cdot M(N/m + R))$ blocks. If $M(N') = O(N')$, in particular this implies that the overall storage complexity is $O(N + mR)$, and thus linear if $m \cdot R \in O(N)$.
  For the recursive case, note that the storage space is going to at least halve after each recursion level by our assumption on M. So if we assume that $N/m > R$, we see that the storage complexity remains $O(m \cdot M(N/m + R))$. Every client needs to store R blocks, and moreover, it needs polylog(N) memory for implementing $\mathcal{OD}$ and the underlying ORAM scheme $\mathcal{O}$, which we assume to have polylog(N) client storage overhead.

- **Server Communication.** In contrast to Subtree-OPRAM above, a generic construction does not necessarily allow us to parallelize accesses to the data structure. The number of $\mathcal{OD}_i(\mathcal{R}\&\mathcal{D}, \cdot)$ operations a client performs can thus vary in each round, but we can apply the same analysis as for Subtree-OPRAM above. Namely, given we are using $\log N$ levels of recursion, the per-client server communication is $O(\log N \cdot \alpha(N/m + R))$ in the amortized case, and $O((\log N + \omega(\log \lambda)) \cdot \alpha(N/m + R))$ in the worst case.
- **Inter-Client Communication.** The analysis is the same as the one for Subtree-OPRAM.

Correctness. We analyze our scheme and show that it is indeed a valid OPRAM scheme.

Lemma 2. *Generic-OPRAM satisfies correctness, and in particular only overflows with negligible probability, as long as $\mathcal{OD}$ is also correct and only fails with negligible probability.*

We omit part of the correctness proof, and restrict ourselves to the more involved part of the analysis, proving that none of the stashes $\mathsf{SS_i}$ ever overflows, and that none of the partition is supposed to hold more than $2N/m+R$ elements. Conditioned on no overflows, correctness can then be verified by inspection.

The final result on the overflow probability summarized by the following lemma.

Lemma 3. *For every constant $\kappa \geq 2$, every $T = T(\lambda)$, and every logical access sequence $\boldsymbol{y}$ of T batches of m parallel logical instructions,*

$$\Pr[\textit{The protocol outputs } \text{“}\mathsf{overflow}''] \leq T \cdot m \cdot e^{-\Theta(R)},$$

where the randomness is taken over the partition assignment, and the constant in the exponent depends on κ only.

We split the proof of the lemma into two propositions – the first pertaining to $\mathsf{SS_i}$ overflowing, the second to partition load. We stress that the proof first proposition relies on some interesting (and non-elementary) fact from basic queueing theory to ensure that a constant outflow of (at most) two blocks is sufficient to avoid an overflow.

Proposition 2. *For every constant $\kappa \geq 2$, every $T = T(\lambda)$, and every logical access sequence $\boldsymbol{y}$ of T batches of m logical instructions,*

$$\Pr[\textit{One of the stashes } \mathsf{SS_i} \textit{ overflows}] \leq T \cdot m \cdot e^{-\Theta(R)},$$

where the randomness is taken over the partition assignment, and the constant in the exponent depends on κ only.

Proof. We prove the lemma for $\kappa = 2$. It will be clear that the bound only improve for larger $\kappa > 2$. Let us look at what happens with one particular stash $\mathsf{SS_i}$ for some $i \in [m]$ over time, and compute the probability that it ever contains more than R elements. We model this via the following process:

Single-bin process: In each iteration, m balls are thrown into one out of m bins independently, and each one lands in the single bin we are looking at with probability $1/m$. Then, $\kappa = 2$ balls are taken out of the bin (if the bin contains at least $\kappa = 2$ balls), and otherwise the bin is emptied.

Note that in the actual protocol execution, less than m balls may be thrown into bins at each round because of possible repetition patterns, but it is clear that by always assuming that up to potential m balls can be thrown in the bin can only increase the probability of overflowing, and thus this will be assumed without loss of generality.

To analyze the probability that the bin overflows at some point in time (i.e., it contains more than R balls), we use the stochastic process proposed in Example 23 of [22], with $a = 0, b = +\infty$. There, it is shown that the number of balls in the bin at iteration T is distributed as the random variable

$$X_T = \max_{0 \le i \le T} Z_i$$

where $Z_0 = 0$ and

$$Z_i = \sum_{j \le i} (V_j - U_j),$$

with V_j denoting the number of balls going to the bin in iteration j and U_j denoting the *potential* number of balls taken out from the bin in iteration j. Here $U_j = 2$ for every j, thus

$$Z_i = T_i - 2i,$$

where $T_i = \sum_{j \le i} V_j$. Note that V_j can be seen as the sum of m independent Bernoulli random variables, each being one with probability $1/m$. Therefore, T_i is the sum of $m \cdot i$ Bernoulli random variables with expected value i. We want to show now that with very high probability, $T_i \le 2i + R$. We can simply use the Chernoff bound, and consider two cases. First, if $R/i \ge 1$, then

$$\Pr\left[T_i \ge 2i + R\right] \le \Pr[T_i \ge i \cdot (1 + R/i)] \le e^{-\frac{\varepsilon^2}{2+\varepsilon} i},$$

where $\varepsilon = R/i$. Note that

$$\frac{\varepsilon^2}{2+\varepsilon} i = R \cdot \frac{1}{1 + 2i/R} \ge R/3.$$

Thus $\Pr\left[T_i \ge 2i + R\right] \le e^{-R/3}$. The second case is that $R/i \le 1$. Then,

$$\Pr\left[T_i \ge 2i + R\right] \le \Pr[T_i \ge i \cdot (1 + 1)] \le e^{-i/3} \le e^{-R/3}.$$

Therefore, by the union bound, the probability that there exists some i such that $Z_i \ge R$ is at most $T \cdot e^{-R/3}$, i.e., $X_T \le R$, except with probability $T \cdot e^{-R/3}$. To conclude, once again by the union bound, we obtain the bound on the probability that one of the m stashes overflows. □

We also need to analyze the probability that too many elements are assigned to one partition, as otherwise our protocol would also fail.

Lemma 4. *For a given partition map* $\mathsf{partition} : [N] \to [m]$*, denote by* L *the maximum numbers of addresses* $\mathsf{a} \in [N]$ *assigned to the same partition* p*. Then, for any sequence of* T *batches of* m *operations and any* $R \geq 2$*, the probability that any point in time,* $L \geq 2N/m + R$ *ist at most* $T \cdot m \cdot e^{-R/2}$*.*

Proof. Take the partition map contents at some fixed point in time, and fix some partition $i \in [m]$. The entire contants of the partition map are N independent random variables, and each one of them is equal to i with probability $1/m$. Let L^i be the number of addresses assigned to this given i, and let $L = \max_i L^i$. Note that L^i is a sum of Bernoulli random variables with expectation N/m. We can then use the Chernoff bound to see that

$$\Pr\left[L^i \geq 2N/m + R\right] = \Pr\left[S \geq N/m(1 + 1 + \varepsilon)\right] \leq e^{-R/2},$$

for $\varepsilon = Rm/N$. By the union bound,

$$\Pr\left[L \geq N/m + R\right] \leq m \cdot e^{-R/2}.$$

And finally, note that there are at most T different "assignments" of position maps due to the structure of the protocol, and thus the overall bound on the probability follows – once again – by the union bound. □

Obliviousness. Generic-OPRAM also satisfies the obliviousness property. The formal proof (which we omit) relies on the obliviousness of the $\mathcal{OD}_i$'s and the fact that whenever processing a batch of m logical accesses, the above scheme accesses first m randomly chosen partitions, and moreover, in the second phase, each partition is accessed exactly twice.

References

1. Boneh, D., Mazieres, D., Popa, R.: Remote oblivious storage: making oblivious ram practical. MIT Tech-report: MIT-CSAIL-TR-2011-018 (2011)
2. Boyle, E., Chung, K.-M., Pass, R.: Oblivious parallel RAM. In: Kushilevitz, E., Malkin, T. (eds.), TCC 2016A, LNCS (2016, To appear). http://eprint.iacr.org/2014/594
3. Chung, K.-M., Liu, Z., Pass, R.: Statistically-secure ORAM with $\tilde{O}(\log^2 n)$ overhead. In: Sarkar, P., Iwata, T. (eds.) ASIACRYPT 2014, Part II. LNCS, vol. 8874, pp. 62–81. Springer, Heidelberg (2014)
4. Chung, K.-M., Pass, R.: A simple ORAM. Cryptology ePrint Archive, Report 2013/243 (2013). http://eprint.iacr.org/2013/243
5. Fletcher, C.W., van Dijk, M., Devadas, S.: Towards an interpreter for efficient encrypted computation. In: Proceedings of the 2012 ACM Workshop on Cloud Computing Security, CCSW 2012, Raleigh, NC, USA, October 19, 2012, pp. 83–94 (2012)

6. Gentry, C., Goldman, K.A., Halevi, S., Julta, C., Raykova, M., Wichs, D.: Optimizing ORAM and using it efficiently for secure computation. In: De Cristofaro, E., Wright, M. (eds.) PETS 2013. LNCS, vol. 7981, pp. 1–18. Springer, Heidelberg (2013)
7. Goldreich, O.: Towards a theory of software protection. In: Odlyzko, A.M. (ed.) CRYPTO 1986. LNCS, vol. 263, pp. 426–439. Springer, Heidelberg (1987)
8. Goldreich, O.: Towards a theory of software protection and simulation by oblivious RAMs. In: Aho, A. (ed.) 19th ACM STOC, pp. 182–194. ACM Press, New York City, New York, USA (25–27 May 1987)
9. Goldreich, O., Ostrovsky, R.: Software protection and simulation on oblivious RAMs. J. ACM **43**(3), 431–473 (1996)
10. Goodrich, M.T., Hirschberg, D.S., Mitzenmacher, M., Thaler, J.: Cache-oblivious dictionaries and multimaps with negligible failure probability. In: Even, G., Rawitz, D. (eds.) MedAlg 2012. LNCS, vol. 7659, pp. 203–218. Springer, Heidelberg (2012)
11. Goodrich, M.T., Mitzenmacher, M.: Privacy-preserving access of outsourced data via oblivious RAM simulation. In: Aceto, L., Henzinger, M., Sgall, J. (eds.) ICALP 2011, Part II. LNCS, vol. 6756, pp. 576–587. Springer, Heidelberg (2011)
12. Goodrich, M.T., Mitzenmacher, M., Ohrimenko, O., Tamassia, R.: Oblivious RAM simulation with efficient worst-case access overhead. In: Proceedings of the 3rd ACM Cloud Computing Security Workshop, CCSW 2011, Chicago, IL, USA, October 21, 2011, pp. 95–100 (2011)
13. Goodrich, M.T., Mitzenmacher, M., Ohrimenko, O., Tamassia, R.: Oblivious storage with low I/O overhead. CoRR, abs/1110.1851 (2011)
14. Goodrich, M.T., Mitzenmacher, M., Ohrimenko, O., Tamassia, R.: Privacy-preserving group data access via stateless oblivious RAM simulation. In: Rabani, Y. (ed.) 23rd SODA, pp. 157–167. ACM-SIAM, Kyoto, Japan (17–19 January 2012)
15. Dautrich, J.L. Jr., Stefanov, E., Shi, E.: Burst ORAM: minimizing ORAM response times for bursty access patterns. In: Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20–22, 2014, pp. 749–764 (2014)
16. Kushilevitz, E., Lu, S., Ostrovsky, R.: On the (in)security of hash-based oblivious RAM and a new balancing scheme. In: Rabani, Y. (ed.) 23rd SODA, pp. 143–156. ACM-SIAM, Kyoto, Japan (17–19 January 2012)
17. Lorch, J.R., Parno, B., Mickens, J.W., Raykova, M., Schiffman, J.: Shroud: ensuring private access to large-scale data in the data center. In: Proceedings of the 11th USENIX Conference on File and Storage Technologies, FAST 2013, San Jose, CA, USA, February 12–15, 2013, pp. 199–214 (2013)
18. Maas, M., Love, E., Stefanov, E., Tiwari, M., Shi, E., Asanovic, K., Kubiatowicz, J., Song, D.: PHANTOM: practical oblivious computation in a secure processor. In: Sadeghi, A.-R., Gligor, V.D., Yung, M. (eds.) ACM CCS 2013, pp. 311–324. ACM Press, Berlin, Germany (4–8 November 2013)
19. Pinkas, B., Reinman, T.: Oblivious RAM revisited. In: Rabin, T. (ed.) CRYPTO 2010. LNCS, vol. 6223, pp. 502–519. Springer, Heidelberg (2010)
20. Ren, L., Fletcher, C.W., Kwon, A., Stefanov, E., Shi, E., van Dijk, M., Devadas, S.: Constants count: practical improvements to oblivious RAM. In: Proceedings of the 24th USENIX Security Symposium (SECURITY 2015), pp. 415–430 (2015)
21. Ren, L., Yu, X., Fletcher, C.W., van Dijk, M., Devadas, S.: Design space exploration and optimization of path oblivious RAM in secure processors. In: The 40th Annual International Symposium on Computer Architecture, ISCA 2013, Tel-Aviv, Israel, June 23–27, 2013, pp. 571–582 (2013)

22. Serfozo, R.: Basics of Applied Stochastic Processes. Springer Science & Business Media, Berlin (2009). http://www.stat.yale.edu/~jtc5/251/readings/Basics%20of%20Applied%20Stochastic%20Processes_Serfozo.pdf
23. Shi, E., Chan, T.-H.H., Stefanov, E., Li, M.: Oblivious RAM with $O((\log N)^3)$ worst-case cost. In: Lee, D.H., Wang, X. (eds.) ASIACRYPT 2011. LNCS, vol. 7073, pp. 197–214. Springer, Heidelberg (2011)
24. Stefanov, E., Shi, E.: Oblivistore: high performance oblivious cloud storage. In: 2013 IEEE Symposium on Security and Privacy (SP), pp. 253–267. IEEE (2013)
25. Stefanov, E., Shi, E., Song, D.: Towards practical oblivious ram. In: NDSS (2012)
26. Stefanov E., van Dijk, M., Shi, E., Fletcher, C.W., Ren, L., Yu, X., Devadas, S.: Path ORAM: an extremely simple oblivious RAM protocol. In: Sadeghi, A.-R., Gligor, V.D., Yung, M. (eds.) ACM CCS 13, pp. 299–310. ACM Press, Berlin, Germany, (4–8 November 2013)
27. Wang, S., Ding, X., Deng, R.H., Bao, F.: Private information retrieval using trusted hardware. In: Gollmann, D., Meier, J., Sabelfeld, A. (eds.) ESORICS 2006. LNCS, vol. 4189, pp. 49–64. Springer, Heidelberg (2006)
28. Wang, X., Hubert Chan, T.-H., Shi, E.: Circuit ORAM: on tightness of the goldreich-ostrovsky lower bound. In: Ray, I., Li, N., Kruegel, C. (eds.) Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security. ACM, Denver, CO, USA, October 12–6, 2015, pp. 850–861 (2015)
29. Williams, P., Sion, R., Carbunar, B.: Building castles out of mud: practical access pattern privacy and correctness on untrusted storage. In: Ning, P., Syverson, P.F., Jha, S. (eds.) ACM CCS 2008, pp. 139–148. ACM Press, Alexandria, Virginia, USA (27–31 October 2008)
30. Williams, P., Sion, R., Tomescu, A.: PrivateFS: a parallel oblivious file system. In: Yu, T., Danezis, G., Gligor, V.D. (eds.) ACM CCS 2012, pp. 977–988. ACM Press, Raleigh, NC, USA (16–18 October 2012)