More is Less: Perfectly Secure Oblivious Algorithms in the Multi-server Setting

T.-H. Hubert Chan[1], Jonathan Katz[2], Kartik Nayak[2,3](✉), Antigoni Polychroniadou[4], and Elaine Shi[5]

[1] The University of Hong Kong, Pokfulam, Hong Kong
hubert@cs.hku.hk

[2] University of Maryland, College Park, USA
jkatz@cs.umd.edu

[3] VMware Research, Palo Alto, USA
nkartik@vmware.com

[4] Cornell Tech, New York, USA
antigoni@cornell.edu

[5] Cornell University, Ithaca, USA
runting@gmail.com

Abstract. The problem of Oblivious RAM (ORAM) has traditionally been studied in the single-server setting, but more recently the multi-server setting has also been considered. Yet it is still unclear whether the multi-server setting has any *inherent* advantages, e.g., whether the multi-server setting can be used to achieve stronger security goals or provably better efficiency than is possible in the single-server case.

In this work, we construct a perfectly secure 3-server ORAM scheme that outperforms the best known single-server scheme by a logarithmic factor. In the process we also show, for the first time, that there exist specific algorithms for which multiple servers can overcome known lower bounds in the single-server setting.

Keywords: Oblivious RAM · Perfect security

1 Introduction

Oblivious RAM (ORAM) protocols [12] allow a client to outsource storage of its data such that the client can continue to read/write its data while hiding both the data itself as well as the client's access pattern. ORAM was historically considered in the single-server setting, but has recently been considered in the multi-server setting [1,16,17,19,21,25] where the client can store its data on multiple, non-colluding servers. Current constructions of multi-server ORAM are more efficient than known protocols in the single-server setting; in particular, the best known protocols in the latter setting (when server-side computation is

An online full version of our paper [5] is available at http://arxiv.org/abs/1809.00825.

T. Peyrin and S. Galbraith (Eds.): ASIACRYPT 2018, LNCS 11274, pp. 158–188, 2018.
https://doi.org/10.1007/978-3-030-03332-3_7

not allowed) require bandwidth $O(\log^2 N/\log\log N)$ [3,7,15,18] for storing an array of length N, whereas multi-server ORAM schemes achieve logarithmic bandwidth[1] [21].

Nevertheless, there are several unanswered questions about the multi-server setting. First, all work thus far in the multi-server setting achieves either computational or statistical security, but not *perfect security* where correctness is required to hold with probability 1 and security must hold even against computationally unbounded attackers. Second, although (as noted above) we have examples of multi-server schemes that beat existing single-server constructions, it is unclear whether this reflects a limitation of existing single-server schemes or whether there are *inherent* advantages to the multi-server setting.

We address the above questions in this work. (Unless otherwise noted, our results hold for arbitrary block size B as long as it is large enough to store an address, i.e., $B = \Omega(\log N)$.) We construct a perfectly secure, multi-server ORAM scheme that improves upon the overhead of the best known construction in the single-server setting. Specifically, we show the following — henceforth if a multi-server ORAM scheme incurs, on average, $X(N)$ bandwidth (measured in terms of number of blocks transmitted) per logical memory access on a logical memory of length N, we say that the scheme has $X(N)$ *bandwidth blowup.*

Theorem 1. *There exists a 3-server ORAM scheme that is perfectly secure for any single semi-honest server corruption, and achieves $O(\log^2 N)$ bandwidth blowup. Further, our scheme does not rely on server-side computation or server-to-server communication.*

As a point of comparison, the best known *single-server*, perfectly secure ORAM schemes require $O(\log^3 N)$ bandwidth [6,9]. While Theorem 1 holds for any block size $B = \Omega(\log N)$, we show that for block sizes $B = \Omega(\log^2 N)$ our scheme achieves bandwidth blowup as small as $O(\log N)$.

As part of our construction, we introduce new building blocks that are of independent theoretical interest. Specifically, we show:

Theorem 2. *There exists a 3-server protocol for stable compaction that is perfectly secure for any single semi-honest server corruption, and achieves $O(n)$ bandwidth to compact an array of length n (that is secret-shared among the servers). The same result holds for merging two sorted arrays of length n.*

In the single-server setting, Lin, Shi, and Xie [20] recently proved a lower bound showing that any oblivious algorithm for stable compaction or merging in the balls-and-bins model must incur at least $\Omega(n \log n)$ bandwidth. The balls-and-bins model characterizes a wide class of natural algorithms where each element is treated as an atomic "ball" with a numeric label; the algorithm may perform arbitrary boolean computation on the labels, but is only allowed to

[1] Although Lu and Ostrovsky [21] describe their multi-server scheme using server-side computation, it is not difficult to see that it can be replaced with client-side computation instead.

move the balls around and not compute on their values. Our scheme works in the balls-and-bins model, and thus shows *for the first time* that the multi-server setting can overcome known lower bounds in the single-server setting for oblivious algorithms. Furthermore, for stable compaction and merging no previous multi-server scheme was known that is asymptotically faster than existing single-server algorithms, even in the weaker setting of computational security. We note finally that our protocols are asymptotically optimal since clearly any correct algorithm has to read the entire array.

1.1 Technical Roadmap

Oblivious sorting is an essential building block in hierarchical ORAM schemes. At a high level, our key idea is to replace oblivious sorting, which costs $O(n \log n)$ time on an array of length n, with cheaper, linear-time operations. Indeed, this was also the idea of Lu and Ostrovsky [21], but they apply it to a computationally secure hierarchical ORAM. Prior single-server ORAM schemes are built from logarithmically many cuckoo hash tables of doubling size. Every time a memory request has been served, one needs to merge multiple stale cuckoo hash tables into a newly constructed cuckoo hash table — this was previously accomplished by oblivious sorting [3,15,18]. Lu and Ostrovsky show how to avoid cuckoo hashing, by having one *permutation server* permute the data in linear time, and by having a separate *storage server*, that is unaware of the permutation, construct a cuckoo hash table from the permuted array in linear time (with the client's help). Unfortunately, Lu and Ostrovsky's technique fails for the perfect security context due to its intimate reliance on pseudorandom functions (PRFs) and cuckoo hashing — the former introduces computational assumptions and the latter leads to statistical failures (albeit with negligible probability).

We are, however, inspired by Lu and Ostrovsky's permutation-storage-separation paradigm (and a similar approach that was described independently by Stefanov and Shi [25]). The key concept here is to have one permutation-server that permutes the data; and have operations and accesses be performed by a separate storage server that is unaware of the permutation applied. One natural question is whether we can apply this technique to directly construct a linear-time multi-server oblivious sorting algorithm — unfortunately we are not aware of any way to achieve this. Chan et al. [4] and Tople et al. [27] show that assuming the data is already randomly permuted (where the permutation is hidden), one can simply apply any comparison-based sorting algorithm and it would retain obliviousness. Unfortunately, it is well-known that comparison-based sorting must incur $\Omega(n \log n)$ time, and this observation does not extend to non-comparison-based sorting techniques since in general RAM computations (on numeric keys) can leak information through access patterns.

New techniques at a glance. We propose two novel techniques that allow us to achieve the stated results, both of which rely on the permutation-storage-separation paradigm:

- Despite known lower bounds in the single-server setting [20], we show that with multiple servers, we can indeed achieve linear-time oblivious stable compaction and merging. As prior works [3,4,7,14] observe, merging and compaction are important building blocks in designing oblivious algorithms — we thus believe that our new building blocks are of independent interest.
- We use the linear-time oblivious stable compaction and merging algorithms to design a three-server ORAM. We adapt the single-server perfect ORAM scheme by Chan et al. [6] into a new multiserver variant to save a logarithmic factor. Specifically, in Chan et al. [6], the reshuffling operation was realized with oblivious sorting. This operation can now be expressed entirely with linear-time merging and stable compaction operations without relying on oblivious sorting.

Stable Compaction and Merging. We first explain the intuition behind our stable compaction algorithm. For simplicity, for the time being we will consider only 2 servers and assume perfectly secure encryption for free (this assumption can later be removed by using secret-sharing and by introducing one additional server). Imagine that we start out with an array of length n that is encrypted and resides on one server. The elements in the array are either real or dummy, and we would like to move all dummy elements to the end of the array while preserving the order of the real elements as they appear in the original array. For security, we would like that any single server's view in the protocol leaks no information about the array's contents.

Strawman scheme. An extremely simple strawman scheme is the following: the client makes a scan of the input array on one server; whenever it encounters a real element, it re-encrypts the element and writes it to the other server by appending it to the end of the output array (initially the output array is empty). When the entire input array has been consumed, the client pads the output array with an appropriate number of (encrypted) dummy elements.

At first sight, this algorithm seems to preserve security: each server basically observes a linear scan of either the input or the output array; and the perfectly-secure encryption hides array contents. However, upon careful examination, the second server can observe the time steps in which a write has happened to the output array — this leaks which elements are real in the original array. Correspondingly, in our formal modeling (Sect. 2), each server can not only observe each message sent and received by itself, but also the time steps in which these events occurred.

A second try. For simplicity we will describe our approach with server computation and server-to-server communication — but it is not hard to modify the scheme such that servers are completely passive. Roughly speaking, the idea is for the first server (called the *permutation server*) to randomly permute all elements and store the permuted array on the second server (called the *storage server*), such that the permutation is hidden from the storage server. Moreover, in this permuted array, we would like the elements to be tagged with pointers to form two linked lists: a real linked list and a dummy linked list. In both linked lists, the ordering of elements respects the ordering in the original array. If such a permuted array encoding two linked lists can be constructed, the client can simply traverse the real linked list first from the storage server, and then traverse the dummy linked list — writing down each element it encounters on the first server (we always assume re-encryption upon writes). Since the storage server does not know the random permutation and since every element is accessed exactly once, it observes a completely random access pattern; and thus it cannot gain any secret information.

The challenge remains as to how to tag each real (resp. dummy) element with the position of the next real (resp. dummy) element in the permuted array. This can be achieved in the following manner: the permutation server first creates a random permutation in linear time (e.g., by employing Fisher-Yates [11]), such that each element in the input array is now tagged with where it wants to be in the permuted array (henceforth called the position label). Now, the client makes a reverse scan of this input array. During this process, it remembers the position labels of the last real element seen and of the last dummy element seen so far — this takes $O(1)$ client-side storage. Whenever a real element is encountered, the client tags it with the position label of the last real seen. Similarly, whenever a dummy is encountered, the client tags it with the position label of the last dummy seen. Now, the permutation server can permute the array based on the predetermined permutation (which can also be done in linear time). At this moment, it sends the permuted, re-encrypted array to the storage server and the linked list can now be traversed from the storage server to read real elements followed by dummy elements.

It is not difficult to see that assuming that the encryption scheme is perfectly secure and every write involves re-encrypting the data, then the above scheme achieves perfect security against any single semi-honest corrupt server, and completes in linear time. Later we will replace the perfectly secure encryption with secret-sharing and this requires the introduction of one additional server.

Extending the idea for merging. We can extend the above idea to allow linear-time oblivious merging of two sorted arrays. The idea is to prepare both arrays such that they are in permuted form on the storage server and in a linked list format; and now the client can traverse the two linked lists on the storage server, merging them in the process. In each step of the merging, only one array is being consumed — since the storage server does not know the permutation, it sees random accesses and cannot tell which array is being consumed.

3-Server Perfectly Secure ORAM. We now explain the techniques for constructing a 3-server perfectly secure ORAM. A client, with $O(1)$ blocks of local cache, stores N blocks of data (secret-shared) on the 3 servers, one of which might be semi-honest corrupt. In every iteration, the client receives a memory request of the form $(\mathsf{read}, \mathsf{addr})$ or $(\mathsf{write}, \mathsf{addr}, \mathsf{data})$, and it completes this request by interacting with the servers. We would like to achieve $O(\log^2 N)$ amortized bandwidth blowup per logical memory request.

Background on single-server perfect ORAM. We start out from a state-of-the-art single-server perfectly secure scheme by Chan et al. [6] that achieves $O(\log^3 N)$ amortized bandwidth per memory request. Their scheme extends from the original hierarchical ORAM framework of Goldreich and Ostrovsky [12,13] where data blocks are stored in levels of geometrically increasing sizes. Recall that Goldreich and Ostrovsky [12,13] achieve only computational security due to the use of a PRF; and thus one of the key ideas of Chan et al. [6] is how to remove the need for a PRF. More concretely, each level in Goldreich and Ostrovsky's hierarchical ORAM is an oblivious hash table capable of supporting non-recurrent requests (henceforth called one-time memory). Within each level, the position of a data block is determined by applying a PRF to the block's logical address. To achieve perfect security, the key requirement is to eliminate the use the PRF. Therefore, in Chan et al. [6], blocks within a level are secretly and randomly permuted using an oblivious sort. To access a block within a level, the client must first figure out the block's correct location within the level. To achieve this, a trivial method is for the client to locally store the entire mapping of the correct locations (henceforth called position labels), but this would consume linear client space. Instead Chan et al. recursively store the position labels in a smaller hierarchical ORAM, inspired by a standard recursion technique commonly adopted by tree-based ORAMs [24] (but Chan et al. show how to adapt it to the hierarchical ORAM setting). Thus, in Chan et al.'s construction, there are logarithmically many hierarchical ORAMs (also called position-based ORAMs), where the ORAM at depth d (called the parent depth) stores position labels for the ORAM at depth $d+1$ (called the child depth); and finally, the ORAM at the maximum depth $D = O(\log N)$ stores the real data blocks.

Our multi-server perfect ORAM. We now explain how to build on top of Chan et al. [6]'s idea and obtain a multi-server ORAM that saves a logarithmic factor in bandwidth. The key to enabling this is a method for passing information between adjacent recursion depths, *without oblivious sort*. Below, we first explain how Chan et al. [6] passes information between adjacent recursion depths using oblivious sort, and then we explain our novel techniques to accomplish the same, but now relying only on *merging* and *compaction* in the multi-server setting.

As Chan et al. [6] point out, whenever a data block's location is updated at depth d through a shuffle operation, the position label at depth $d-1$ needs to be updated to reflect the new location. This information passing between an ORAM at depth d to its parent ORAM at depth $d-1$ is performed by using a coordinated shuffle between the logarithmically many ORAMs upon every memory request. This turns out to be the most intricate part of their scheme. During this shuffle, suppose that the parent and the child each has an array of logical addresses and a position label for each address. It is guaranteed by the ORAM construction that all addresses the child has must appear in the parent's array. Moreover, if some address appears in both the parent and child, then the child's version is fresher. We would like to combine the information held by the parent and the child by retaining the freshest copy of position label for every address. Chan et al. relied on oblivious sorting to achieve this goal: if some address is held by both the parent and child, they will appear adjacent to each other in the sorted array; and thus in a single linear scan one can easily cross out all stale copies.

To save a logarithmic factor, we must solve the above problem using only merging and compaction and not sorting. Notice that if both the parent's and the child's arrays are already sorted according to the addresses, then the aforementioned information propagation from child to parent can be accomplished through merging rather than sorting (in the full scheme we would also need stable compaction to remove dummy blocks in a timely fashion to avoid blowup of array sizes over time). But how can we make sure that these arrays are sorted in the first place without oblivious sorting? In particular, these arrays actually correspond to levels in a hierarchical ORAM in Chan et al. [6]'s scheme, and all blocks in a level must appear in randomly permuted order to allow safe (one-time) accesses — this seems to contradict our desire for sortedness. Fortunately, here we can rely again on the permutation-storage-separation paradigm — for simplicity again we describe our approach for 2 servers assuming perfectly secure (re-)encryption upon every write. The idea is the following: although the storage server is holding each array (i.e., level) in a randomly permuted order, the permutation server will remember an inverse permutation such that when this permutation is applied to the storage server's copy, sortedness is restored. Thus whenever shuffling is needed, the permutation server would first apply the inverse permutation to the storage server's copy to restore sortedness, and then we could rely on merging (and compaction) to propagate information between adjacent depths rather than sorting.

Outline. In Sect. 3, we explain our protocol for permuting and unpermuting a list of blocks under the permutation-storage-separation paradigm and build upon it to describe a protocol for oblivious stable compaction and merge. In Sect. 4, we show the protocol for a three-server oblivious one-time memory; this corresponds to a single level in position-based ORAM in Chan et al. [6]. In Sect. 5, we first show how a three-server position-based ORAM can be built using the one-time memory (Sect. 5.1), and then construct our final ORAM scheme consisting of logarithmic number of position-based ORAMs (Sect. 5.2).

1.2 Related Work

The notion of Oblivious RAM (ORAM) was introduced by the seminal work of Goldreich and Ostrovsky around three decades ago [12,13]. Their construction used a hierarchy of buffers of exponentially increasing size, which was later known as the hierarchical ORAM framework. Their construction achieved an amortized bandwidth blowup of $O(\log^3 N)$ and was secure against a computationally bounded adversary. Subsequently, several works have improved the bandwidth blowup from $O(\log^3 N)$ to $O(\log^2 N/\log\log N)$ [3,7,15,18] under the same adversarial model. Ajtai [2] was the first to consider the notion of a statistically secure oblivious RAM that achieves $O(\log^3 N)$ bandwidth blowup. This was followed by the statistically secure ORAM construction by Shi et al. [24], who introduced the tree-based paradigm. ORAM constructions in the tree-based paradigm have improved the bandwidth blowup from $O(\log^3 N)$ to $O(\log^2 N)$ [8,23,24,26,28]. Though the computational assumptions have been removed, the statistically secure ORAMs still fail with a failure probability that is negligibly small in the number of data blocks stored in the ORAM.

Perfectly secure ORAMs. Perfectly secure ORAM was first studied by Damgård et al. [9]. Perfect security requires that a computationally unbounded server does not learn anything other than the number of requests with probability 1. This implies that the oblivious program's memory access patterns should be *identically distributed* regardless of the inputs to the program; and thus with probability 1, no information can be leaked about the secret inputs to the program. Damgård et al. [9] achieve an *expected* $O(\log^3 N)$ simulation overhead and $O(\log N)$ space blowup relative to the original RAM program. Raskin et al. [22] and Demertzis et al. [10] achieve a *worst-case* bandwidth blowup of $O(\sqrt{N}\frac{\log N}{\log\log N})$ and $O(N^{1/3})$, respectively. Chan et al. [6] improve upon Damgård et al.'s result [9] by avoiding the $O(\log N)$ blowup in space, and by showing a construction that is conceptually simpler. Our construction builds upon Chan et al. and improves the bandwidth blowup to *worst-case* $O(\log^2 N)$ while assuming three non-colluding servers.

We note that since both Damgård et al. [9] and Chan et al. [6] employ perfectly oblivious random permutations, their schemes are Las Vegas algorithms and there is a negligibly small failure probability that the algorithm exceeds the stated runtime (however, perfect security is maintained nonetheless). Our multi-server ORAM avoids the need for oblivious random permutation and thus the algorithm's runtime is deterministic.

Multi-server ORAMs. ORAMs in this category assume multiple non-colluding servers to improve bandwidth blowup [1,16,17,19,21]. A comparison of the relevant schemes is presented in Table 1. Among these, the work that is closely related to ours is by Lu and Ostrovsky [21] which achieves a bandwidth blowup of $O(\log N)$ assuming two non-colluding servers. In their scheme, each server performs permutations for data that is stored by the other server. While their construction is computationally secure, we achieve perfect security for access patterns as well as the data itself. Moreover, our techniques can be used to perform an oblivious tight stable compaction and an oblivious merge operation in linear time; how to perform these operations in linear time were not known even for the computationally secure setting. On the other hand, our scheme achieves an $O(\log^2 N)$ bandwidth blowup and uses three servers. We remark that if we assume a perfectly secure encryption scheme, our construction can achieve perfectly secure access patterns using two servers. Abraham et al. [1], Gordon et al. [16] and Kushilevitz and Mour [19] construct multi-server ORAMs using PIR. Each of these constructions require the server to perform computation for using PIR operations. While Abraham et al. [1] achieve statistical security for access patterns, other work [16,19] is only computationally secure. While the work of Gordon et al. achieves a bandwidth blowup of $O(\log N)$, they require linear-time server computation. Abraham et al. and Kushilevitz and Mour, on the other hand, are poly-logarithmic and logarithmic respectively, both in computation and bandwidth blowup. In comparison, our construction achieves perfect security and requires a passive server (i.e., a server that does not perform any computation) at a bandwidth blowup of $O(\log^2 N)$.

Table 1. Comparison with existing multi-server Oblivious RAM schemes for block size $\Omega(\log N)$. All of the other schemes (including the statistically-secure schemes [1]) require two servers but assume the existence of an unconditionally secure encryption scheme. With a similar assumption, our work would indeed need only two servers too.

Construction	Bandwidth Blowup	Server Computation	Security
Lu-Ostrovsky [21]	$O(\log N)$	-	Computational
Gordon et al. [16]	$O(\log N)$	$O(N)$	Computational
Kushilevitz et al. [19]	$O(\log N \cdot \omega(1))$	$O(\log N \cdot \omega(1))$	Computational
Abraham et al. [1]	$O(\log^2 N \cdot \omega(1))$	$O(\log^2 N \cdot \omega(1))$	Statistical
Our work	$O(\log^2 N)$	-	Perfect

2 Definitions

In this section, we revisit how to define multi-server ORAM schemes for the case of semi-honest corruptions. Our definitions require that the adversary, controlling a subset of semi-honest corrupt servers, learns no secret information during the execution of the ORAM protocol. Specifically our adversary can observe all messages transmitted to and from corrupt servers, the rounds in which they were transmitted, as well as communication patterns between honest parties (including the client and honest servers). Our definition generalizes existing works [1] where they assume free encryption of data contents (even when statistical security is desired).

2.1 Execution Model

Protocol as a system of Interactive RAMs. We consider a protocol between multiple parties including a client, henceforth denoted by $\mathbf{C}$, and k servers, denoted by $\mathbf{S}_0, \ldots, \mathbf{S}_{k-1}$, respectively. The client and all servers are Random Access Machines (RAMs) that interact with each other. Specifically, the client or each server has a CPU capable of computation and a memory that supports reads and writes; the CPU interacts with the memory to perform computation. The atomic unit of operation for memory is called a *block*. We assume that all RAMs can be *probabilistic*, i.e., they can read a random tape supplying a stream of random bits.

Communication and timing. We assume pairwise channels between all parties. There are two notions of time in our execution model, CPU cycles and communication rounds. Without loss of generality, henceforth we assume that it takes the same amount of time to compute each CPU instruction and to transmit each memory block over the network to another party (since we can always take the maximum of the two). Henceforth in this paper we often use the word *round* to denote the time that has elapsed since the beginning of the protocol.

Although we define RAMs on the servers as being capable of performing any arbitrary computation, all of our protocols require the servers to be passive, i.e., the server RAMs only perform read/write operations from the memory stored by it.

2.2 Perfect Security Under a Semi-Honest Adversary

We consider the client to be *trusted*. The adversary can corrupt a subset of the servers (but it cannot corrupt the client) — although our constructions are secure against any individual corrupt server, we present definitions for the more general case, i.e., when the adversary can control more than one corrupt server.

We consider a *semi-honest* adversary, i.e., the corrupt servers still honestly follow the protocol; however, we would like to ensure that no undesired information will leak. To formally define security, we need to first define what the adversary can observe in a protocol's execution.

View of adversary $\mathsf{view}^{\mathcal{A}}$. Suppose that the adversary $\mathcal{A}$ controls a subset of the servers — we abuse notation and use $\mathcal{A} \subset [k]$ to denote the set of corrupt servers. The view of the adversary, denoted by $\mathsf{view}^{\mathcal{A}}$ in a random run of the protocol consists of the following:

1. *Corrupt parties' views:* These views include (1) corrupt parties' inputs, (2) all randomness consumed by corrupt parties, and (3) an ordered sequence of all messages received by corrupt parties, including which party the message is received from, as well as the *round* in which each message is received. We assume that these messages are ordered by the round in which they are received, and then by the party from which it is received.
2. *Honest communication pattern*: when honest parties (including the client) exchange messages, the adversary observes their communication pattern, including which pairs of honest nodes exchange messages in which round.

We stress that in our model only one block can be exchanged between every pair in a round — thus the above $\mathsf{view}^{\mathcal{A}}$ definition effectively allows $\mathcal{A}$ to see the total length of messages exchanged between honest parties.

Remark 1. We remark that this definition captures a notion of timing patterns along with access patterns. For instance, suppose two servers store two sorted lists that needs to be merged. The client performs a regular merge operation to read from the two lists, reading the heads of the lists in each round. In such a scenario, depending on the rounds in which blocks are read from a server, an adversary that corrupts that server can compute the relative ordering of blocks between the two lists.

Defining security in the ideal-real paradigm. Consider an ideal functionality $\mathcal{F}$: upon receiving the input $\mathbf{I}_0$ from the client and inputs $\mathbf{I}_1, \ldots, \mathbf{I}_k$ from each of the k servers, respectively, and a random string ρ sampled from some distribution, $\mathcal{F}$ computes

$$(\mathbf{O}_0, \mathbf{O}_1, \ldots, \mathbf{O}_k) := \mathcal{F}(\mathbf{I}_0, \mathbf{I}_1, \ldots, \mathbf{I}_k; \rho)$$

where $\mathbf{O}_0$ is the client's output, and $\mathbf{O}_1, \ldots, \mathbf{O}_k$ denote the k servers' outputs, respectively.

Definition 1 (Perfect security in the presence of a semi-honest adversary). *We say that "a protocol* Π *perfectly securely realizes an ideal functionality* $\mathcal{F}$ *in the presence of a semi-honest adversary corrupting* t *servers" if and only if for every adversary* $\mathcal{A}$ *that controls up to* t *corrupt servers, there exists a simulator* Sim *such that for every input vector* $(\mathbf{I}_0, \mathbf{I}_1, \ldots, \mathbf{I}_k)$*, the following real- and ideal-world experiments output identical distributions:*

- *Ideal-world experiment. Sample* ρ *at random and compute* $(\mathbf{O}_0, \mathbf{O}_1, \ldots, \mathbf{O}_k) := \mathcal{F}(\mathbf{I}_0, \mathbf{I}_1, \ldots, \mathbf{I}_k, \rho)$*. Output the following tuple where we abuse notation and use* $i \in \mathcal{A}$ *to denote the fact that* i *is corrupt:*
$$\mathsf{Sim}(\{\mathbf{I}_i, \mathbf{O}_i\}_{i \in \mathcal{A}}), \quad \mathbf{O}_0, \{\mathbf{O}_i\}_{i \notin \mathcal{A}}$$
- *Real-world experiment. Execute the (possibly randomized) real-world protocol, and let* $\mathbf{O}_0, \mathbf{O}_1, \ldots, \mathbf{O}_k$ *be the outcome of the client and each of the* k *servers, respectively. Let* $\mathsf{view}^{\mathcal{A}}$ *denote the view of the adversary* $\mathcal{A}$ *in this run. Now, output the following:*
$$\mathsf{view}^{\mathcal{A}}, \; \mathbf{O}_0, \{\mathbf{O}_i\}_{i \notin \mathcal{A}}$$

Note that throughout the paper, we will define various building blocks that realize different ideal functionalities. The security of all building blocks can be defined in a unified approach with this paradigm. When we compose these building blocks to construct our full protocol, we can prove perfect security of the full protocol in a composable manner. By modularly proving the security of each building block, we can now think of each building block as interacting with an ideal functionality. This enables us to prove the security of the full protocol in the ideal world assuming the existence of these ideal functionalities.

We note that while the definitions in this paper apply to both active-server protocols (where the server can perform arbitrary computation) as well as passive server protocols (where the server performs no computation), our scheme does not require server computation.

2.3 Definition of k-Server Oblivious RAM

Ideal logical memory. The ideal logical memory is defined in the most natural way. There is a memory array consisting of N blocks where each block is $\Omega(\log N)$ bits long, and each block is identified by its unique *address* which takes value in the range $\{0, 1, \ldots, N-1\}$.

Initially all blocks are set to 0. Upon receiving $(\mathtt{read}, \mathsf{addr})$, the value of the block residing at address addr is returned. Upon receiving $(\mathtt{write}, \mathsf{addr}, \mathsf{data})$, the block at address addr is overwritten with the data value data, and its old value (before being rewritten) is returned.

k-server ORAM. A k-server Oblivious RAM (ORAM) is a protocol between a client $\mathbf{C}$ and k servers $\mathbf{S}_1, \ldots, \mathbf{S}_k$ which realizes an ideal logical memory. The execution of this protocol proceeds in a sequence of iterations: in each interaction, the client $\mathbf{C}$ receives a logical memory request of the form $(\mathtt{read}, \mathsf{addr})$ or $(\mathtt{write}, \mathsf{addr}, \mathsf{data})$. It then engages in some (possibly randomized) protocol with the servers, at the end of which it produces some output thus completing the current iteration.

We require perfect correctness and perfect security as defined below. We refer to a sequence of logical memory requests as a *request sequence* for short.

- *Perfect correctness.* For any request sequence, with probability 1, all of the client's outputs must be correct. In other words, we require that with probability 1, all of the client's outputs must match what an ideal logical memory would have output for the same request sequence.
- *Perfect security under a semi-honest adversary.* We say that a k-server ORAM scheme satisfies perfect security w.r.t. a semi-honest adversary corrupting t servers, if and only if for every $\mathcal{A}$ that controls up to t servers, and for every two request sequences $\mathbf{R}_0$ and $\mathbf{R}_1$ of equal length, the views $\mathsf{view}^{\mathcal{A}}(\mathbf{R}_0)$ and $\mathsf{view}^{\mathcal{A}}(\mathbf{R}_1)$ are identically distributed, where $\mathsf{view}^{\mathcal{A}}(\mathbf{R})$ denotes the view of $\mathcal{A}$ (as defined earlier in Sect. 2.2) under the request sequence $\mathbf{R}$.

Since we require perfect security (and is based on information-theoretic secret-sharing), our notion resists adaptive corruptions and is composable.

2.4 Resource Assumptions and Cost Metrics

We assume that the client can store $O(1)$ blocks while the servers can store $O(N)$ blocks. We will use the metric *bandwidth blowup* to characterize the performance of our protocols. Bandwidth blowup is the (amortized) number of blocks queried in the ORAM simulation to query a single virtual block. We also note that since the servers do not perform any computation, and the client always performs an $O(1)$ computation on its $O(1)$ storage, an $O(X)$ bandwidth blowup also corresponds to an $O(X)$ *runtime* for our protocol.

3 Core Building Blocks: Definitions and Constructions

Imagine that there are three servers denoted $\mathbf{S}_0$, $\mathbf{S}_1$, and $\mathbf{S}_2$, and a client denoted $\mathbf{C}$. We use $\mathbf{S}_b, b \in \mathbb{Z}_3$ to refer to a specific server. Arithmetic performed on the subscript b is done modulo 3.

3.1 Useful Definitions

Let T denote a list of blocks where each block is either a real block containing a payload string and a logical address; or a dummy block denoted $\perp$. We define *sorted* and *semi-sorted* as follows:

– *Sorted:* T is said to be sorted *iff* all real blocks appear before dummy ones; and all the real blocks appear in increasing order of their logical addresses. If multiple blocks have the same logical address, their relative order can be arbitrary.
– *Semi-sorted:* T is said to be semi-sorted *iff* all the real blocks appear in increasing order of their logical addresses, and ties may be broken arbitrarily. However, the real blocks are allowed to be interspersed by dummy blocks.

Array Notation. We assume each location of an array T stores a *block* which is a bit-string of length B. Given two arrays T_1 and T_2, we use $\mathsf{T}_1 \oplus \mathsf{T}_2$ to denote the resulting array after performing bitwise-XOR on the corresponding elements at each index of the two arrays; if the two arrays are of different lengths, we assume the shorter array is appended with a sufficient number of zero elements.

Permutation Notation. When a permutation $\pi : [n] \to [n]$ is applied to an array T indexed by $[n]$ to produce $\pi(\mathsf{T})$, we mean the element currently at location i will be moved to location $\pi(i)$. When we compose permutations, $\pi \circ \sigma$ means that π is applied *before* σ. We use $\mathfrak{e}$ to denote the identity permutation.

Layout. A layout is a way to store some data T on three servers such that the data can be recovered by combining information on the three servers. Recall that the client has only $O(1)$ blocks of space, and our protocol does not require that the client stores any persistent data.

Whenever some data T is stored on a server, informally speaking, we need to ensure two things: (1) The server does not learn the data T itself, and (2) The server does not learn *which* index i of the data is accessed. In order to ensure the prior, we XOR secret-share the data $\mathsf{T} := \mathsf{T}_0 \oplus \mathsf{T}_1 \oplus \mathsf{T}_2$ between three servers $\mathbf{S}_b, b \in \mathbb{Z}_3$ such that $\mathbf{S}_b$ stores T_b. For a server to not learn *which* index i in T is accessed, we ensure that the data is permuted, and the access happens to the permuted data. If the data is accessed on the same server that permutes the data, then the index i will still be revealed. Thus, for each share T_b, we ensure that one server permutes it and we access it from another server, i.e., we have two types of servers:

- Each server $\mathbf{S}_b$ acts as a *storage server* for the b-th share, and thus it knows T_b.
- Each server $\mathbf{S}_b$ also acts as the *permutation server* for the $(b+1)$-th share, and thus it also knows T_{b+1} as well as π_{b+1}.

Throughout the paper, a layout is of the following form

$$\text{3-server layout : } \quad \{\pi_b, \mathsf{T}_b\}_{b \in \mathbb{Z}_3}$$

where T_b and $(\pi_{b+1}, \mathsf{T}_{b+1})$ are stored by server $\mathbf{S}_b$. As mentioned, $\mathbf{S}_b$ not only knows its own share (T_b) but also the permutation and share of the next server $(\pi_{b+1}, \mathsf{T}_{b+1})$.

Specifically, $\mathsf{T}_0, \mathsf{T}_1, \mathsf{T}_2$ denote lists of blocks of equal length: we denote $n = |\mathsf{T}_0| = |\mathsf{T}_1| = |\mathsf{T}_2|$. Further, $\pi_{b+1} : [n] \to [n]$ is a permutation stored by server $\mathbf{S}_b$ for the list T_{b+1}. Unless there is ambiguity, we use $\oplus_b$ to mean applying $\oplus_{b \in \mathbb{Z}_3}$ to three underlying arrays.

The above layout is supposed to store the array that can be recovered by:

$$\oplus_b \pi_b^{-1}(\mathsf{T}_b).$$

Henceforth, given a layout $\{\pi_b, \mathsf{T}_b\}_{b \in \mathbb{Z}_3}$, we say that the layout is *sorted* (or semi-sorted) iff $\oplus_b \pi_b^{-1}(\mathsf{T}_b)$ is sorted (or semi-sorted).

Special Case. Sometimes the blocks secret-shared among $\mathbf{S}_0$, $\mathbf{S}_1$, $\mathbf{S}_2$ may be unpermuted, i.e., for each $b \in \mathbb{Z}_3$, π_b is the identity permutation $\mathfrak{e}$. In this case, the layout is

$$\text{Unpermuted layout : } \quad \{\mathfrak{e}, \mathsf{T}_b\}_{b \in \mathbb{Z}_3}$$

For brevity, the unpermuted layout $\{\mathfrak{e}, \mathsf{T}_b\}_{b \in \mathbb{Z}_3}$ is also denoted by the abstract array T.

Definition 2 (Secret Write). *An* abstract *array* T *corresponds to some unpermuted layout* $\{\mathfrak{e}, \mathsf{T}_b\}_{b \in \mathbb{Z}_3}$. *We say that the client* secretly writes *a value* B *to the array* T *at index* i*, when it does the following:*

- *Sample random values* B_0 *and* B_1 *independently, and compute* $\mathsf{B}_2 := \mathsf{B} \oplus \mathsf{B}_0 \oplus \mathsf{B}_1$.
- *For each* $b \in \mathbb{Z}_3$*, the client writes* $\mathsf{T}_b[i] := \mathsf{B}_b$ *on server* $\mathbf{S}_b$ *(and* $\mathbf{S}_{b-1}$*).*

Definition 3 (Reconstruct). *Given some layout* $\{\pi_b, \mathsf{T}_b\}_{b \in \mathbb{Z}_3}$*, the client* reconstructs *a value from using tuple* (i_0, i_1, i_2) *of indices, when it does the following:*

- *For each* $b \in \mathbb{Z}_3$*, the client reads* $\mathsf{T}_b[i_b]$ *from server* $\mathbf{S}_b$*. (It is important that the client reads* T_b *from* $\mathbf{S}_b$*, even though* T_b *is stored in both* S_b *and* S_{b-1}*.)*
- *The reconstructed value is* $\oplus_b \mathsf{T}_b[i_b]$.

Protocol Notation. All protocols are denoted as $\mathsf{out} \leftarrow \mathsf{Prot}(\mathsf{sin}, \mathsf{cin})$. Here, sin and cin are respectively server and client inputs to the protocol Prot. Except for in an ORAM Lookup, all the outputs out are sent to the server.

3.2 Permute and Unpermute

Non-oblivious random permutation. Fisher and Yates [11] show how to generate a uniformly random permutation $\pi : [n] \to [n]$ in $O(n)$ time steps. This implies that the client can write a random permutation on a server with $O(n)$ bandwidth. The permutation is non-oblivious, i.e., the server *does* learn the permutation generated.

Definition of Permute. Permute is a protocol that realizes an ideal functionality $\mathcal{F}_{\text{perm}}$ as defined below. Intuitively, this functionality takes some unpermuted input layout (i.e., unpermuted secret-shared inputs) and three additional permutations π_{b+1} from the three permutation servers $\mathbf{S}_b$. The functionality produces an output such that the three shares are secret-shared again, and the share received by storage server $\mathbf{S}_{b+1}$ is permuted using π_{b+1}. Secret-sharing the data again before applying the new permutations ensures that a storage server $\mathbf{S}_{b+1}$ does not learn the permutation π_{b+1} applied to its share.

– $\{\pi_b, \mathsf{T}'_b\}_{b\in\mathbb{Z}_3} \leftarrow \mathsf{Permute}\big((\{\mathfrak{e}, \mathsf{T_b}\}_{\mathsf{b}\in\mathbb{Z}_3}, \{\pi_{\mathsf{b}}\}_{\mathsf{b}\in\mathbb{Z}_3}), \perp\big)$:
  - *Input*: Let $\{\mathfrak{e}, \mathsf{T}_b\}_{b\in\mathbb{Z}_3}$ be the unpermuted layout provided as input. (Recall that T_b and T_{b+1} are stored in server $\mathbf{S}_b$.)
    Moreover, for each $b \in \mathbb{Z}_3$, $\mathbf{S}_b$ has an additional permutation π_{b+1} as input (which could be generated by the client for instance).
    The arrays have the same length $|\mathsf{T}_0| = |\mathsf{T}_1| = |\mathsf{T}_2| = n$, for some n. The client obtains $\perp$ as the input.
  - *Ideal functionality* $\mathcal{F}_{\text{perm}}$:
    Sample independently and uniformly random $\widehat{\mathsf{T}}_0, \widehat{\mathsf{T}}_1$ of length n.
    Now, define $\widehat{\mathsf{T}}_2 := \widehat{\mathsf{T}}_0 \oplus \widehat{\mathsf{T}}_1 \oplus (\oplus_b \mathsf{T}_b)$, i.e., $\oplus_b \widehat{\mathsf{T}}_b = \oplus_b \mathsf{T}_b$.
    For each $b \in \mathbb{Z}_3$, define $\mathsf{T}'_b := \pi_b(\widehat{\mathsf{T}}_b)$.
    The output layout is $\{\pi_b, \mathsf{T}'_b\}_{b\in\mathbb{Z}_3}$, and the client's output is $\perp$.

Protocol Permute. The implementation of $\mathcal{F}_{\text{perm}}$ proceeds as follows:

1. *Mask shares.* For each data block, the client first generates block "masks" that sum up to *zero*, and then applies mask to T_{b+1} on server $\mathbf{S}_b$. Specifically, the client does the following, for each $i \in [n]$:
   – Generate block "masks" that sum up to *zero*, i.e., sample independent random blocks B_0^i and B_1^i, and compute $\mathsf{B}_2^i := \mathsf{B}_0^i \oplus \mathsf{B}_1^i$.
   – Apply mask B_{b+1}^i to $\mathsf{T}_{b+1}[i]$ stored on server $\mathbf{S}_b$, i.e., for each $i \in [b]$, the client writes $\widehat{\mathsf{T}}_{b+1}[i] \leftarrow \mathsf{T}_{b+1}[i] \oplus \mathsf{B}_{b+1}^i$ on server $\mathbf{S}_b$.
2. *Permute share of* $\mathbf{S}_{b+1}$ *and send result to* $\mathbf{S}_{b+1}$. The client uses π_{b+1} to permute a share on the permutation server and then sends this permuted share to the storage server, i.e., for each $b \in \mathbb{Z}_3$, the client computes computes $\mathsf{T}'_{b+1} := \pi_{b+1}(\widehat{\mathsf{T}}_{b+1})$ on server $\mathbf{S}_b$, and sends the result T'_{b+1} to $\mathbf{S}_{b+1}$. Each server $\mathbf{S}_b$ stores T'_b and $(\pi_{b+1}, \mathsf{T}'_{b+1})$; hence, the new layout $\{\pi_b, \mathsf{T}'_b\}_{b\in\mathbb{Z}_3}$ is achieved.

Theorem 3. *The* $\mathsf{Permute}$ *protocol perfectly securely realizes the ideal functionality* $\mathcal{F}_{\text{perm}}$ *(as per Definition 1) in the presence of a semi-honest adversary corrupting a single server with* $O(n)$ *bandwidth.*

Due to lack of space, the proof is in the full version of the paper [5].

Definition of $\mathsf{Unpermute}$ and Protocol Description. Similar to $\mathsf{Permute}$, we also need a complementary $\mathsf{Unpermute}$ protocol. Its definition and protocol are described in the full version [5].

3.3 Stable Compaction

Definition of $\mathsf{StableCompact}$. $\mathsf{StableCompact}$ is a protocol that realizes an ideal functionality $\mathcal{F}_{\text{compact}}$, as defined below:

- $\{\mathfrak{e}, \mathsf{T}'_b\}_{b\in\mathbb{Z}_3} \leftarrow \mathsf{StableCompact}(\{\mathfrak{e}, \mathsf{T}_b\}_{b\in\mathbb{Z}_3}, \perp)$:
  - *Input layout*: A *semi-sorted*, unpermuted layout denoted $\{\mathfrak{e}, \mathsf{T}_b\}_{b\in\mathbb{Z}_3}$.
  - *Ideal functionality* $\mathcal{F}_{\text{compact}}$: $\mathcal{F}_{\text{compact}}$ computes $\mathsf{T}^* := \mathsf{T}_0 \oplus \mathsf{T}_1 \oplus \mathsf{T}_2$; it then moves all dummy blocks in T^* to the end of the array, while keeping the relative order of real blocks unchanged.
    Now, $\mathcal{F}_{\text{compact}}$ randomly samples $\mathsf{T}'_0, \mathsf{T}'_1$ of appropriate length and computes T'_2 such that $\mathsf{T}^* = \mathsf{T}'_0 \oplus \mathsf{T}'_1 \oplus \mathsf{T}'_2$. The output layout is a *sorted*, unpermuted layout $\{\mathfrak{e}, \mathsf{T}'_b\}_{b\in\mathbb{Z}_3}$.

$\mathsf{StableCompact}$ Protocol. The input is a *semi-sorted*, unpermuted layout, and we would like to turn it into a *sorted*, unpermuted layout obliviously. The key idea is to permute each share of the list (stored on the 3 servers respectively), such that the storage server for each share does not know the permutation. Now, the client accesses all real elements in a sorted order, and then accesses all dummy elements, writing down the elements in a secret-shared manner as the accesses are made. We can achieve this if each real or dummy element is tagged with a pointer to its next element, and the pointer is in fact a 3-tuple that is also secret-shared on the 3 servers — each element in the 3-tuple indicates where the next element is in one of the 3 permutations.

Therefore, the crux of the algorithm is to tag each (secret-shared) element with a (secret-shared) position tuple, indicating where its next element is — this will effectively create two linked list structures (one for real and one for dummy): each element in the linked lists is secret-shared in to 3 shares, and each share resides on its storage server at an independent random location.

The detailed protocol is as follows:

1. First, each server $\mathbf{S}_b$ acts as the permutation server for $\mathbf{S}_{b+1}$. Thus, the client generates a random permutation π_{b+1} on the permutation server $\mathbf{S}_b$ using the Fisher-Yates algorithm described in Sect. 3.2. Basically, for each index i of the original list the client writes down, on each $\mathbf{S}_b$, that its $(b+1)$-th share (out of 3 shares), wants to be in position $\pi_{b+1}(i)$.

2. Next, the client makes a reverse scan of $(\mathsf{T}_0, \pi_0), (\mathsf{T}_1, \pi_1), (\mathsf{T}_2, \pi_2)$ for $i = n$ down to 1. The client can access $(\mathsf{T}_{b+1}[i], \pi_{b+1}(i))$ by talking to $\mathbf{S}_b$. In this reverse scan, the client always locally remembers the position tuple of the last real element encountered (henceforth denoted $\mathfrak{p}_{\text{real}}$) and the position tuple of the last dummy element encountered (henceforth denoted $\mathfrak{p}_{\text{dummy}}$). Thus, if $\mathsf{T}[k_{\text{real}}]$ is the last seen real element, then the client remembers $\mathfrak{p}_{\text{real}} = (\pi_b(k_{\text{real}}) : b \in \mathbb{Z}_3)$. $\mathfrak{p}_{\text{dummy}}$ is updated analogously. Initially, $\mathfrak{p}_{\text{real}}$ and $\mathfrak{p}_{\text{dummy}}$ are set to $\perp$.
   During this scan, whenever a real element $\mathsf{T}[i]$ is encountered, the client secretly writes the link $\mathsf{L}[i] := \mathfrak{p}_{\text{real}}$, i.e., $\mathsf{L}[i]$ represents secret-shares of the next pointers for the real element and L itself represents an abstract linked list of real elements. The links for dummy elements are updated analogously using $\mathfrak{p}_{\text{dummy}}$.
   At the end of this reverse scan, the client remembers the position tuple for the first real of the linked list denoted $\mathfrak{p}^1_{\text{real}}$ and position tuple for the first dummy denoted $\mathfrak{p}^1_{\text{dummy}}$.
3. Next, we call Permute inputting (1) the original layout — but importantly, now each element is tagged with a position tuple (that is also secret-shared); and (2) the three permutations chosen by each $\mathbf{S}_b$ (acting as the permutation server for $\mathbf{S}_{b+1}$). Thus, Permute is applied to the combined layout $\{\mathfrak{e}, (\mathsf{T}_b, \mathsf{L}_b)\}_{b \in \mathbb{Z}_3}$, where $\mathbf{S}_b$ has input permutation π_{b+1}. Let the output of Permute be denoted by $\{\pi_b, (\mathsf{T}'_b, \mathsf{L}'_b)\}_{b \in \mathbb{Z}_3}$.
4. Finally, the client traverses first the real linked list (whose start position tuple is $\mathfrak{p}^1_{\text{real}}$) and then the dummy linked list (whose start position tuple is $\mathfrak{p}^1_{\text{dummy}}$). During this traversal, the client secretly writes each element encountered to produce the sorted and unpermuted output layout.
   More precisely, the client secretly writes an abstract array T'' element by element. Start with $k \leftarrow 0$ and $\mathfrak{p} \leftarrow \mathfrak{p}^1_{\text{real}}$.
   The client reconstructs element $\mathsf{B} := \oplus \mathsf{T}'_b[\mathfrak{p}_b]$ and the next pointer of the linked list $\mathsf{next} := \oplus \mathsf{L}'_b[\mathfrak{p}_b]$; the client secretly writes to the abstract array $\mathsf{T}''[k] := \mathsf{B}$.
   Then, it updates $k \leftarrow k+1$ and $\mathfrak{p} \leftarrow \mathsf{next}$, and continues to the next element; if the end of the real list is reached, then it sets $\mathfrak{p} \leftarrow \mathfrak{p}^1_{\text{dummy}}$. This continues until the whole (abstract) T'' is secretly written to the three servers.
5. The new layout $\{\mathfrak{e}, \mathsf{T}''_b\}_{b \in \mathbb{Z}_3}$ is constructed.

Theorem 4. *The* StableCompact *protocol perfectly securely realizes the ideal functionality $\mathcal{F}_{\text{compact}}$ (as per Definition 1) in the presence of a semi-honest adversary corrupting a single server with $O(n)$ bandwidth.*

Due to lack of space, the proof is in the full version of the paper [5].

3.4 Merging

Definition of Merge. Merge is a protocol that realizes an ideal functionality $\mathcal{F}_{\text{merge}}$ as defined below:

– $\{\mathfrak{e}, \mathsf{U}''_b\}_{b\in\mathbb{Z}_3} \leftarrow \mathsf{Merge}\big(\{\mathfrak{e}, (\mathsf{T_b}, \mathsf{T'_b})\}_{\mathsf{b}\in\mathbb{Z}_3}, \perp\big)$:
  - *Input layout*: Two *semi-sorted*, unpermuted layouts denoted $\{\mathfrak{e}, \mathsf{T}_b\}_{b\in\mathbb{Z}_3}$ and $\{\mathfrak{e}, \mathsf{T}'_b\}_{b\in\mathbb{Z}_3}$ denoting abstract lists T and T', where all the arrays have the same length n.
  - *Ideal functionality* $\mathcal{F}_{\text{merge}}$: First, $\mathcal{F}_{\text{merge}}$ merges the two lists $\mathsf{T}_0 \oplus \mathsf{T}_1 \oplus \mathsf{T}_2$ and $\mathsf{T}'_0 \oplus \mathsf{T}'_1 \oplus \mathsf{T}'_2$, such that the resulting array is sorted with all dummy blocks at the end. Let U'' be this merged result. Now, $\mathcal{F}_{\text{merge}}$ randomly samples U''_0 and U''_1 independently of appropriate length and computes U''_2 such that $\mathsf{U}'' = \mathsf{U}''_0 \oplus \mathsf{U}''_1 \oplus \mathsf{U}''_2$. The output layout is a *sorted*, unpermuted layout $\{\mathfrak{e}, \mathsf{U}''_b\}_{b\in\mathbb{Z}_3}$.

Merge Protocol. The protocol receives as input, two semi-sorted, unpermuted layouts and produces a merged, sorted, unpermuted layout as the output. The key idea is to permute the concatenation of the two semi-sorted inputs such that the storage servers do not know the permutation. Now, the client accesses real elements in both lists in the sorted order using the storage servers to produce a merged output. Given that a *concatenation of the lists* is permuted together, elements from *which* list is accessed is not revealed during the merge operation, thereby allowing us to merge the two lists obliviously. In order to access the two lists in a sorted order, the client creates a linked list of real and dummy elements using the permutation servers, similar to the StableCompact protocol in Sect. 3.3.

The detailed protocol works as follows:

1. First, the client concatenates the two abstract lists T and T' to obtain an abstract list U of size $2n$, i.e., we interpret U_b as the concatenation of T_b and T'_b for each $b \in \mathbb{Z}_3$. Specifically, $\mathsf{U}_b[0, n-1]$ corresponds to T_b and $\mathsf{U}_b[n, 2n-1]$ corresponds to T'_b.
2. Now, each server $\mathbf{S}_b$ acts as the permutation server for $\mathbf{S}_{b+1}$. The client generates a random permutation $\pi_{b+1} : [2n] \to [2n]$ on server $\mathbf{S}_{b+1}$ using the Fisher-Yates algorithm described in Sect. 3.2. $\pi_{b+1}(i)$ represents the position of the $(b+1)$-th share and is stored on server $\mathbf{S}_b$.
3. The client now performs a reverse scan of $(\mathsf{U}_0, \pi_0), (\mathsf{U}_1, \pi_1), (\mathsf{U}_2, \pi_2)$ for $i = n$ down to 1. During this reverse scan, the client always locally remembers the position tuples of the last real element and last dummy element encountered for both the lists. Let them be denoted by $\mathfrak{p}_{\text{real}}$, $\mathfrak{p}'_{\text{real}}$, $\mathfrak{p}_{\text{dummy}}$, and $\mathfrak{p}'_{\text{dummy}}$. Thus, if $\mathsf{U}[k_{\text{real}}]$ is the last seen real element from the first list, the client remembers $\mathfrak{p}_{\text{real}} = (\pi_b(k_{\text{real}}) : b \in \mathbb{Z}_3)$. The other position tuples are updated analogously. Each of these tuples are initially set to $\perp$.
   During the reverse scan, the client maintains an abstract linked list L in the following manner. When $\mathsf{U}[i]$ is processed, if it is a real element from the first list, then the client secretly writes the link $\mathsf{L}[i] := \mathfrak{p}_{\text{real}}$. $\mathsf{L}[i]$ represents secret-shares of the next pointers for a real element from the first list. The cases for $\mathfrak{p}'_{\text{real}}, \mathfrak{p}_{\text{dummy}}$, and $\mathfrak{p}'_{\text{dummy}}$ are analogous.
   At the end of this reverse scan, the client remembers the position tuple for the first real and first dummy elements of both linked lists. They are denoted by $\mathfrak{p}^1_{\text{real}}$, $\mathfrak{p}'^1_{\text{real}}$, $\mathfrak{p}^1_{\text{dummy}}$, and $\mathfrak{p}'^1_{\text{dummy}}$.

4. We next call $\mathsf{Permute}$ to the combined layout $\{\mathfrak{e}, (\mathsf{U}_b, \mathsf{L}_b)\}_{b \in \mathbb{Z}_3}$, where each server $\mathbf{S}_b$ has input π_{b+1}, to produce $\{\pi_b, (\mathsf{U}'_b, \mathsf{L}'_b)\}_{b \in \mathbb{Z}_3}$ as output.
5. The linked lists can now be accessed using the four position tuples $\mathfrak{p}^1_{\text{real}}$, $\mathfrak{p}'^1_{\text{real}}$, $\mathfrak{p}^1_{\text{dummy}}$, and $\mathfrak{p}'^1_{\text{dummy}}$. The client first starts accessing real elements in the two lists using $\mathfrak{p}^1_{\text{real}}$ and $\mathfrak{p}'^1_{\text{real}}$ to merge them. When a real list ends, it starts accessing the corresponding dummy list.
   More precisely, the client secretly writes the merged result to the abstract output array U''.
   Start with $k \leftarrow 0$, $\mathfrak{p}^1 \leftarrow \mathfrak{p}^1_{\text{real}}$, $\mathfrak{p}^2 \leftarrow \mathfrak{p}^2_{\text{real}}$.
   For each $s \in \{1, 2\}$, the client reconstructs $\mathsf{B}^s := \oplus_b \mathsf{U}'_b[\mathfrak{p}^s_b]$ and $\mathsf{next}^s := \oplus_b \mathsf{L}'_b[\mathfrak{p}^s_b]$ at most once, i.e., if B^s and next^s have already been reconstructed once with the tuple $(\mathfrak{p}^p_b : b \in \mathbb{Z}_3)$, then they will not be reconstructed again.
   If B^1 should appear before B^2, then the client secretly writes $\mathsf{U}''[k] \leftarrow \mathsf{B}^1$ and updates $k \leftarrow k + 1$, $\mathfrak{p}^1 \leftarrow \mathsf{next}^1$; if the end of the real list is reached, then it updates $\mathfrak{p}^1 \leftarrow \mathfrak{p}^1_{\text{dummy}}$. The case when B^2 should appear before B^1 is analogous.
   The next element is processed until the client has secretly constructed the whole abstract array U''.
6. The new merged layout $\{\mathfrak{e}, \mathsf{U}''_b\}_{b \in \mathbb{Z}_3}$ is produced.

Theorem 5. *The* Merge *protocol perfectly securely realizes the ideal functionality* $\mathcal{F}_{\text{merge}}$ *(as per Definition 1) in the presence of a semi-honest adversary corrupting a single server with* $O(n)$ *bandwidth.*

Due to lack of space, the proof is in the full version of the paper [5].

4 Three-Server One-Time Oblivious Memory

We construct an abstract datatype to process non-recurrent memory lookup requests, i.e., between rebuilds of the data structure, each distinct address is requested at most once. Our abstraction is similar to the perfectly secure one-time oblivious memory by Chan et al. [6]. However, while Chan et al. only consider perfect security with respect to access pattern, our three-server one time memory in addition information-theoretically encrypts the data itself. Thus, in [6], since the algorithm does not provide guarantees for the data itself, it can modify the data structure while performing operations. In contrast, our one-time oblivious memory is a read-only data structure. In this data structure, we assume every request is tagged with a position label indicating which memory location to lookup in each of the servers. In this section, we assume that such a position is magically available during lookup; but in subsequent sections we show how this data structure can be maintained and provided during a lookup.

4.1 Definition: Three-Server One-Time Oblivious Memory

Our (three-server) one-time oblivious memory supports three operations: (1) Build, (2) Lookup, and (3) Getall. Build is called once upfront to create the data structure: it takes in a set of data blocks (tagged with its logical address), permutes shares of the data blocks at each of the servers to create a data structure that facilitates subsequent lookup from the servers. Once the data structure is built, lookup operations can be performed on it. Each lookup request consists of a logical address to lookup and a position label for each of the three servers, thereby enabling them to perform the lookup operation. The lookup can be performed for a real logical address, in which case the logical address and the position labels for each of the three servers are provided; or it can be a dummy request, in which case $\perp$ is provided. Finally, a Getall operation is called to obtain a list U of all the blocks that were provided during the Build operation. Later, in our ORAM scheme, the elements in the list U will be combined with those in other lists to construct a potentially larger one-time oblivious memory.

Our three-server one-time oblivious memory maintains obliviousness as long as (1) for each real block in the one-time memory, a lookup is performed at most once, (2) at most n total lookups (all of which could potentially be dummy lookups) are performed, and (3) no two servers collude with each other to learn the shares of the other server.

Formal Definition. Our three-server one-time oblivious memory scheme $\mathsf{OTM}[n]$ is parameterized by n, the number of memory lookup requests supported by the data structure. It is comprised of the following randomized, stateful algorithms:

- $\Big(U, \big(\{\pi_b, (\widehat{\mathsf{T}_b}, \widehat{\mathsf{L}_b})\}_{b\in\mathbb{Z}_3}, \mathsf{dpos}\big)\Big) \leftarrow \mathsf{Build}(\mathsf{T}, \perp)$:
  - *Input:* A sorted, unpermuted layout denoted $\{\mathfrak{e}, \mathsf{T}_b\}_{b\in\mathbb{Z}_3}$ representing an abstract sorted list T. $\mathsf{T}[i]$ represents a key-value pair (key_i, v_i) which are either real and contains a real address key_i and value v_i, or dummy and contains a $\perp$. The list T is sorted by the key key_i. The client's input is $\perp$.
  - *Functionality:* The Build algorithm creates a layout $\{\pi_b, (\widehat{\mathsf{T}_b}, \widehat{\mathsf{L}_b})\}_{b\in\mathbb{Z}_3}$ of size $2n$ that will facilitate subsequent lookup requests; intuitively, n extra dummy elements are added, and the $\widehat{\mathsf{L}_b}$'s maintain a singly-linked list for these n dummy elements. Moreover, the tuple of head positions is secret-shared $\oplus_b \mathsf{dpos}_b$ among the three servers.

    It also outputs a sorted list U of n key-value pairs $(\mathsf{key}, \mathsf{pos})$ sorted by key where each $\mathsf{pos} := (\mathsf{pos}_0, \mathsf{pos}_1, \mathsf{pos}_2)$; the invariant is that if $\mathsf{key} \neq \perp$, then the data for key is $\oplus_b \widehat{\mathsf{T}_b}[\mathsf{pos}_b]$.

    The output list U is stored as a sorted, unpermuted layout $\{\mathfrak{e}, U_b\}_{b\in\mathbb{Z}_3}$. Every real key from T appears exactly once in U and the remaining entries of U are $\perp$'s. The client's output is $\perp$.

    Later in our scheme, U will be propagated back to the corresponding data structure with preceding recursion depth during a coordinated rebuild. Hence, U does not need to carry the value v_i's.
- $v \leftarrow \mathsf{Lookup}\Big(\big(\{\pi_b, (\widehat{\mathsf{T}_b}, \widehat{\mathsf{L}_b})\}_{b\in\mathbb{Z}_3}, \mathsf{dpos}\big), \big(\mathsf{key}, \mathsf{pos}\big)\Big)$:
  - *Input:* The client provides a key key and a position label tuple $\mathsf{pos} := (\mathsf{pos}_0, \mathsf{pos}_1, \mathsf{pos}_2)$. The servers input the data structure $\{\pi_b, (\widehat{\mathsf{T}_b}, \widehat{\mathsf{L}_b})\}_{b\in\mathbb{Z}_3}$ and dpos created during Build.

  - *Functionality:* If $\mathsf{key} \neq \perp$, return $\oplus_b \widehat{\mathsf{T}_b}[\mathsf{pos}_b]$ else, return $\perp$.
- $R \leftarrow \mathsf{Getall}\Big(\{\pi_b, (\widehat{\mathsf{T}_b}, \widehat{\mathsf{L}_b})\}_{b\in\mathbb{Z}_3}, \perp\Big)$:
  - *Input:* The servers input the data structure $\{\pi_b, (\widehat{\mathsf{T}_b}, \widehat{\mathsf{L}_b})\}_{b\in\mathbb{Z}_3}$ created during Build.
  - *Functionality:* the Getall algorithm returns a sorted, unpermuted layout $\{\mathfrak{e}, R_b\}_{b\in\mathbb{Z}_3}$ of length n. This layout represents an abstract sorted list R of key-value pairs where each entry is either real and of the form (key, v) or dummy and of the form $(\perp, \perp)$. The list R contains all real elements inserted during Build including those that have been looked up, padded with $(\perp, \perp)$ to a length of n[2].

Valid request sequence. Our three-server one-time oblivious memory ensures obliviousness only if lookups are non-recurrent (i.e., the same real key is never looked up more than once); and the number of lookups is upper bounded by n, the size of the input list provided to Build. More formally, a sequence of operations is valid, iff the following holds:

- The sequence begins with a single call to Build, followed by a sequence of at most n Lookup calls, and finally the sequence ends with a call to Getall.
- All real keys in the input provided to Build have distinct keys.
- For every Lookup concerning a real element with client's input $(\mathsf{key}, \mathsf{pos} := (\mathsf{pos}_0, \mathsf{pos}_1, \mathsf{pos}_2))$, the key should have existed in the input to Build. Moreover, the position label tuple $(\mathsf{pos}_0, \mathsf{pos}_1, \mathsf{pos}_2)$ must be the correct position labels for each of the three servers.
- No two Lookup requests should request the same real key.

Correctness. Correctness requires that:

1. For any valid request sequence, with probability 1, every Lookup request must return the correct value v associated with key key that was supplied in the Build operation.
2. For any valid request sequence, with probability 1, Getall must return an array R containing every (key, v) pair that was supplied to Build, padded with dummies to have n entries.

Perfect obliviousness. Suppose the following sequence of operations are executed: the initial Build, followed by a valid request sequence of ℓ Lookup's, and the final Getall. Perfect obliviousness requires that for each $b \in \mathbb{Z}_3$, the joint distribution of the communication pattern (between the client and the servers) and the view^b of $\mathbf{S}_b$ is fully determined by the parameters n and ℓ.

[2] The Getall function returns as output the unpermuted layout that was input to Build. It primarily exists for ease of exposition.

4.2 Construction

Intuition. The intuition is to store shares of the input list on storage servers such that each share is independently permuted and each server storing a share does not know its permutation (but some other server does). In order to lookup a real element, if a position label for all three shares are provided, then the client can directly access the shares. Since the shares are permuted and the server storing a share does not know the permutation, each lookup corresponds to accessing a completely random location and is thus perfectly oblivious. This is true so far as each element is accessed exactly once and the position label provided is correct; both of these constraints are satisfied by a valid request sequence. However, in an actual request sequence, some of the requests may be dummy and these requests do not carry a position label with them. To accommodate dummy requests, before permuting the shares, we first append shares of dummy elements to shares of the unpermuted input list. We add enough dummy elements to support all lookup requests before the one time memory is destroyed. Then we create a linked list of dummy elements so that a dummy element stores the position label of the location where the next dummy element is destined to be after permutation. The client maintains the head of this linked list, updating it every time a dummy request is made. To ensure obliviousness, the links (position labels) in the dummy linked list are also stored secret-shared and permuted along with the input list.

Protocol <u>Build</u>. Our oblivious Build algorithm proceeds as follows. Note that the input list T is stored as an unpermuted layout $\{\mathfrak{e}, \mathsf{T}_b\}_{b \in \mathbb{Z}_3}$ on the three servers.

1. *Initialize to add dummies.* Construct an extended abstract $\mathsf{T}'[0..2n-1]$ of length $2n$ such that the first n entries are key-value pairs copied from the input T (some of which may be dummies).
   The last n entries of T' contain *special* dummy keys. For each $i \in [1..n]$, the special dummy key i is stored in $\mathsf{T}'[n-1+i]$, and the entry has a key-value pair denoted by $\perp_i$. For each $i \in [1..n]$, the client secretly writes $\perp_i$ to $\mathsf{T}'[n-1+i]$.
2. *Generate permutations for* OTM. Each server $\mathbf{S}_b$ acts as the permutation server for $\mathbf{S}_{b+1}$. For each $b \in \mathbb{Z}_3$, the client generates a random permutation $\pi_{b+1} : [2n] \to [2n]$ on permutation server $\mathbf{S}_b$.
3. *Construct a dummy linked list.* Using the newly generated permutation π_{b+1} on server $\mathbf{S}_b$, the client constructs a linked list of dummy blocks. This is to enable accessing the dummy blocks linearly, i.e., for each $i \in [1..n-1]$, after accessing dummy block $\perp_i$, the client should be able to access $\perp_{i+1}$.
   The client simply leverages $\pi_{b+1}(n..2n-1)$ stored on server $\mathbf{S}_b$ to achieve this. Specifically, for i from $n-1$ down to 1, to create a link between i-th and $(i+1)$-st dummy, the client reads $\pi_{b+1}(n+i)$ from server $\mathbf{S}_b$ and secretly writes the tuple $(\pi_{b+1}(n+i) : b \in \mathbb{Z}_3)$ to the abstract link $\mathsf{L}[n+i-1]$.
   There are no links between real elements, i.e., for $j \in [0..n-1]$, the client

secretly writes $(\perp, \perp, \perp)$ to (abstract) $\mathsf{L}[j]$.
Observe that these links are secret-shared and stored as an unpermuted layout $\{\mathfrak{e}, \mathsf{L}_b\}_{b \in \mathbf{S}_b}$.
Finally, the client records the positions of the head of the lists and secretly writes the tuple across the three servers, i.e., $\oplus_b \mathsf{dpos}_b := (\pi_b(n) : b \in \mathbb{Z}_3)$, where dpos_b is stored on server $\mathbf{S}_b$.
4. *Construct the key-position map U.* The client can construct the (abstract) key-position map $U[0..n-1]$ sorted by the key from the first n entries of T' and the π_b's. Specifically, for each $i \in [0..n-1]$, the client secretly writes $(\mathsf{key}_i, (\pi_b(i) : b \in \mathbb{Z}_3))$ to $U[i]$.
Recall that U is stored as a sorted, unpermuted layout $\{\mathfrak{e}, U_b\}_{b \in \mathbb{Z}_3}$.
5. *Permute the lists along with the links.* Invoke $\mathsf{Permute}$ with input $\{\mathfrak{e}, (\mathsf{T}'_b, \mathsf{L}_b)\}_{b \in \mathbb{Z}_3}$, and permutation π_{b+1} as the input for $\mathbf{S}_b$. The $\mathsf{Permute}$ protocol returns a permuted output layout $\{\pi_b, (\widehat{\mathsf{T}_b}, \widehat{\mathsf{L}_b})\}_{b \in \mathbb{Z}_3}$.
6. As the data structure, each server $\mathbf{S}_b$ stores $(\widehat{\mathsf{T}_b}, \widehat{\mathsf{L}_b})$, $(\pi_{b+1}, (\widehat{\mathsf{T}_{b+1}}, \widehat{\mathsf{L}_{b+1}}))$, and $\mathsf{dpos}_{\mathsf{b}+1}$. The algorithm returns key-position map list U as output, which is stored as an unpermuted layout $\{\mathfrak{e}, U_b\}_{b \in \mathbb{Z}_3}$. This list will later be passed to the preceding recursion depth in the ORAM scheme during a coordinated rebuild operation.

Fact 6. *The* Build *algorithm for building an* OTM *supporting n lookups requires an $O(n)$ bandwidth.*

Protocol <u>Lookup.</u> Our oblivious $\mathsf{Lookup}\Big(\big(\{\pi_b, (\widehat{\mathsf{T}_b}, \widehat{\mathsf{L}_b})\}_{b \in \mathbb{Z}_3}, \mathsf{dpos}\big), \big(\mathsf{key}, (\mathsf{pos}_0, \mathsf{pos}_1, \mathsf{pos}_2)\big)\Big)$ algorithm proceeds as follows:

1. The client reconstructs $(\mathsf{pos}'_0, \mathsf{pos}'_1, \mathsf{pos}'_2) \leftarrow \oplus_b \mathsf{dpos}_b$.
2. *Decide position to fetch from.* If $\mathsf{key} \neq \perp$, set $\mathsf{pos} \leftarrow (\mathsf{pos}_0, \mathsf{pos}_1, \mathsf{pos}_2)$, i.e., we want to use the position map supplied from the input; if $\mathsf{key} = \perp$, set $\mathsf{pos} \leftarrow (\mathsf{pos}'_0, \mathsf{pos}'_1, \mathsf{pos}'_2)$, i.e., the dummy list will be used.
3. *Reconstruct data block.* Reconstruct $v \leftarrow \oplus \widehat{\mathsf{T}_b}[\mathsf{pos}_b]$ and $(\widehat{\mathsf{pos}}_0, \widehat{\mathsf{pos}}_1, \widehat{\mathsf{pos}}_2) \leftarrow \oplus \widehat{\mathsf{L}_b}[\mathsf{pos}_b]$.
4. *Update head of the dummy linked list.* If $\mathsf{key} \neq \perp$, the client re-shares the secrets $\oplus_b \mathsf{dpos}_b \leftarrow (\mathsf{pos}'_0, \mathsf{pos}'_1, \mathsf{pos}'_2)$ with the same head; if $\mathsf{key} = \perp$, the client secretly shares the updated head $\oplus_b \mathsf{dpos}_b \leftarrow (\widehat{\mathsf{pos}}_0, \widehat{\mathsf{pos}}_1, \widehat{\mathsf{pos}}_2)$.
5. *Read value and return.* Return v.

Fact 7. *The* OTM Lookup *algorithm requires $O(1)$ bandwidth.*

Protocol <u>Getall.</u> For Getall, the client simply invokes the $\mathsf{Unpermute}$ protocol on input layout $\{\pi_b, (\widehat{\mathsf{T}_b}, \widehat{\mathsf{L}_b})\}_{b \in \mathbb{Z}_3}$ and returns the first n entries of the sorted, unpermuted layout (and ignores the links created). This output is also stored as a sorted, unpermuted layout $\{\mathfrak{e}, \mathsf{T}_b\}_{b \in \mathbb{Z}_3}$. The data structure created on the servers during Build can now be destroyed.

Fact 8. *The* OTM Getall *algorithm requires an $O(n)$ bandwidth.*

Lemma 1. *The subroutines* Build*,* Lookup *and* Getall *are correct and perfectly oblivious in the presence of a semi-honest adversary corrupting a single server.*

Due to lack of space, the proofs for these statements are described in the full version of the paper [5].

5 3-Server ORAM with $O(\log^2 N)$ Simulation Overhead

Recall that Sect. 4 provided a construction for a three-server one-time memory that allows non-recurrent lookups so far as its position label is provided. In this section, we first extend this construction to create a hierarchy of one-time memories called position-based ORAM (similar to [6]) where each level acts as a "cache" for larger levels. We will first assume that position-labels are magically available in this position-based ORAM (Sect. 5.1). If a PRF could be used, the position labels could have been obtained using the PRF and this would indeed be an ORAM construction. However, to achieve perfect security, we instead maintain the position labels by recursively storing them in smaller hierarchies (Sect. 5.2).

Our ORAM scheme will consist of logarithmically many position-based ORAMs of geometrically increasing sizes, henceforth denoted ORAM_0, ORAM_1, ..., ORAM_D where $D := \log_2 N$. Specifically, ORAM_d stores $\Theta(2^d)$ blocks where $d \in \{0, 1, \ldots, D\}$. The actual data blocks are stored in ORAM_D whereas all other $\mathsf{ORAM}_d, d < D$ recursively store position labels for the next depth $d + 1$.

5.1 Position-Based ORAM

In this subsection, we focus on describing ORAM_d assuming the position labels are magically available. In the next subsection, we will describe how position labels are maintained across different depths.

Data Structure. For $0 \leq d \leq D$ each ORAM_d consists of $d + 1$ levels of three-server one-time oblivious memory that are geometrically increasing in size. We denote these one-time oblivious memories as $(\mathsf{OTM}_j : j = 0, \ldots, d)$ where $\mathsf{OTM}_j := \mathsf{OTM}[2^j]$ stores at most 2^j real blocks.

Every level j is marked as either *empty* (when the corresponding OTM_j has not been built) or *full* (when OTM_j is ready and in operation). Initially, all levels are empty.

Position label. To access a block stored in ORAM_d, its position label specifies (1) the level $l \in [0..d]$ such that the block resides in OTM_ℓ; and (2) the tuple $\mathsf{pos} := (\mathsf{pos}_0, \mathsf{pos}_1, \mathsf{pos}_2)$ to reconstruct the block from OTM_ℓ.

Operations. Each position-based ORAM supports two operations, Lookup and $\mathsf{Shuffle}$.

Protocol <u>Lookup:</u>

– *Input:* The client provides $\big(\mathsf{key}, \mathsf{pos} := (l, (\mathsf{pos}_0, \mathsf{pos}_1, \mathsf{pos}_2))\big)$ as input, where key is the logical address for the lookup request, l represents the level such that the block is stored in OTM_l, and $(\mathsf{pos}_0, \mathsf{pos}_1, \mathsf{pos}_2)$ is used as an argument for $\mathsf{OTM}_l.\mathsf{Lookup}$.
  The servers store OTM_j for $0 \leq j \leq d$ where OTM stores layout $\big\{\pi_b, (\widehat{\mathsf{T}_b}, \widehat{\mathsf{L}_b})\big\}_{b \in \mathbb{Z}_3}$ and dpos for the level. Moreover, some of the OTMs may be empty.
– *Algorithm:* The lookup operation proceeds as follows:
  1. For each non-empty level $j = 0, \ldots, d$, perform the following:
     - The position label specifies that the block is stored at level OTM_l. For level $j = l$, set $\mathsf{key}' := \mathsf{key}$ and $\mathsf{pos}' := (\mathsf{pos}_0, \mathsf{pos}_1, \mathsf{pos}_2)$. For all other levels, set $\mathsf{key}' := \bot$, $\mathsf{pos}' := \bot$.
     - $v_j \leftarrow \mathsf{OTM}_j.\mathsf{Lookup}\Big(\big(\big\{\pi_b, (\widehat{\mathsf{T}_b}, \widehat{\mathsf{L}_b})\big\}_{b \in \mathbb{Z}_3}, \mathsf{dpos}\big), (\mathsf{key}', \mathsf{pos}')\Big)$.
  2. Return v_l.

Fact 9. *For* ORAM_d*,* Lookup *requires an* $O(d)$ *bandwidth.*

Protocol <u>Shuffle.</u> The shuffle operation is used in hierarchical ORAMs to shuffle data blocks in consecutive smaller levels and place them in the first empty level (or the largest level). Our shuffle operation, in addition, accepts another input U that is used to update the contents of data blocks stored in the position based ORAM. In the final ORAM scheme, the list U passed as an input to ORAM_d will contain the (new) position labels of blocks in ORAM_{d+1}. Similarly, the shuffle operation returns an output U' that will be passed as input to ORAM_{d-1}. More formally, our shuffle operation can be specified as follows:
$(U', \widehat{\mathsf{T}}) \leftarrow \mathsf{Shuffle}_d\big((\mathsf{OTM}_0, \ldots, \mathsf{OTM}_l, U), l\big)$:

– *Input:* The shuffle operation for ORAM_d accepts as input from the client a level l in order to build OTM_l from data blocks currently in levels $0, \ldots, l$. In addition, ORAM_d consists of an extra OTM, denoted by OTM'_0, containing only a single element. Jumping ahead, this single element represents a freshly fetched block.
  The inputs of the servers consist of OTMs for levels up to level l, each of which is stored as a permuted layout $\{\pi_b, (\widehat{\mathsf{T}_b}, \widehat{\mathsf{L}_b})\}_{b \in \mathbb{Z}_3}$ and an array of key-value pairs U, stored as a sorted, unpermuted layout $\{\mathfrak{e}, U_b\}_{b \in \mathbb{Z}_3}$. The array U is used to update the blocks during the shuffle operation.
  Throughout the shuffle operation we maintain the following invariant:
  - For every ORAM_d, $l \leq d$. Moreover, either level l is the smallest empty level of ORAM_d or l is the largest level, i.e., $l = d$.
  - Each logical address appears at most once in U.

  - The input U contains a subset of logical addresses that appear in levels $0, \ldots, l$ of the ORAM_d (or OTM'_0).
    Specifically, given a key-value pair (key, v), the corresponding block (key, v') should already appear in some level in $[0..l]$ or OTM'_0. An update rule will determine how v and v' are combined to produce a new value $\widehat{v}$ for key.

– The Shuffle algorithm proceeds as follows:

1. **Retrieve key-value pairs from** $(\mathsf{OTM}_0, \ldots, \mathsf{OTM}_l)$. The client first retrieves the key-value pairs of real blocks from $(\mathsf{OTM}_0, \ldots, \mathsf{OTM}_l)$ and restore each array to its unpermuted form. More specifically, the client constructs the unpermuted sorted $\mathsf{T}^j \leftarrow \mathsf{OTM}_j.\mathsf{Getall}(\{\pi_b, (\widehat{\mathsf{T}}_b, \widehat{\mathsf{L}_b})\}_{b \in \mathbb{Z}_3}, \perp)$, for $0 \leq j \leq l$, and $\mathsf{T}^0 \leftarrow \mathsf{OTM}'_0.\mathsf{Getall}(\{\pi_b, (\widehat{\mathsf{T}}_b, \widehat{\mathsf{L}_b})\}_{b \in \mathbb{Z}_3}, \perp)$[3] Now, the old $\mathsf{OTM}_0, \ldots, \mathsf{OTM}_l$ instances can be destroyed.
2. **Create a list for level** l**.** The client then creates a level l list of keys from $(\mathsf{OTM}_0, \ldots, \mathsf{OTM}_l)$.
   - *Merge lists from consecutive levels to form level l list.* The merge procedure proceeds as follows:

     For $j = 0, \ldots, l-1$ do:

     $\widehat{\mathsf{T}}^{j+1} \leftarrow \mathsf{Merge}((\widehat{\mathsf{T}}^j, \mathsf{T}^j), \perp)$ where T^j and $\widehat{\mathsf{T}}^j$ are of size 2^j

     Moreover, the lists are individually sorted but may contain blocks that have already been accessed. In the Merge protocol, for two elements with the same key and belonging to different OTM levels, we prefer the one at the smaller level first. For the case where $l = d$, perform another merge $\widehat{\mathsf{T}}^d \leftarrow \mathsf{Merge}((\widehat{\mathsf{T}}^d, \mathsf{T}^d), \perp)$ to produce an array of size 2^{d+1}; Jumping ahead, the size will be reduced back to 2^d in subsequent steps.

     At the end of this step, we obtain a merged sorted list $\widehat{\mathsf{T}}^l$, stored as $\widehat{\mathsf{T}}^l := \{\mathfrak{e}, \widehat{\mathsf{T}}^l_b\}_{b \in \mathbb{Z}_3}$, containing duplicate keys that are stored multiple times (with potentially different values).
   - *Mark duplicate keys as dummy.* From the stored duplicate keys, we only need the value of the one that corresponds to the latest access. All other duplicate entries can be marked as dummies. At a high level, this can be performed in a single pass by the client by scanning consecutive elements of the unpermuted sorted layout $\widehat{\mathsf{T}}^l$. The client keeps the most recent version, i.e., the version that appears first (and has come from the smallest OTM), and marks other versions as dummies. To maintain obliviousness, the secret-shares need to be re-distributed for each scanned entry.

     More specifically, suppose that there are λ duplicate keys. Then, the client scans through the unpermuted layout $\widehat{\mathsf{T}}^l := \{\mathfrak{e}, \widehat{\mathsf{T}}^l_b\}_{b \in \mathbb{Z}_3}$. For consecutive λ elements, $j, \ldots, j + \lambda - 1$ with the same key, the client re-distributes the secret for $\widehat{\mathsf{T}}^l[j]$ for position j, and secretly writes $\perp$

[3] The layout inputs to the Getall operation are restricted to the ones stored in OTM_j for $0 \leq j \leq l$, respectively.

for positions $j+1, \ldots, j+\lambda-1$.
After this step, the resulting (abstract) $\widehat{\mathsf{T}}^l$ is semi-sorted.

- *Compaction to remove dummies.* The client invokes the $\mathsf{StableCompact}$ protocol with input $\widehat{\mathsf{T}}^l := \{\mathfrak{e}, \widehat{\mathsf{T}}^l_b\}_{b \in \mathbb{Z}_3}$, i.e., $\widehat{\mathsf{T}}^l \leftarrow \mathsf{StableCompact}(\widehat{\mathsf{T}}^l, \perp)$ to obtain a sorted, unpermuted layout (where the dummies are at the end). We keep the first 2^l entries.

3. **Update $\widehat{\mathsf{T}}^l$ with values from U.** The client updates $\widehat{\mathsf{T}}^l$ so that it contains updated position values from U. Looking ahead, in our final scheme, U will contain the new position labels from an ORAM at a larger depth. Given that ORAM_D is the largest depth and does not store position values, this step is skipped for ORAM_D.
We do this as follows:
   - *Merge* $\widehat{\mathsf{T}}^l$ with U. The client performs $A \leftarrow \mathsf{Merge}((\widehat{\mathsf{T}}^l, U), \perp)$ to obtain a sorted, unpermuted layout. Ties on the same key break by choosing the blocks in $\widehat{\mathsf{T}}^l$.
   - *Scan and Update A.* In a single pass through the sorted, unpermuted layout A, it can operate on every adjacent pair of entries. If they share the same key, the following update rule is used to update both the values (the precise update rule is provided in the $\mathsf{Convert}$ subroutine in Sect. 5.2). In particular, in the final ORAM scheme, the keys in A correspond to logical addresses. Each address in a position-based ORAM at depth-d stores position labels for two children addresses at depth-$(d+1)$. The entries in A that come from $\widehat{\mathsf{T}}^l$ contain the old position labels for both children. For the entries from U, if children position labels exist, they correspond to the new labels. For each of the child addresses, if U contains a new position label, the update function chooses the new one; otherwise, it chooses the old label from $\widehat{\mathsf{T}}^l$.
   - *Compaction to remove dummies.* The client invokes the $\mathsf{StableCompact}$ protocol $A \leftarrow \mathsf{StableCompact}(A, \perp)$ to obtain an updated sorted, unpermuted layout A. We keep the first 2^l entries.
4. **Build OTM_l.** The client invokes $U' \leftarrow \mathsf{Build}(A, \perp)$ to generate a data structure OTM_l and U'. Mark OTM_l as *full* and OTM_i, for $i < l$, as *empty*.

We prove that the above position-based ORAM is correct and satisfies perfect obliviousness in the presence of a semi-honest adversary corrupting a single server in the full version of the paper [5].

5.2 ORAM Construction from Position-Based ORAM

Our ORAM scheme consists of $D+1$ position-based ORAMs denoted as ORAM_0, ..., ORAM_D where $D = \log_2 N$. ORAM_D stores data blocks whereas ORAM_d for $d < D$ stores a position map for ORAM_{d+1}. The previous section specified the construction of a position-based ORAM. However, it assumed that position labels are magically available at some ORAM_d. In this section, we show a full

ORAM scheme and specify (1) how these position labels for ORAM_d are obtained from ORAM_{d-1}, and (2) after a level of ORAM_d is built, how the position labels of blocks from the new level are updated at ORAM_{d-1}.

Format of block address at depth d. Suppose that a block's logical address is a $\log_2 N$-bit string denoted by $\mathsf{addr}^{\langle D\rangle} := \mathsf{addr}[1..(\log_2 N)]$ (expressed in binary format), where $\mathsf{addr}[1]$ is the most significant bit. In general, at depth d, an address $\mathsf{addr}^{\langle d\rangle}$ is the length-d prefix of the full address $\mathsf{addr}^{\langle D\rangle}$. Henceforth, we refer to $\mathsf{addr}^{\langle d\rangle}$ as a depth-d address (or the depth-d truncation of addr).

When we look up a data block, we would look up the full address $\mathsf{addr}^{\langle D\rangle}$ in recursion depth D; we look up $\mathsf{addr}^{\langle D-1\rangle}$ at depth $D-1$, $\mathsf{addr}^{\langle D-2\rangle}$ at depth $D-2$, and so on. Finally at depth 0, only one block is stored at ORAM_0.

A block with the address $\mathsf{addr}^{\langle d\rangle}$ *in* ORAM_d *stores the position labels for two blocks in* ORAM_{d+1}, *at addresses* $\mathsf{addr}^{\langle d\rangle}||0$ *and* $\mathsf{addr}^{\langle d\rangle}||1$ *respectively.* Henceforth, we say that the two addresses $\mathsf{addr}^{\langle d\rangle}||0$ and $\mathsf{addr}^{\langle d\rangle}||1$ are *siblings* to each other; $\mathsf{addr}^{\langle d\rangle}||0$ is called the left sibling and $\mathsf{addr}^{\langle d\rangle}||1$ is called the right sibling. We say that $\mathsf{addr}^{\langle d\rangle}||0$ is the left child of $\mathsf{addr}^{\langle d\rangle}$ and $\mathsf{addr}^{\langle d\rangle}||1$ is the right child of $\mathsf{addr}^{\langle d\rangle}$.

An ORAM Lookup. An ORAM lookup request is denoted as $(\mathsf{op}, \mathsf{addr}, \mathsf{data})$ where $\mathsf{op} \in \{\mathsf{read}, \mathsf{write}\}$. If $\mathsf{op} = \mathsf{read}$ then $\mathsf{data} := \bot$. Here, addr denotes the address to lookup from the ORAM. The inputs are all provided by the client whereas the servers store position-based $\mathsf{ORAM}_0, \ldots, \mathsf{ORAM}_D$ as discussed in the previous section. We perform the following operations:

1. **Fetch.** For $d := 0$ to D, perform the following:
   - Let $\mathsf{addr}^{\langle d\rangle}$ denote the depth-d truncation of $\mathsf{addr}^{\langle D\rangle}$.
   - Call $\mathsf{ORAM}_d.\mathsf{Lookup}$ to lookup $\mathsf{addr}^{\langle d\rangle}$. Recall that the position labels for the block will be obtained from the lookup of ORAM_{d-1}. For ORAM_0, no position label is needed.
   - The block returned from Lookup is placed in a special OTM'_0 in ORAM_d. Jumping ahead, this will be merged with the rest of the data structure in the maintain phase.
   - If $d < D$, each lookup will return two positions for addresses $\mathsf{addr}^{\langle d\rangle}||0$ and $\mathsf{addr}^{\langle d\rangle}||1$. One of these will correspond to the position of $\mathsf{addr}^{\langle d+1\rangle}$ which will be required in the lookup for ORAM_{d+1}.
   - If $d = D$, the outcome of Lookup will contain the data block fetched.
2. **Maintain.** We first consider depth D. Set depth-D's update array $U^D := \emptyset$. Suppose l^D is the smallest empty level in ORAM_D. We have the invariant that for all $0 \leq d < D$, if $l^D < d$, then l^D is also the smallest empty level in ORAM_d.

   For $d := D$ to 0, perform the following:
   (a) If $d < l^D$, set $l := d$; otherwise, set $l := l^D$.
   (b) Call $U \leftarrow \mathsf{ORAM}_d.\mathsf{Shuffle}((\mathsf{OTM}^d_0, \ldots, \mathsf{OTM}^d_l, U^d), l)$.
   Recall that to complete the description of $\mathsf{Shuffle}$, we need to specify

the update rule that determines how to combine the values of the same address that appears in both the current ORAM_d and U^d.
For $d < D$, in U^d and ORAM_d, each depth-d logical address $\mathsf{addr}^{\langle d \rangle}$ stores the position labels for both children addresses $\mathsf{addr}^{\langle d \rangle}||0$ and $\mathsf{addr}^{\langle d \rangle}||1$ (in depth $d+1$). For each of the child addresses, if U^d contains a new position label, choose the new one; otherwise, choose the old label previously in ORAM_{d-1}.

(c) If $d \geq 1$, we need to send the updated positions involved in U to depth $d-1$. We use the $\mathsf{Convert}$ subroutine (detailed description below) to convert U into an update array for depth-$(d-1)$ addresses, where each entry may pack the position labels for up to two sibling depth-d addresses.
Set $U^{d-1} \leftarrow \mathsf{Convert}(U, d)$, which will be used in the next iteration for recursion depth $d-1$ to perform its shuffle.

The Convert subroutine. U is a sorted, unpermuted layout representing the abstract array $\{(\mathsf{addr}_i^{\langle d \rangle}, \mathsf{pos}_i) : i \in [|U|]\}$. The subroutine $\mathsf{Convert}(U, d)$ proceeds as follows.

For $i := 0$ to $|U|$, the client reconstructs $(\mathsf{addr}_{i-1}^{\langle d \rangle}, \mathsf{pos}_{i-1}), (\mathsf{addr}_i^{\langle d \rangle}, \mathsf{pos}_i)$ and $(\mathsf{addr}_{i+1}^{\langle d \rangle}, \mathsf{pos}_{i+1})$, computes u_i' using the rules below and secretly writes u_i' to U^{d-1}.

– If $\mathsf{addr}_i^{\langle d \rangle} = \mathsf{addr}||0$ and $\mathsf{addr}_{i+1}^{\langle d \rangle} = \mathsf{addr}||1$ for some addr, i.e., if my right neighbor is my sibling, then write down $u_i' := (\mathsf{addr}, (\mathsf{pos}_i, \mathsf{pos}_{i+1}))$, i.e., both siblings' positions need to be updated.
– If $\mathsf{addr}_{i-1}^{\langle d \rangle} = \mathsf{addr}||0$ and $\mathsf{addr}_i^{\langle d \rangle} = \mathsf{addr}||1$ for some addr, i.e., if my left neighbor is my sibling, then write down $u_i' := \perp$.
– Else if i does not have a neighboring sibling, parse $\mathsf{addr}_i^{\langle d \rangle} = \mathsf{addr}||b$ for some $b \in \{0, 1\}$, then write down $u_i' := (\mathsf{addr}, (\mathsf{pos}_i, *))$ if $b = 0$ or write down $u_i' := (\mathsf{addr}, (*, \mathsf{pos}_i))$ if $b = 1$. In these cases, only the position of one of the siblings needs to be updated in ORAM_{d-1}.
– Let $U^{d-1} := \{u_i' : i \in [|U|]\}$. Note here that each entry of U^{d-1} contains a depth-$(d-1)$ address of the form addr, as well as the update instructions for two position labels of the depth-d addresses $\mathsf{addr}||0$ and $\mathsf{addr}||1$ respectively. We emphasize that when $*$ appears, this means that the position of the corresponding depth-d address does not need to be updated in ORAM_{d-1}.
– Output U^{d-1}.

Lemma 2. *The above ORAM scheme is perfectly oblivious in the presence of a semi-honest adversary corrupting a single server.*

Fact 10. *Each ORAM access takes an amortized bandwidth blowup of $O(\log^2 N)$.*

Due to lack of space, the proofs are in the full version of the paper [5]. Summarizing the above, we arrive at the following main theorem:

Theorem 11 (Perfectly secure 3-server ORAM). *There exists a 3-server ORAM scheme that satisfies perfect correctness and perfect security (as per Sect. 2.3) against any single semi-honest server corruption with $O(\log^2 N)$ amortized bandwidth blowup (where N denotes the total number of logical blocks).*

Finally, similar to existing works that rely on the recursion technique [24,26], we can achieve better bandwidth blowup with larger block sizes: suppose each data block is at least $\Omega(\log^2 N)$ in size, and we still set the position map blocks to be $O(\log N)$ bits long, then our scheme achieves $O(\log N)$ bandwidth blowup.

Acknowledgments. T-H. Hubert Chan was supported in part by the Hong Kong RGC under grant 17200418. Jonathan Katz was supported in part by NSF award #1563722. Kartik Nayak was supported by a Google Ph.D. fellowship. Antigoni Polychroniadou was supported by the Junior Simons Fellowship awarded by the Simons Society of Fellows. Elaine Shi was supported in part by NSF award CNS-1601879, a Packard Fellowship, and a DARPA Safeware grant (subcontractor under IBM).

References

1. Abraham, I., Fletcher, C.W., Nayak, K., Pinkas, B., Ren, L.: Asymptotically tight bounds for composing ORAM with PIR. In: Fehr, S. (ed.) PKC 2017. LNCS, vol. 10174, pp. 91–120. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54365-8_5
2. Ajtai, M.: Oblivious RAMs without cryptographic assumptions. In: STOC (2010)
3. Chan, T.-H.H., Guo, Y., Lin, W.-K., Shi, E.: Oblivious hashing revisited, and applications to asymptotically efficient ORAM and OPRAM. In: Takagi, T., Peyrin, T. (eds.) ASIACRYPT 2017. LNCS, vol. 10624, pp. 660–690. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-70694-8_23
4. Chan, T.-H.H., Guo, Y., Lin, W.K., Shi, E.: Cache-oblivious and data-oblivious sorting and applications. In: SODA (2018)
5. Chan, T.-H.H., Katz, J., Nayak, K., Polychroniadou, A., Shi, E.: More is less: Perfectly secure oblivious algorithms in the multi-server setting. CoRR, abs/1809.00825 (2018)
6. Chan, T.-H.H., Nayak, K., Shi, E.: Perfectly secure oblivious parallel RAM. In: TCC (2018)
7. Chan, T.-H.H., Shi, E.: Circuit OPRAM: a unifying framework for computationally and statistically secure ORAMs and OPRAMs. In: TCC (2017)
8. 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. LNCS, vol. 8874, pp. 62–81. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-45608-8_4
9. Damgård, I., Meldgaard, S., Nielsen, J.B.: Perfectly secure oblivious RAM without random oracles. In: Ishai, Y. (ed.) TCC 2011. LNCS, vol. 6597, pp. 144–163. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-19571-6_10
10. Demertzis, I., Papadopoulos, D., Papamanthou, C.: Searchable encryption with optimal locality: achieving sublogarithmic read efficiency. In: Shacham, H., Boldyreva, A. (eds.) CRYPTO 2018. LNCS, vol. 10991, pp. 371–406. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96884-1_13

11. Knuth, D.E.: The Art of Computer Programming, Volume 2 (3rd edn.): Seminumerical Algorithms. Addison-Wesley Longman Publishing Co., Inc., Boston (1997). ISBN: 0-201-89684-2
12. Goldreich, O.: Towards a theory of software protection and simulation by oblivious RAMs. In: STOC (1987)
13. Goldreich, O., Ostrovsky, R.: Software protection and simulation on oblivious RAMs. J. ACM **43**(3), 431–473 (1996). https://doi.org/10.1145/233551.233553. ISSN: 0004-5411
14. Goodrich, M.T.: Data-oblivious external-memory algorithms for the compaction, selection, and sorting of outsourced data. In: SPAA (2011)
15. 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. LNCS, vol. 6756, pp. 576–587. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22012-8_46
16. Gordon, D., Katz, J., Wang, X.: Simple and efficient two-server ORAM. In: Asiacrypt (2018)
17. Hoang, T., Ozkaptan, C.D., Yavuz, A.A., Guajardo, J., Nguyen, T.: S3ORAM: a computation-efficient and constant client bandwidth blowup ORAM with shamir secret sharing. In: CCS (2017)
18. Kushilevitz, E., Lu, S., Ostrovsky, R.: On the (in)security of hash-based oblivious RAM and a new balancing scheme. In: SODA (2012)
19. Kushilevitz, E., Mour, T.: Sub-logarithmic distributed oblivious RAM with small block size. CoRR, abs/1802.05145 (2018)
20. Lin, W.-K., Shi, E., Xie, T.: Can we overcome the $n \log n$ barrier for oblivious sorting? Cryptology ePrint Archive, Report 2018/227 (2018)
21. Lu, S., Ostrovsky, R.: Distributed oblivious RAM for secure two-party computation. In: Sahai, A. (ed.) TCC 2013. LNCS, vol. 7785, pp. 377–396. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36594-2_22
22. Raskin, M., Simkin, M.: Oblivious RAM with small storage overhead. Cryptology ePrint Archive, Report 2018/268 (2018). https://eprint.iacr.org/2018/268
23. Ren, L., et al.: Constants count: practical improvements to oblivious RAM. In: USENIX Security Symposium, pp. 415–430 (2015)
24. 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). https://doi.org/10.1007/978-3-642-25385-0_11
25. Stefanov, E., Shi, E.: Multi-cloud oblivious storage. In: CCS (2013)
26. Stefanov, E., et al.: Path ORAM - an extremely simple oblivious RAM protocol. In: CCS (2013)
27. Tople, S., Dang, H., Saxena, P., Chang, E.-C.: Permuteram: Optimizing oblivious computation for efficiency. Cryptology ePrint Archive, Report 2017/885 (2017)
28. Wang, X.S., Chan, T.-H.H., Shi, E.: Circuit ORAM: on tightness of the Goldreich-Ostrovsky lower bound. In: ACM CCS (2015)