Is there an Oblivious RAM Lower Bound for Online Reads?

Mor Weiss[*] Daniel Wichs[†]

June 19, 2018

Oblivious RAM (ORAM), introduced by Goldreich and Ostrovsky (JACM 1996), can be used to read and write to memory in a way that hides which locations are being accessed. The best known ORAM schemes have an $O(\log n)$ overhead per access, where n is the data size. The work of Goldreich and Ostrovsky gave a lower bound showing that this is optimal for ORAM schemes that operate in a "balls and bins" model, where memory blocks can only be shuffled between different locations but not manipulated otherwise. The lower bound even extends to weaker settings such as *offline* ORAM, where all of the accesses to be performed need to be specified ahead of time, and *read-only* ORAM, which only allows reads but not writes. But can we get lower bounds for general ORAM, beyond "balls and bins"?

The work of Boyle and Naor (ITCS '16) shows that this is unlikely in the *offline* setting. In particular, they construct an offline ORAM with $o(\log n)$ overhead assuming the existence of small sorting circuits. Although we do not have instantiations of the latter, ruling them out would require proving new circuit lower bounds. On the other hand, the recent work of Larsen and Nielsen (CRYPTO '18) shows that there indeed is an $\Omega(\log n)$ lower bound for general *online* ORAM.

This still leaves the question open for online *read-only* ORAM or for *read/write* ORAM where we want very small overhead for the read operations. In this work, we show that a lower bound in these settings is also unlikely. In particular, our main result is a construction of online ORAM where *reads* (but not *writes*) have an $o(\log n)$ overhead, assuming the existence of small sorting circuits as well as very good *locally decodable codes (LDCs)*. Although we do not have instantiations of either of these with the required parameters, ruling them out is beyond current lower bounds.

[*] Department of Computer Science, Northeastern University, Boston, Massachusetts, USA, m.weiss@northeastern.edu.

[†] Department of Computer Science, Northeastern University, Boston, Massachusetts, USA, wichs@ccs.neu.edu.

1 Introduction

An *Oblivious RAM (ORAM)*, first introduced by Goldreich and Ostrovsky [Gol87, Ost90, GO96], is a scheme that allows a client to read and write to a database stored on a remote server, while entirely hiding the access pattern, i.e., which operations were performed and at which locations. More precisely, for every pair of equal length access patterns, the resultant distributions over actual accesses to the server's storage are computationally (or statistically) close. Following its introduction, there has been a large body of work on ORAM constructions and security [SCSL11, GMOT12, KLO12, WS12, SvDS$^+$13, RFK$^+$15, DvDF$^+$16], as well as its uses in various application scenarios (see, e.g., [GKK$^+$12, GGH$^+$13, LPM$^+$13, LO13, MLS$^+$13, SS13, YFR$^+$13, CKW13, WHC$^+$14, MBC14, KS14, LHS$^+$14, GHJR15, BCP15, HOWW18]).

One can always trivially hide the memory access pattern by performing a linear scan of the entire memory for *every* memory access. Consequently, an important measure of an ORAM scheme is its *overhead*, namely the number of memory blocks which need to be accessed to answer a *single* read or write request. Goldreich and Ostrovsky [GO96] proved a lower bound of $\Omega(\log n)$ on the ORAM overhead, where n denotes the number of memory blocks in the database. There are also ORAM constructions achieving this bound [SvDS$^+$13, WCS15], at least if the block size is set to a sufficiently large polylogarithmic term. We note that one can circumvent the [GO96] lower bound by *relaxing* the notion of ORAM to either allow server-side computation [AKST14], or multiple non-colluding servers [LO13], and several works have obtained *sub-logarithmic overhead* in these settings [AKST14, FNR$^+$15, DvDF$^+$16, ZMZQ16, AFN$^+$17, WGK18, KM18].

In some respects, the lower bound of [GO96] is very general. First, it applies to all block sizes. Second, it holds also in restricted settings: when the ORAM is only required to work for *offline* programs in which, roughly, all memory accesses are stated explicitly in advance; and for *read-only* programs that do not update the memory contents. However, in other respects, the bound is restricted since it only applies to ORAM schemes that operate in the *"balls and bins"* model, in which memory can only be manipulated by moving memory blocks ("balls") from one memory location ("bin") to another. Therefore, the main question left open by the work of [GO96] is: *is there an ORAM lower bound for* general *ORAM schemes, that are not restricted to operate in the "balls and bins" model?*

Almost 20 years after Goldreich and Ostrovsky proved their lower bound, it was revisited by Boyle and Naor [BN16], who show how to construct an ORAM scheme *in the offline setting* with $o(\log n)$ overhead, using sorting circuits of size $o(n \log n)$. Though sorting circuits of such size are not known, ruling out their existence seems currently out of our reach. This result can be interpreted in two ways. On the one hand, an optimist will view it as a possible approach towards an ORAM construction in the offline setting, which uses "small" sorting circuits as a building block. On the other hand, a pessimist may view this result as a barrier towards proving a lower bound. Indeed, the [BN16] construction shows that proving a lower bound on the overhead of offline ORAM schemes would yield lower bounds on the size of sorting circuits, and proving circuit lower bounds is notoriously difficult. We note that unlike sorting *networks*, which only contain "compare-and-swap" gates that operate on the two input words as a whole, and for which a simple $\Omega(n \log n)$ lower bound exists, sorting *circuits* can arbitrarily operate over the input bits, and no such lower bounds are known for them.

The main drawback of the Boyle and Naor result [BN16] is that it only applies to the *offline* setting, which is not very natural and is insufficient for essentially any imaginable ORAM application. More specifically, the offline setting requires that *the entire sequence of accesses* be specified in advance - including *which operation* is performed, on *which address*, and in case of a write operation, *what value* is written. However, even very simple and natural RAM programs (e.g., binary search) require dynamic memory accesses that depend on the results of previous operations. Despite this drawback, the result of Boyle and Naor is still very interesting since it shows that lower bounds which are easy to prove in the "balls and bins" model might not extend to the general model. However, it does not answer the question of whether general ORAM lower bounds exist in the *online* setting, which is the one of interest for virtually all ORAM applications.

Very recently, and concurrently with our work, Larsen and Nielsen [LN18] proved that the [GO96] lower bound *does* indeed extend to general *online* ORAM. Concretely, they show an $\Omega(\log n)$ lower bound on the *combined* overhead of read and write operations in any general online ORAM, even with computational security. Their elegant proof employs techniques from the field of data-structure lower bounds in the cell-probe model, and in particular the "information-transfer" method of Pătraşcu and Demaine [PD06].

1.1 Our Contributions

In this work, we explore the *read overhead* of general ORAM schemes *beyond the "balls and bins" model* and in the *online setting*. We first consider *read-only* ORAM schemes that only support reads – but not writes – to the database. We note that read-only ORAM already captures many interesting applications such as private search over a database, or fundamental algorithmic tasks such as binary search. We show how to construct *online* read-only ORAM schemes with $o(\log n)$ overhead assuming "small" sorting circuits and "good" Locally Decodable Codes (LDCs). We then extend our results to a setting which also supports sub-linear writes but does not try to hide whether an operation is a read or a write and, in particular, allows different overheads for these operations.

We note that, similar to [BN16], our results rely on primitives that we do not know how to instantiate with the required parameters, but also do not have any good lower bounds for. One can therefore interpret our results either positively, as a blueprint for an ORAM construction, or negatively as a barrier to proving a lower bound in these settings. For simplicity of the exposition, we choose to present our results through the "optimistic" lens.

We now describe our results in more detail.

Read-Only (RO) ORAM. We construct a read-only ORAM scheme, based on sorting circuits and smooth locally decodable codes. Roughly, a Locally Decodable Code (LDC) [KT00] has a decoder algorithm that can recover any message symbol by querying only few codeword symbols. In a smooth code, every individual decoder query is uniformly distributed. Given a size-n database, our scheme has $O(\log\log n)$ overhead, assuming the existence of linear-size sorting circuits, and smooth LDCs with constant query complexity and polynomial length codewords. Concretely, we get the following theorem.

Theorem 1.1 (Informal statement of Corollary 3.2)**.** *Suppose there exist linear-size boolean sorting circuits, and smooth LDCs with constant query complexity and polynomial length codewords. Then there exists a statistically-secure read-only ORAM scheme for databases of size* n *and blocks of size* $\mathrm{poly}\log n$*, with* $O(1)$ *client storage and* $O(\log\log n)$ *overhead.*

In Section 3, we also show a read-only ORAM scheme with $o(\log n)$ overhead based on milder assumptions – concretely, smooth LDCs with $\log\log n$ query complexity, and the existence of sorting circuits of size $o\left(\frac{n\log n}{\log^2\log n}\right)$; see Corollary 3.3. We note that if an a-priori polynomial bound on the number of memory accesses is known, then the constructions can be based solely on LDCs, and the assumption regarding small sorting circuits can be removed.

ORAM schemes supporting writes. The read-only ORAM scheme described above still leaves the following open question: *is there a lower bound on* read *overhead for ORAM schemes supporting* write *operations?* To partially address this question, we extend our ORAM construction to a scheme that supports writes but does not hide whether an operation was a read or a write. In this setting, read and write operations may have different overheads, and we focus on minimizing the overhead of read operations while preserving efficiency of write operations as much as possible. Our construction is based on the existence of sorting circuits and smooth LDCs as in Theorem 1.1, where additionally the LDC has a"small" encoding circuit. Assuming the existence of such building blocks, our scheme has $O(\log\log n)$ read overhead and $O(n^{\epsilon})$ write overhead for an arbitrarily small constant $\epsilon \in (0,1)$, whose exact value depends on the efficiency of the LDC encoding. Concretely, we show the following:

Theorem 1.2 (Informal statement of Theorem 4.1)**.** *Assume the existence of LDCs and sorting circuits as in Theorem 1.1, where the LDC has an encoding circuit of size $n^{1+\gamma}$ for some $\gamma \in (0,1)$. Then there exists a statistically-secure ORAM scheme for databases of size n and blocks of size $\operatorname{poly}\log n$ with $O(1)$ client storage, $O(\log\log n)$ read overhead, and $O(n^{\epsilon})$ write overhead for a constant $\epsilon \in (0,1)$.*

Similar to the read-only setting, we also instantiate (Section 4, Theorem 4.2) the ORAM with writes scheme based on milder assumptions regarding the parameters of the underlying sorting circuits and LDCs, while only slightly increasing the read overhead. Additionally, we describe a variant of our scheme with improved write complexity, again at the cost of slightly increasing the read overhead:

Theorem 1.3 (Informal statement of Theorem 4.3)**.** *Assume the existence of LDCs and sorting circuits as in Theorem 1.1, where the LDC has an encoding circuit of size $n^{1+o(1)}$. Then there exists a statistically-secure ORAM scheme for databases of size n and blocks of size $\operatorname{poly}\log n$ with $O(1)$ client storage, $o(\log n)$ read overhead, and $n^{o(1)}$ write overhead.*

A note on block vs. word size. In our constructions we distinguish between *words* (which are bit strings) and *blocks* (which consist of several words). We measure the overhead as the number of *blocks* communicated to read or write to a single memory block, but allow client-server communication to be in *words*. Since it is easier to construct schemes with *smaller* word size, we would generally like to support *larger* word size, ideally having words and blocks of equal size. Our constructions can handle any sufficiently large word size, as long as blocks are poly-logarithmically larger (for a sufficiently large poylogarithmic factor).

A note regarding assumptions. We instantiate our constructions in two parameter regimes: one based on the existence of "best possible" sorting circuits and smooth LDCs (as described above), and one based on milder assumptions regarding the parameters of these building blocks (as discussed in Sections 3 and 4). We note that despite years of research in these fields, we currently seem very far from ruling out the existence of even the "best possible" sorting circuits and smooth LDCs. Concretely, to the best of our knowledge there are no specific lower bounds for sorting circuits (as opposed to sorting networks, see discussion above and in Section 2.2), and even for general boolean circuits only linear lower bounds of $c \cdot n$ for some constant $c > 1$ are known [Blu84, IM02, FGHK16]. Regarding LDCs, research has focused on the relation between the query complexity and codeword length in the constant query regime, but there are currently no non-trivial lower bounds for *general* codes. Even for restricted cases, such as *binary* codes, or *linear* codes over arbitrary fields, the bounds are extremely weak. Specifically, the best known lower bound shows that codewords in q-query LDCs must have length $\Omega\left(n^{(q+1)/(q-1)}\right)/\log n$ [Woo07] (which, in particular, does not rule out the existence of 4-query LDCs with codeword length $n^{5/3}$), so it is plausible that for a sufficiently large constant, constant-query LDCs with polynomial length codewords exist. We note that a recent series of breakthrough results construct *3-query* LDCs with *sub-exponential* codewords of length $\exp\left(\exp\left(O\left(\sqrt{\log n \log\log n}\right)\right)\right) = 2^{n^{o(1)}}$, as well as extensions to larger (constant) query complexity [Yek07, Rag07, Efr09, IS10, CFL$^+$13]. Notice that lower bounds on the size of the encoding circuit of such codes will similarly yield circuit lower bounds.

1.2 Our Techniques

We now give a high-level overview of our ORAM constructions. We start with the read-only setting, and then discuss how to enable writes.

We note that our technique departs quite significantly from that of Boyle and Naor [BN16], whose construction seems heavily tied to the offline setting. Indeed, the high-level idea underlying their scheme is to use the sorting circuit to sort by location the list of operations that need to be performed, so that the outcomes of the read operations can then be easily determined by making one linear scan of the list. It does not appear that this strategy can naturally extend to the *online* setting in which the memory accesses are not known a-priori.

1.2.1 Read-Only ORAM

We first design a Read-Only (RO) ORAM scheme that is secure only for an *a-priori bounded* number of accesses, then extend it to a scheme that remains secure for *any* polynomial number of accesses.

Bounded-access RO-ORAM using metadata. Our RO-ORAM scheme employs a smooth LDC, using the decoder to read from the database. Recall that a k-query LDC is an error-correcting code in which every message symbol can be recovered by querying k codeword symbols. The server in our scheme stores k copies of the codeword, each permuted using a separate, random permutation. (We note that permuted LDCs were already used – but in a very different way – in several prior works [HO08, HOSW11, CHR17, BIPW17].) To read the memory block at address j, the client runs the decoder on j, and sends the decoder queries to the server, who uses the i'th permuted codeword copy to answer the i'th decoding query. This achieves correctness, but does not yet guarantee obliviousness since the server learns, for each $1 \leq i \leq k$, which read operations induced the same i'th decoding query.

To prevent the server from obtaining this additional information, we restrict the client to use only *fresh* decoding queries in each read operation, namely a set $q_1, \ldots, q_k$ of queries such that no q_i was issued before as the i'th query. The metadata regarding which decoding queries are fresh, as well as the description of the permutations, can be stored on the server using any sufficiently efficient (specifically, polylogarithmic-overhead) ORAM scheme. Each block in the metadata ORAM will consist of a single word, so using the metadata ORAM will not influence the overall complexity of the scheme, since for databases with sufficiently large memory blocks the metadata blocks are significantly smaller. In summary, restricting the client to make fresh queries guarantees that the server only sees uniformly random decoding queries, which reveal no information regarding the identity of the accessed memory blocks.

However, restricting the client to only make fresh decoding queries raises the question of whether the ORAM is still correct, namely whether this restriction has not harmed functionality. Specifically, *can the client always "find" fresh decoding queries?* We show this is indeed the case as long as the number of read operations is $M/2k$, where M denotes the codeword length. More precisely, the smoothness of the code guarantees that for security parameter λ and any index $j \in [n]$, λ independent executions of the decoder algorithm on index j will (with overwhelming probability) produce at least one set of fresh decoding queries. Thus, the construction is secure as long as the client performs at most $M/2k$ read operations.

We note that given an appropriate LDC, this construction already gives a read-only ORAM scheme which is secure for an a-priori *bounded* number of accesses, *without relying on sorting circuits*. Indeed, given a bound B on the number of accesses, all we need is a smooth LDC with length-M codewords, in which the decoder's query complexity is at most $M/2B$.

Handling an unlimited number of reads. To obtain security for an *unbounded* number of read operations, we "refresh" the permuted codeword copies every $M/2k$ operations. (We call each such set of read operations an "epoch".) Specifically, to refresh the codeword copies the client picks k fresh, random permutations, and together with the server uses the sorting circuit to permute the codeword copies according to the new permutations. Since the database is read-only, the refreshing operations can be spread-out across the $M/2k$ read operations of the epoch.

1.2.2 ORAM with Writes

We extend our RO-ORAM scheme to support write operations, while preserving $o(\log n)$ overhead for read operations. The construction is loosely based on hierarchical ORAM [Ost90, GO96]. The high-level idea is to store the database on the server in a sequence of ℓ levels of increasing size, each containing a RO-ORAM. We think of the levels as growing from the top down, namely level-1 (the smallest) is the top-most level, and level-ℓ (the largest) is the bottom-most. Initially, all the data is stored in the bottom level ℓ, and all the remaining levels are empty. To read the memory block at some location j, the client performs a read for location j in the RO-ORAMs of all levels, where the output is the block from the highest level that contains the j'th block. When the client writes to some location j, the server places that memory block in the top level $i = 1$. After every l_i write operations – where l_i denotes the size of level i – the i'th level becomes full. All the values in level i are then moved to level $i + 1$, a process which we call a "reshuffle" of level i into level $i + 1$. Formalizing this high-level intuition requires some care, and the final scheme is somewhat more involved. See Section 4 for details.

We note that our construction differs from Hierarchical ORAM in two main points. First, Hierarchical ORAM uses $\Omega(\log n)$ levels, whereas to preserve $o(\log n)$ read overhead, we must use $o(\log n)$ levels. Second, in Hierarchical ORAM level i is reshuffled into level $i + 1$ every l_i *read or write operations*, whereas in our scheme only write are "counted" towards reshuffle (in that respect, read operations are "free"). This is because the data is stored in each level using a RO-ORAM which already guarantees privacy for read operations.

2 Preliminaries

Throughout the paper λ denotes a security parameter. For a length-n string $\mathbf{x}$ and a subset $I = \{i_1, \ldots, i_l\} \subseteq [n]$, $\mathbf{x}_I$ denotes $(x_{i_1}, \ldots, x_{i_l})$.

Terminology. We consider algorithms that operate over *words* represented as bit strings, and *blocks* which consist of lists of words. The client and server may *locally* perform bit operations on the bit representation of words, but can only send full words. We will usually measure complexity in blocks. More specifically, unless explicitly stated otherwise, client and server storage are measured as the number of *blocks* they store, and computation/communication overhead measures the number of blocks one needs to read or write to implement a read or write operation on a single block. Formally:

Definition 2.1 (Overhead)**.** For a block size B and input length n, we say that a protocol between client C and server S has computation (respectively, communication) overhead Ovh for a function $\mathsf{Ovh} : \mathbb{N} \to \mathbb{N}$, if implementing a read or write operation on a single memory block stored at the server requires the server to perform (respectively, to send) at most $\mathsf{B} \cdot \mathsf{Ovh}\left(n\right)$ bit operations (respectively, bits).

2.1 Locally Decodable Codes (LDCs)

Locally decodable codes were first formally introduced by [KT00]. We rely on the following definition of smooth LDCs.

Definition 2.2 (Smooth LDC)**.** A smooth k-query *Locally Decodable Code (LDC)* with message length n, and codeword length M over alphabet Σ, denoted by $(k, n, M)_{\Sigma}$*-smooth LDC*, is a triplet $(\mathsf{Enc}, \mathsf{Query}, \mathsf{Dec})$ of PPT algorithms with the following properties.

- **Syntax.** Enc is given a message $\mathsf{msg} \in \Sigma^n$ and outputs a codeword $c \in \Sigma^M$, Query is given an index $\ell \in [n]$ and outputs a vector $\mathbf{r} = (r_1, \ldots, r_k) \in [M]^k$, and Dec is given $c_{\mathbf{r}} = (c_{r_1}, \ldots, c_{r_k}) \in \Sigma^k$ and outputs a symbol in Σ.
- **Local decodability.** For every message $\mathsf{msg} \in \Sigma^n$, and every index $\ell \in [n]$,
$$\Pr\left[\mathbf{r} \leftarrow \mathsf{Query}\left(\ell\right) \; : \; \mathsf{Dec}\left(\mathsf{Enc}\left(\mathsf{msg}\right)_{\mathbf{r}}\right) = \mathsf{msg}_{\ell}\right] = 1.$$
- **Smoothness.** For every index $\ell \in [n]$, every query in the output of $\mathsf{Query}\left(\ell\right)$ is distributed uniformly at random over $[M]$.

To simplify notations, when $\Sigma = \{0, 1\}$ we omit it from the notation.

Remark on Smooth LDCs for Block Messages. We will use smooth LDCs for messages consisting of *blocks* $\{0,1\}^{\mathsf{B}}$ of bits (for some block size $\mathsf{B} \in \mathbb{N}$), whose existence is implied by the existence of smooth LDCs over $\{0, 1\}$. Indeed, given a (k, n, M)-smooth LDC $(\mathsf{Enc}, \mathsf{Query}, \mathsf{Dec})$, one can obtain a $(k, n, M)_{\{0,1\}^{\mathsf{B}}}$-smooth LDC $\left(\mathsf{Enc}', \mathsf{Query}', \mathsf{Dec}'\right)$ by "interpreting" the message and codeword as B individual words, where the j'th word consists of the j'th bit in all blocks. Concretely, Enc' on input a message $\left(\mathsf{msg}^1, \ldots, \mathsf{msg}^n\right) \in \left(\{0,1\}^{\mathsf{B}}\right)^n$, computes $y_j^1 \ldots y_j^M = \mathsf{Enc}\left(\mathsf{msg}_j^1, \ldots, \mathsf{msg}_j^n\right)$ for every $1 \leq j \leq \mathsf{B}$, sets $c^i = y_1^i \ldots y_{\mathsf{B}}^i$, and outputs $c = \left(c^1, \ldots, c^M\right)$. Query' operates exactly as Query does. Dec', on input $c^{r_1}, \ldots, c^{r_k} \in \{0,1\}^{\mathsf{B}}$, computes $z_j = \mathsf{Dec}\left(c_j^{r_1}, \ldots, c_j^{r_k}\right)$ for every $1 \leq j \leq \mathsf{B}$, and outputs $z_1 \ldots z_{\mathsf{B}}$.

2.2 Oblivious-Access Sort Algorithms

Our construction employ an Oblivious-Access Sort algorithm [BN16] which is, roughly, a RAM program that sorts an input database, such that the access patterns of the algorithm on any two input databases of equal size are statistically close. Thus, oblivious-access sort is the "RAM version" of boolean sorting circuits. (Informally, a boolean sorting circuit is a boolean circuit ensemble $\left\{C\left(n, \mathsf{B}\right)\right\}_{n,\mathsf{B}}$ such that each $C\left(n, \mathsf{B}\right)$ takes as input n size-B tagged blocks, and outputs the blocks in sorted order according to their tags.)

Definition 2.3 (Oblivious-Access Sort Algorithm, [BN16])**.** An *Oblivious-Access Sort* algorithm for input size n and block size B, with complexity $\mathsf{comp}\left(n,\mathsf{B}\right)$, is a (possibly randomized) protocol Sort between a server S and a client C, with the following properties:

- **Operation:** The input to the server consists of n tagged blocks which are represented as length-B bit strings (the tag is a substring of the block).[1] The client and server can perform local bit operations on the input blocks, but can only communication full blocks.
- **Efficiency:** The communication and server computation overhead of Sort is $\mathsf{comp}\left(n,\mathsf{B}\right)$.
- **Correctness:** With overwhelming probability in n, at the end of the protocol the database contains the n input blocks in sorted order according to their tags.
- **Oblivious Access:** For a database DB consisting of n blocks of size B, let $\mathsf{vAP}_{n,\mathsf{B}}\left(\mathsf{Sort},\mathsf{DB}\right)$ (for *visible access pattern*) denote the random variable consisting of the list of database addresses that S accesses in a random execution of the protocol Sort on database DB. Then for every pair DB,DB' of databases with n size-B blocks, $\mathsf{vAP}_{n,\mathsf{B}}\left(\mathsf{Sort},\mathsf{DB}\right)\approx^{s}\mathsf{vAP}_{n,\mathsf{B}}\left(\mathsf{Sort},\mathsf{DB}'\right)$, where $\approx^{s}$ denotes $\mathsf{negl}\left(n\right)$ statistical distance.

Boyle and Naor [BN16] show that the existence of sorting circuits implies the existence of oblivious-access sort algorithms with related parameters:

Theorem 2.4 (Oblivious-access sort from sorting circuits, [BN16])**.** *If there exist boolean sorting circuits* $\left\{C\left(n,\mathsf{B}\right)\right\}_{n,\mathsf{B}}$ *of size* $s\left(n,\mathsf{B}\right)$*, then there exists an oblivious-access sort algorithm for* n *distinct elements with* $O\left(1\right)$ *client storage,* $O\left(n\cdot\log\mathsf{B}+s\left(\frac{2n}{\mathsf{B}},\mathsf{B}\right)\right)$ *complexity, and* $e^{-n^{\Omega(1)}}$ *probability of error.*

Remark on the Existence of Small Oblivious-Access Sort Algorithms. We note that for blocks of poly-logarithmic size $\mathsf{B}=\operatorname{poly}\log n$, the existence of sorting circuits of size $s\left(n,\mathsf{B}\right)=O\left(n\cdot\mathsf{B}\cdot\log\log n\right)$ guarantees (through Theorem 2.4) the existence of oblivious-access sort algorithms of size $O\left(n\cdot\log\log n\right)$.

Remark on the Relation to Sorting Networks. The related notion of a *sorting network* has been extensively used in ORAM constructions. Similar to oblivious-access sort algorithms, sorting networks sort n size-B blocks in an oblivious manner. (More specifically, a sorting network is *data oblivious*, namely its memory accesses are independent of the input.) However, unlike oblivious-access sort algorithms, and boolean sorting circuits, which can operate *locally* on the bits in the bit representation of the input blocks, a sorting network consist of a single type of *compare-exchange* gate which takes a pair of blocks as inputs, and outputs them in sorted order. We note that a simple information-theoretic lower bound of $\Omega\left(n\log n\right)$ on the network size is known for sorting networks (as well as matching upper bounds, e.g. [AKS83, Goo14]), whereas no such bound is known for boolean sorting circuits or oblivious-access sorting algorithms.

2.3 Oblivious RAM (ORAM)

Oblivious RAMs were introduced by Goldreich and Ostrovskey [Gol87, Ost90, GO96]. To define oblivious RAMs, we will need the following notation of an *access pattern*.

[1] In [BN16], the blocks consist solely of the tag, but the protocol is usually run when tags are concatenated with memory blocks (which are carried as a "payload", and the complexity increases accordingly). We choose to explicitly include the data portion in the block.

Notation 2.5 (Access pattern)**.** A length-q *access pattern* Q consists of a list $(\mathsf{op}_l, \mathsf{val}_l, \mathsf{addr}_l)_{1 \le l \le q}$ of instructions, where instruction $(\mathsf{op}_l, \mathsf{val}_l, \mathsf{addr}_l)$ denotes that the client performs operation $\mathsf{op}_l \in \{\mathsf{read}, \mathsf{write}\}$ at address addr_l with value val_l (which, if $\mathsf{op}_l = \mathsf{read}$, is $\perp$).

Definition 2.6 (Oblivious RAM (ORAM))**.** An *Oblivious RAM (ORAM)* scheme with block size B consists of procedures $(\mathsf{Setup}, \mathsf{Read}, \mathsf{Write})$, with the following syntax:

- $\mathsf{Setup}(1^\lambda, \mathsf{DB})$ is a function that takes as input a security parameter λ, and a database $\mathsf{DB} \in \left(\{0,1\}^{\mathsf{B}}\right)^n$, and outputs an initial server state st_S and a client key ck. We require that the size of the client key $|\mathsf{ck}|$ be bounded by some fixed polynomial in the security parameter λ, independent of $|\mathsf{DB}|$.

- Read is a protocol between the server S and the client C. The client holds as input an address $\mathsf{addr} \in [n]$ and the client key ck, and the server holds its current state st_S. The output of the protocol is a value val to the client, and an updated server state st'_S.

- Write is a protocol between the server S and the client C. The client holds as input an address $\mathsf{addr} \in [n]$, a value v, and the client key ck, and the server holds its current state st_S. The output of the protocol is an updated server state st'_S.

We require the following correctness and security properties.

- **Correctness:** In any execution of the Setup algorithm followed by a sequence of Read and Write protocols between the client and the server, where the Write protocols were executed with a sequence V of values, the output of the client in every execution of the Read protocol is with overwhelming probability the value he would have read from the database in the corresponding read operation, if the prefix of V performed before the Read protocol was performed directly on the database.

- **Security:** For a database DB, and an access pattern Q, let $\mathsf{vAP}\,(\mathsf{DB}, Q)$ denote the random variable consisting of the list of addresses accessed in the ORAM when the Setup algorithm is executed on DB, followed by the execution of a sequence of Read and Write protocols according to Q. Then for every pair $\mathsf{DB}^0, \mathsf{DB}^1 \in \left(\{0,1\}^{\mathsf{B}}\right)^n$ of databases, and any pair $Q^0 = \left(\mathsf{op}_l, \mathsf{val}^0_l, \mathsf{addr}^0_l\right)_{1 \le l \le q}, Q^1 = \left(\mathsf{op}_l, \mathsf{val}^1_l, \mathsf{addr}^1_l\right)_{1 \le l \le q}$ of access patterns of length $q = \mathrm{poly}\,(\lambda)$, $\mathsf{vAP}\left(\mathsf{DB}^0, Q^0\right) \approx^s \mathsf{vAP}\left(\mathsf{DB}^1, Q^1\right)$, where $\approx^s$ denotes $\mathsf{negl}\,(\lambda)$ statistical distance.

Definition 2.6 does not explicitly specify who runs the Setup procedure. It can be performed by the client, who then sends the server state st_S to the server S, or (to save on client computation) can be delegated to a trusted third party.

Remark 2.7. Notice that Definition 2.6 does not hide whether the performed operation is a read or a write, whereas an ORAM scheme is usually defined to hide this information. However, any such scheme can be generically made to hide the identity of operations by always performing both a read and a write. (Specifically, in a write operation, one first performs a dummy read; in a read operation, one writes back the value that was read.) Revealing the identity of operations allows us to obtain more fine-grained overheads.

Remark on Hiding Database Contents. The security property of Definition 2.6 implicitly assumes that the server does not see the *contents* of the memory (i.e., the ORAM memory): if the server is allowed to see the memory contents, he might be able to learn some non-trivial information regarding the access pattern, and thus violate the security property. Hiding memory contents from the server can be achieved by encrypting the database entries, but security will then only hold against *computationally-bounded* servers, and so we choose to define security with the implicit assumption that the server does not see the memory contents.

We will also consider the more restricted notion of a *Read-Only (RO) ORAM* scheme which, roughly, is an ORAM scheme that supports only read operations.

Definition 2.8 (Read-Only Oblivious RAM (RO-ORAM))**.** A *Read-Only Oblivious RAM (RO-ORAM)* scheme consists of procedures $(\mathsf{Setup}, \mathsf{Read})$ with the same syntax as in Definition 2.6, in which correctness holds for any sequence of Read protocols between the client and the server, and security holds for any pair of access patterns R^0, R^1 that contain only read operations.

3 Read-Only ORAM from Oblivious-Access Sort and Smooth LDCs

In this section we construct a Read-Only Oblivious RAM (RO-ORAM) scheme from oblivious-access sort algorithms and smooth LDCs. Concretely, we prove the following:

Theorem 3.1. *Suppose there exist:*

- (k, n, M)*-smooth LDCs with* $M = \mathrm{poly}(n)$ *in which the decoder stores* $\mathsf{space}(k)$ *blocks.*
- *An oblivious-access sort algorithm* Sort *with complexity* $s(n, \mathsf{B})$ *for input size* n *and block size* B*.*

Then there exists a RO-ORAM scheme for databases of size n *and blocks of size* $\mathsf{B} = \Omega\left(\lambda \cdot k^2 \cdot \log^3(kn) \log^7 \log(kn)\right)$ *with* $O(\mathsf{space}(k))$ *client storage, and* $k + \frac{2k^2}{M} \cdot s(M, \mathsf{B}) + O(1)$ *computation and communication overhead.*

Theorem 1.1 now follows from Theorem 3.1 (using also Theorem 2.4) for an appropriate instantiation of the sorting algorithm and LDC.

Corollary 3.2 (RO-ORAM, "dream" parameters; formal statement of Theorem 1.1)**.** *Suppose there exist:*

- (k, n, M)*-smooth LDCs with* $k = O(1)$ *and* $M = \mathrm{poly}(n)$*.*
- *Boolean sorting circuits* $\{C(n, \mathsf{B})\}_{n,\mathsf{B}}$ *of size* $s(n, \mathsf{B}) = O(n \cdot \mathsf{B})$ *for input size* n *and block size* B*.*

Then there exists a RO-ORAM scheme for databases of size n *and blocks of size* $\Omega\left(\lambda \cdot \log^4 n\right)$ *with* $O(1)$ *client storage, and* $O(\log\log n)$ *computation and communication overhead.*

We also instantiate our construction with sorting algorithms and LDCs with more "conservative" parameters, to obtain the following corollary.

Corollary 3.3 (RO-ORAM, milder parameters)**.** *Suppose there exist:*

- (k, n, M)*-smooth LDCs with* $k = \mathrm{poly}\log\log n$*,* $M = \mathrm{poly}(n)$*, and* $\mathsf{space}(k) = \tilde{O}(k)$*.*
- *Boolean sorting circuits* $\{C(n, \mathsf{B})\}_{n,\mathsf{B}}$ *of size* $s(n, \mathsf{B}) \in o\left(\frac{n \cdot \mathsf{B} \cdot \log n}{k^2}\right)$ *for input size* n *and block size* B*.*

Then there exists a RO-ORAM scheme for databases of size n *and blocks of size* $\Omega\left(\lambda \cdot \log^4 n\right)$ *with* $\mathrm{poly}\log\log n$ *client storage, and* $o(\log n)$ *computation and communication overhead.*

Construction Overview. As outlined in the introduction, our construction uses a (k, n, M)-smooth LDC. The server stores k codeword copies, each permuted using a unique uniformly random permutation. To read block j from the ORAM, the client runs the LDC decoder until the decoder generates a set of *fresh* decoding queries (i.e., a set $q_1, \ldots, q_k$ of queries such that for every $1 \leq i \leq k$, q_i was not issued before as the i'th query), and sends these queries to the server. The server uses the i'th permuted codeword copy to answer the i'th decoding query. The metadata regarding which decoding queries are fresh, as well as the description of the permutations, are stored on the server using a (polylogarithmic-overhead) ORAM scheme, which the client accesses to determine whether the decoder queries are fresh, and to permute them according to the random permutations.

The execution is divided into "epochs" consisting of $O(M/k)$ read operations. When an epoch ends, the client "refreshes" the permuted codeword copies by picking k fresh, random permutations, and running an oblivious-access sort algorithm with the server to permute the codeword copies stored on the server according to the new permutations. The description of the new permutations is stored in the metadata ORAM (the client also resets the bits indicating which decoding queries are fresh). The refreshing operations are spread-out across the $O(M/k)$ read operations of the epoch. The resultant increase in complexity depends on k (which determines the epoch length, i.e., the frequency in which refreshing is needed), and on the complexity of the oblivious-access sort algorithm.

Construction 3.4 (RO-ORAM from Oblivious-Access Sort and Smooth LDCs)**.** The scheme uses the following building blocks:

- A $(k, n, M)_{\{0,1\}^{\mathsf{B}}}$-smooth LDC $(\mathsf{Enc}_{\mathsf{LDC}}, \mathsf{Query}_{\mathsf{LDC}}, \mathsf{Dec}_{\mathsf{LDC}})$.
- An oblivious-access sort algorithm Sort.
- An ORAM scheme $(\mathsf{Setup}_{\text{in}}, \mathsf{Read}_{\text{in}}, \mathsf{Write}_{\text{in}})$.

The scheme consists of the following procedures:

- $\mathbf{Setup}(1^\lambda, \mathbf{DB})$**:** Recall that λ denotes the security parameter, and $\mathsf{DB} \in \left(\{0,1\}^{\mathsf{B}}\right)^n$. Instantiate the LDC with message size n over alphabet $\Sigma = \{0,1\}^{\mathsf{B}}$, and let k be the corresponding number of queries, and M be the corresponding codeword size. Proceed as follows.
  - Counter initialization. Initializes a *step counter* $\mathsf{count} = 0$.
  - Data storage generation.
    - Generate the codeword $\widetilde{\mathsf{DB}} = \mathsf{Enc}_{\mathsf{LDC}}(\mathsf{DB})$ with $\widetilde{\mathsf{DB}} \in \Sigma^M$.
    - For every $1 \leq i \leq k$:
      - Generate a random permutation $P^i : [M] \rightarrow [M]$.
      - Let $\widetilde{\mathsf{DB}}^i \in \Sigma^M$ be a permuted database which satisfies $\widetilde{\mathsf{DB}}^i_{P^i(j)} = \widetilde{\mathsf{DB}}_j$ for all $j \in [M]$.
  - Metadata storage generation.
    - For every $1 \leq i \leq k$:
      - Initialize a length-M bit-array $\mathsf{Queried}^i$ to $\vec{0}$.
      - Initialize a length-M array Perm^i over $\{0,1\}^{\log M}$ such that $\mathsf{Perm}^i(j) = P^i(j)$.
    - Let mDB denote the database obtained by concatenating $\mathsf{Queried}^1, \ldots, \mathsf{Queried}^k$ and $\mathsf{Perm}^1, \ldots, \mathsf{Perm}^k$. Run $(\mathsf{ck}_m, \mathsf{st}_m) \leftarrow \mathsf{Setup}_{\text{in}}\left(1^\lambda, \mathsf{mDB}\right)$ to obtain the client key and server state for the metadata ORAM.

  - Output. The long-term client key $\mathsf{ck} = (\mathsf{count}, \mathsf{ck}_m)$ consists of the step counter, and the client key for the metadata ORAM. The server state $\mathsf{st}_S = \left(\left\{\widetilde{\mathsf{DB}}^i \ : \ i \in [k]\right\}, \mathsf{st}_m\right)$ contains the k permuted databases, and the server state for the metadata ORAM.

- **The Read protocol.** To read the database entry at location $\mathsf{addr} \in [n]$ from the server S, the client C with key $\mathsf{ck} = (\mathsf{count}, \mathsf{ck}_m)$ operates as follows, where in all executions of the $\mathsf{Read}_{\text{in}}$ or $\mathsf{Write}_{\text{in}}$ protocols on mDB S plays the role of the server with state st_m and C plays the role of the client with key ck_m.
  - Generating decoder queries. Repeat the following λ times:
    - Run $(q_1, \ldots, q_k) \leftarrow \mathsf{Query}_{\mathsf{LDC}}(\mathsf{addr})$ to obtain decoding queries.
    - For every $1 \leq i \leq k$, run the $\mathsf{Read}_{\text{in}}$ protocol to read $\mathsf{Queried}^i[q_i]$. We say that q_i is *fresh* if $\mathsf{Queried}^i[q_i] = 0$.
    - Let $(\hat{q}_1, \ldots, \hat{q}_k)$ denote the decoding queries in the first iteration in which all queries were fresh. (If no such iteration exists, set $(\hat{q}_1, \ldots, \hat{q}_k)$ to be the decoding queries generated in the last iteration.)
  - Permuting queries. For every $1 \leq i \leq k$, run the $\mathsf{Read}_{\text{in}}$ protocol to read $\mathsf{Perm}^i[\hat{q}_i]$. Let q'_i denote the value that $\mathsf{Read}_{\text{in}}$ outputs to the client.
  - Decoding database entry. Send $(q'_1, \ldots, q'_k)$ to the server S, obtain the answers $\left(\widetilde{\mathsf{DB}}^1_{q'_1}, \ldots, \widetilde{\mathsf{DB}}^k_{q'_k}\right)$, and set the client output to $\mathsf{Dec}_{\mathsf{LDC}}\left(\widetilde{\mathsf{DB}}^1_{q'_1}, \ldots, \widetilde{\mathsf{DB}}^k_{q'_k}\right)$.
  - Updating counter and server state. Let $\ell = \frac{M}{2k}$.
    - If $\mathsf{count} < \ell - 1$, then update $\mathsf{count} := \mathsf{count} + 1$, and for every $1 \leq i \leq k$, run the $\mathsf{Write}_{\text{in}}$ protocol to write "1" to $\mathsf{Queried}^i[\hat{q}_i]$.
    - Otherwise, update $\mathsf{count} := 0$, and for every $1 \leq i \leq k$:
      - Run the $\mathsf{Write}_{\text{in}}$ protocol to write $\vec{0}$ to $\mathsf{Queried}^i$.
      - Replace P^i with a fresh random permutation on $[M]$ by running the Fisher-Yates shuffle algorithm (as presented by Durstenfeld [Dur64]) on Perm^i, using the $\mathsf{Read}_{\text{in}}$ and $\mathsf{Write}_{\text{in}}$ protocols.
      - Use Sort to sort $\widetilde{\mathsf{DB}}^i$ according to the new permutation P^i (each block consists of a codeword symbol, and the index in the codeword which is used as the tag of the block).

      If the complexity of these three steps is c_{epoch}, then the server and client perform c_{epoch}/ℓ steps of this computation in each protocol execution so that it is completed by the end of the epoch.

We prove the following claims about Construction 3.4.

Claim 3.5 (ORAM security)**.** *Assuming the security of all of the building blocks, Construction 3.4 is a secure RO-ORAM scheme.*

Claim 3.6 (ORAM complexity)**.** *Assume that:*

- *The database* DB *and the metadata ORAM have block size* $\mathsf{B}, \mathsf{mB} \geq \log M$*, respectively.*

- *The metadata ORAM has computation and communication overhead* $\mathsf{Ovh}(N)$ *for databases of size* N*.*

- *The oblivious-access sort algorithm has complexity* $\mathsf{comp}\left(n, \mathsf{B}\right)$ *when operating on databases consisting of* n *size-*B *blocks.*

Then each execution of the Read *protocol of Construction 3.4 consists of*

$$\mathsf{mB} \cdot \mathsf{Ovh}\left(\frac{k \cdot (M + M \log M)}{\mathsf{mB}}\right) \cdot O\left(k\lambda + k^2\right) + \mathsf{B} \cdot \left(k + \frac{2k^2}{M} \cdot \mathsf{comp}\left(M, \mathsf{B}\right)\right)$$

bit operations.

Claims Imply Theorem. To prove Theorem 3.1, we instantiate the metadata ORAM of Construction 3.4 with the following variant of path ORAM [SvDS+13]:

Theorem 3.7 (Statistical ORAM with polylog overhead, implicit in [SvDS+13])**.** *There exists a statistical ORAM scheme for databases consisting of* N *blocks of size* $\mathsf{mB} = \log^2 N \log\log N$ *in which the client stores* $O\left(\log N \log\log N\right)$ *blocks, and the overhead is* $O\left(\log N\right)$*.*

Moreover, initializing the scheme requires $O\left(N \cdot \mathsf{mB}\right)$ *bit operations, and the server stores* $O\left(N\right)$ *blocks.*

Proof of Theorem 3.1. Security follows directly from Claim 3.5 since (as noted in Section 2.1) the existence of a (k, n, M)-smooth LDC implies the existence of a $(k, n, M)_{\{0,1\}^{\mathsf{B}}}$-smooth LDC.

As for the complexity of the construction, let $N_m = k\left(M + M \log M\right)$ denote the size (in bits) of the metadata ORAM. Substituting $\mathsf{mB} = \log^2 N_m \log\log N_m$, and $\mathsf{Ovh}\left(N\right) = O\left(\log N\right)$ (according to Theorem 3.7), Claim 3.6 guarantees that every execution of the Read protocol consists of

$$\log^2 N_m \log\log N_m \cdot O\left(\log \frac{N_m}{\log^2 N_m \log\log N_m}\right) \cdot O\left(k\lambda + k^2\right) + \mathsf{B} \cdot \left(k + \frac{2k^2}{M} \cdot \mathsf{comp}\left(M, \mathsf{B}\right)\right)$$

bit operations. The first summand can be upped bounded by

$$\log^2\left(kM\right)\log^3\log\left(kM\right) \cdot O\left(\log\left(kM\right) \cdot \lambda \cdot k^2\right) \leq \log^3\left(kM\right)\log^3\log\left(kM\right) \cdot O\left(\lambda \cdot k^2\right).$$

For $\mathsf{B} = \Omega\left(\lambda \cdot k^2 \cdot \log^3\left(kn\right)\log^7\log\left(kn\right)\right)$ (as in the theorem statement) with a sufficiently large constant in the $\Omega\left(\cdot\right)$ notation, and since $M = \mathrm{poly}\left(n\right)$, this corresponds to a constant number of read and write operations on size-B blocks, and so the overall computation and communication complexity is $\mathsf{B} \cdot \left(k + \frac{2k^2}{M} \cdot s\left(M, \mathsf{B}\right) + O\left(1\right)\right)$ bit operations, so the overhead of the construction is $k + \frac{2k^2}{M} \cdot s\left(M, \mathsf{B}\right) + O\left(1\right)$.

Finally, regarding client storage, emulating the LDC decoder requires storing $\mathsf{space}\left(k\right)$ size-B blocks. Operations on mDB require (by Theorem 3.7) storing $O\left(\log N_m \log\log N_m\right)$ size-mB blocks which corresponds to a constant number of size-B blocks. □

Security Analysis: Proof of Claim 3.5. The proof of Claim 3.5 will use the next lemma, which states that with overwhelming probability, every Read protocol execution uses fresh decoding queries. This follows from the smoothness of the underlying LDC.

Lemma 3.8. *Let* $k, M \in \mathbb{N}$*, and let* $X = \left(X_1, \ldots, X_k\right)$ *be a random variable over* $\left[M\right]^k$ *such that for every* $1 \leq i \leq k$*,* X_i *is uniformly distributed over* $\left[M\right]$*. Let* $S_1, \ldots, S_k \subseteq \left[M\right]$ *be subsets of size at most* ℓ*. Then in* l *independent samples according to* X*, with probability at least* $1 - \left(k \cdot \frac{\ell}{M}\right)^l$*, there exists a sample* $\left(x_1, \ldots, x_k\right)$ *such that* $x_i \notin S_i$ *for every* $1 \leq i \leq k$*.*

In particular, if $\ell = \frac{M}{2k}$ *and* $l = \Omega\left(\lambda\right)$ *then except with probability* $\mathsf{negl}\left(\lambda\right)$*, there exists a sample* $\left(x_1, \ldots, x_k\right)$ *such that* $x_i \notin S_i$ *for every* $1 \leq i \leq k$*.*

Proof. Consider a sample $(x_1, \ldots, x_k)$ according to X. Since each X_i is uniformly distributed over $[M]$, then $\Pr[x_i \in S_i] = \frac{\ell}{M}$, so by the union bound, $\Pr[\exists i : x_i \in S_i] \leq k \cdot \frac{\ell}{M}$. Since the l samples are independent, the probability that no such sample exists is $(\Pr[\text{in a single sample, } \exists i : x_i \in S_i])^l \leq \left(k \cdot \frac{\ell}{M}\right)^l$. For the "in particular" part, notice that for $\ell = \frac{M}{2k}$ and $l = \Omega(\lambda)$, $1 - \left(k \cdot \frac{\ell}{M}\right)^l = 1 - 2^{-\Omega(\lambda)}$. □

We are now ready to prove Claim 3.5.

Proof of Claim 3.5. The correctness of the scheme follows directly from the correctness of the underling LDC. We now argue security. Let $\mathsf{DB}^0, \mathsf{DB}^1$ be two databases consisting of n size-B blocks, and let R^0, R^1 be two sequences of read operations of length $q = \mathrm{poly}(\lambda)$. We proceed via a sequence of hybrids. We assume that in each read operation, at least one iteration in the Read protocol succeeded in generating fresh decoder queries, and condition all hybrids on this event. This is without loss of generality since by Lemma 3.8, this happens with overwhelming probability.

$\mathcal{H}_\mathbf{0}^\mathbf{b}$: Hybrid $\mathcal{H}_0^b$ is the visible access pattern $\mathsf{vAP}\left(\mathsf{DB}^b, R^b\right)$ in an execution of read sequence R^b on the RO-ORAM generated for database DB^b.

$\mathcal{H}_\mathbf{1}^\mathbf{b}$: In hybrid $\mathcal{H}_1^b$, for every $1 \leq i \leq k$, we replace the values of $\mathsf{Queried}^i$ and Perm^i with dummy values of (e.g.,) the all-0 string. Moreover, we replace all read and write accesses to the metadata mDB with dummy operations that (e.g.,) read and write the all-0 string to the first location in the metadata. (We note that the accesses to the permuted databases remain unchanged, where each access consists of fresh decoding queries, permuted according to $P^1, \ldots, P^k$.)

Hybrids $\mathcal{H}_0^b$ and $\mathcal{H}_1^b$ are statistically indistinguishable by the security of the metadata ORAM.

$\mathcal{H}_\mathbf{2}^\mathbf{b}$: In hybrid $\mathcal{H}_2^b$, for every $1 \leq i \leq k$, and every epoch j, we replace the permutation on which the oblivious-access sort algorithm Sort is applied, with a dummy permutation (e.g., the identity). (As in $\mathcal{H}_1^b$, the accesses to the codeword copies remain unchanged, and in particular the "right" permutations are used in all epochs.)

Hybrids $\mathcal{H}_1^b$ and $\mathcal{H}_2^b$ are statistically indistinguishable by the obliviousness property of the oblivious-access sort algorithm.

$\mathcal{H}_\mathbf{3}^\mathbf{b}$: In hybrid $\mathcal{H}_3^b$, for every $1 \leq i \leq k$, we replace the queries to the i'th permuted database with queries that are uniformly random subject to the constraint that they are all distinct.

Hybrids $\mathcal{H}_2^b$ and $\mathcal{H}_3^b$ are statistically indistinguishable since by our assumption all the queries sent to the database copies are fresh, and they are permuted using random permutations. (Notice that $\mathcal{H}_2^b, \mathcal{H}_3^b$ contain no additional information regarding these permutations.)

We conclude the proof by noting that $\mathcal{H}_3^0 \equiv \mathcal{H}_3^1$ since neither depend on $\mathsf{DB}^0, \mathsf{DB}^1, R^0$ or R^1. □

Complexity Analysis: Proof of Claim 3.6. We now analyze the complexity of Construction 3.4, proving Claim 3.6. Notice that since $\mathsf{mB} \geq \log M$, an image of any random permutation $P^i : [M] \to [M]$ is contained in a single block of mDB. Notice also that the metadata mDB consists of $k \cdot (M + M \log M)$ bits, and let $N_m := \frac{k \cdot (M + M \log M)}{\mathsf{mB}}$ denote its size in size-mB blocks.

Proof of Claim 3.6. Every execution of the Read protocol consists of the following operations:

- Reading $k \cdot \lambda$ bits from mDB to check if the decoding queries in each of the λ iterations are fresh. In the worst case, reading each bit requires reading a different block from mDB, for a total of $\mathsf{mB} \cdot k\lambda \cdot \mathsf{Ovh}(N_m)$ bit operations.

- Reading k images from $\mathsf{Perm}^1, \ldots, \mathsf{Perm}^k$ to permute the chosen decoding queries. This requires reading k blocks from mDB, for a total of $\mathsf{mB} \cdot k \cdot \mathsf{Ovh}\left(N_m\right)$ bit operations.

- Reading k blocks from the permuted databases $\widetilde{\mathsf{DB}}^1, \ldots, \widetilde{\mathsf{DB}}^k$ to answer the decoder queries, for a total of $\mathsf{B} \cdot k$ bit operations.

- Writing k bits to mDB to update the values $\mathsf{Queried}^i\left[\hat{q}^i\right], 1 \leq i \leq k$, to 1, in total performing $\mathsf{mB} \cdot k \cdot \mathsf{Ovh}\left(N_m\right)$ bit operations. (This operation is only performed when $\mathsf{count} < \ell - 1$, but counting it in *every* Read execution will not increase the overall asymptotic complexity.)

In total, these operations require performing $\mathsf{mB} \cdot \mathsf{Ovh}\left(N_m\right) \cdot O\left(k\lambda\right) + \mathsf{B} \cdot k$ bit operations.

In addition, every Read execution performs its "share" of the operations needed to update the server state at the end of the epoch. More specifically, it performs a $\frac{1}{\ell} = \frac{2k}{M}$-fraction of the following operations:

- Writing $k \cdot \frac{M}{\mathsf{mB}}$ blocks to mDB to reset all entries of $\mathsf{Queried}^i, 1 \leq i \leq k$, as well as reading and writing $k \cdot 2M$ blocks to mDB to update the entries of $\mathsf{Perm}^i, 1 \leq i \leq k$ with the images of the new permutations, using the Fisher-Yates shuffle. In total, this requires performing $\mathsf{mB} \cdot k \cdot M \cdot \left(\frac{1}{\mathsf{mB}} + 4\right) \cdot \mathsf{Ovh}\left(N_m\right)$ bit operations.

- Running k executions of Sort on a length-M database consisting of blocks of size B to re-permute the codeword copies, which requires performing $k \cdot \mathsf{comp}\left(M, \mathsf{B}\right)$ bit operations.

So these update operations add a total of $\mathsf{mB} \cdot \mathsf{Ovh}\left(N_m\right) \cdot O\left(k^2\right) + \mathsf{B} \cdot \frac{2k^2}{M} \cdot \mathsf{comp}\left(M, \mathsf{B}\right)$ bit operations to each execution of the Read protocol.

In summary, reading a single size-B block from DB requires performing $\mathsf{mB} \cdot \mathsf{Ovh}\left(\frac{k \cdot (M + M \log M)}{\mathsf{mB}}\right) \cdot O\left(k \cdot \lambda + k^2\right) + \mathsf{B} \cdot \left(k + \frac{2k^2}{M} \cdot \mathsf{comp}\left(M, \mathsf{B}\right)\right)$ bit operations. □

3.1 Read-Only ORAM with Oblivious Setup

In this section we generalize the notion of a RO-ORAM scheme to allow the client and server to jointly run the ORAM Setup algorithm, when the database itself is already stored at the server. We call this primitive a *RO-ORAM scheme with oblivious setup*,[2] and show that the RO-ORAM scheme of Construction 3.4 extends to this setting. The oblivious setup protocol for Construction 3.4 will be used in the next section to construct an ORAM scheme supporting writes with low read overhead. The fact that the database is already stored at the server will help reduce the client complexity compared to the Setup algorithm of Construction 3.4.

At a high level, a RO-ORAM scheme with oblivious setup is a RO-ORAM scheme $(\mathsf{Setup}, \mathsf{Read})$ associated with an additional protocol $\mathsf{OblSetup}$ which allows the client and server to jointly execute the Setup algorithm, where the execution is oblivious in the sense that the scheme remains secure when the RO-ORAM is generated using $\mathsf{OblSetup}$ instead of Setup. Formally,

Definition 3.9 (RO-ORAM with oblivious setup)**.** A RO-ORAM scheme *with oblivious setup* is a RO-ORAM scheme $(\mathsf{Setup}, \mathsf{Read})$, associated with an additional $\mathsf{OblSetup}$ protocol between the client C and the server S, where:

- The server holds as input a database $\mathsf{DB} \in \left(\{0,1\}^{\mathsf{B}}\right)^n$, and both parties hold 1^λ as input.

[2] We could have similarly defined this notion as an extension of ORAM schemes (that support write operations), but since we only use this property for RO-ORAM schemes, we choose to define it for this (more restricted) setting.

- The correctness property of Definition 2.8 holds when the RO-ORAM is instantiated with $\mathsf{OblSetup}$ instead of Setup.

- The security property of Definition 2.8 holds when the visible access patterns $\mathsf{vAP}\left(\mathsf{DB}^b, R^b\right), b \in \{0,1\}$ is generated using $\mathsf{OblSetup}$ instead of Setup.

Next, we show that the RO-ORAM scheme of Construction 3.4 has oblivious setup. The oblivious setup protocol relies on the building blocks of Construction 3.4, and requires the additional assumption that the LDC encoding procedure is oblivious, in the sense that the its access pattern is independent of the encoded message. (This assumption is satisfied by, e.g., linear LDCs.) Notice that any LDC with a sufficiently small encoding circuit also has oblivious encoding, since the overhead incurred by a naive translation of the circuit into a RAM program is proportional to the word size.

The high-level idea is conceptually simple. First, the server can generate the codeword copies on his own, because he already stores the database, and the LDC encoding procedure is oblivious. Then, the client can use an "empty" metadata (initialized to $\vec{0}$) to generate his keys for the metadata ORAM, and update its contents by running the Write protocol of the metadata ORAM together with the server. Finally, the codeword copies can be obliviously permuted using the oblivious-access sort algorithm. This high-level intuition is formalized in the following construction.

Construction 3.10 (Oblivious setup for Construction 3.4)**.** The protocol $\mathsf{OblSetup}$ is run between the client C, and the server S. Both parties have 1^λ as input, and the server also stores a database $\mathsf{DB} \in \left(\{0,1\}^{\mathsf{B}}\right)^n$. We use the notation of Construction 3.4 (e.g., for the LDC parameters). The protocol is executed as follows:

- The client initializes the step counter $\mathsf{count} = 0$. Additionally, the client initializes a metadata ORAM for a database containing $\frac{k(M+\log M)}{\mathsf{mB}}$ "dummy" blocks of size mB (e.g., the all-0 block), by running $(\mathsf{ck}_m, \mathsf{st}_m) \leftarrow \mathsf{Setup}_{\mathsf{in}}\left(1^\lambda, \mathsf{mDB}\right)$, and sends st_m to the server.[3]

- The server locally encodes the database $\widetilde{\mathsf{DB}} = \mathsf{Enc}_{\mathsf{LDC}}(\mathsf{DB})$, and stores k copies $\widetilde{\mathsf{DB}}^i, i \in [k]$ of $\widetilde{\mathsf{DB}}$.

- For every $1 \le i \le k$, the client and server:
  - Write $\vec{0}$ to $\mathsf{Queried}^i$, using the $\mathsf{Write}_{\mathsf{in}}$ protocol.
  - Write a fresh random permutation $P^i : [M] \rightarrow [M]$ to Perm^i by running the Fisher-Yates shuffle algorithm (as presented by Durstenfeld [Dur64]) on Perm^i, using the $\mathsf{Read}_{\mathsf{in}}$ and $\mathsf{Write}_{\mathsf{in}}$ protocols.
  - Use Sort to sort $\widetilde{\mathsf{DB}}^i$ according to the new permutation P^i (each block consists of a codeword symbol, and the index in the codeword which is used as the tag of the block).

- The output to the client is $(\mathsf{count}, \mathsf{ck}_m)$, and the output to the server is $\left(\left\{\widetilde{\mathsf{DB}}^i\right\}_{i\in[k]}, \mathsf{st}'_m\right)$, where st'_m denote the updated state of the server in the metadata ORAM at the end of this protocol.

We show that Construction 3.4 is a RO-ORAM scheme with oblivious setup by proving that the protocol of Construction 3.10 satisfies the properties of Definition 3.9:

[3] The client complexity in this step can be reduced if the client and server can *jointly* execute the $\mathsf{Setup}_{\mathsf{in}}$ protocol (where the client generates his keys, and the server generates the data structure of dummy blocks on his own). One example of an ORAM scheme which admits such joint setup is the path ORAM scheme of Theorem 3.7.

Lemma 3.11 (RO-ORAM with oblivious setup)**.** *Assume that Construction 3.4 is instantiated with an LDC whose encoding procedure is oblivious in the sense that for every pair of messages $m, m' \in \left(\{0,1\}^{\mathsf{B}}\right)^n$, the memory accesses during encoding of m, m' are statistically close. Then Construction 3.4, together with the oblivious setup protocol of Construction 3.10, is a RO-ORAM scheme with oblivious setup.*

Proof. The fact that correctness holds when Setup is replaced with $\mathsf{OblSetup}$ follows from the correctness of the underling metadata ORAM scheme, the correctness of the oblivious-access sort algorithm, and the description of $\mathsf{OblSetup}$ and Setup.

As for security, let $\mathsf{DB}^0, \mathsf{DB}^1$ be a pair of databases, let R^0, R^1 be a pair of sequences of read operations of equal length, and let $\mathsf{vAP}^0, \mathsf{vAP}^1$ denote the distributions over visible access patterns generated during the execution of $\mathsf{OblSetup}$ on $\mathsf{DB}^0, \mathsf{DB}^1$ (respectively), followed by executions of the Read protocol to perform the sequences R^0, R^1 (respectively). Since the generation of the random permutations, the metadata mDB, and the update operations performed on the metadata ORAM during $\mathsf{OblSetup}$, are independent of the database contents and the read sequences, we can condition both distributions $\mathsf{vAP}^0, \mathsf{vAP}^1$ on these values. Next, notice that the memory accesses during encoding of $\mathsf{DB}^0, \mathsf{DB}^1$ are statistically close (since the LDC encoding is oblivious by the assumption of the lemma). Consequently, the memory accesses throughout the execution of $\mathsf{OblSetup}$ on $\mathsf{DB}^0, \mathsf{DB}^1$ are statistically close. Finally, we claim that the memory accesses caused by the read sequences are statistically close. To see why this holds, notice that the only difference between the ORAMs after $\mathsf{OblSetup}$ is in the contents of the codeword copies. However, these anyway do not affect the execution of the Read protocol of Construction 3.4 (since the LDC decoder queries are independent of the codeword contents), and so the two sequences of Read protocol executions are statistically close by the security of Construction 3.4. □

Lemma 3.12 (Complexity of oblivious setup)**.** *Assume that:*

- *The database DB and the metadata ORAM have block size $\mathsf{B}, \mathsf{mB} \geq \log M$, respectively.*
- *The metadata ORAM has computation and communication overhead $\mathsf{Ovh}(N)$ for databases of size N, its setup algorithm has $\mathsf{T}_m(N)$ runtime, and can be jointly executed between the client and server, where the client (server) stores s_C (s_S) size-mB blocks.*
- *The oblivious-access sort algorithm has complexity $\mathsf{comp}(n, \mathsf{B})$ when operating on databases consisting of n size-B blocks.*
- *The LDC has query complexity k, codeword length M, and its encoding procedure has runtime $\mathsf{T}_{\mathsf{LDC}}(n)$ for messages of length n.*

Then the $\mathsf{OblSetup}$ protocol of Construction 3.10 performs

$$\lambda + \mathsf{T}_m\left(\frac{k(M+\log M)}{\mathsf{mB}}\right) + \mathsf{T}_{\mathsf{LDC}}(n) + kM + \left(\frac{kM}{\mathsf{mB}} + kM\right) \cdot \mathsf{mB} \cdot \mathsf{Ovh}\left(\frac{k(M+\log M)}{\mathsf{mB}}\right) + k \cdot \mathsf{comp}(n, \mathsf{B})$$

bit operations, and communicates

$$\mathsf{T}_m\left(\frac{k(M+\log M)}{\mathsf{mB}}\right) + k \cdot \mathsf{comp}(n, \mathsf{B})$$

bits. Moreover, the client stores $s_C \cdot \frac{\mathsf{mB}}{\mathsf{B}} + O(\lambda)$ size-B blocks, and the server stores $kM + s_S \cdot \frac{\mathsf{mB}}{\mathsf{B}}$ size-B blocks.

Proof. The storage complexity of the client and server follows directly from the assumptions of the lemma, and the description of the $\mathsf{OblSetup}$ algorithm.

As for the complexity of $\mathsf{OblSetup}$, initializing the metadata ORAM requires performing $\mathsf{T}_m(N)$ bit operations, where $N := \frac{k(M+\log M)}{\mathsf{mB}}$ denotes the size of mDB in size-mB blocks. Moreover, $|\mathsf{st}_m| \leq \mathsf{T}_m(N)$, so (even in the worst case, when the client generates the entire metadata ORAM on his own), at most $\mathsf{T}_m(N)$ bits need to be communicated. Initializing the step counter requires at most λ bit operations (since a λ-bit counter can count any arbitrary polynomial number of steps).

Generating the encoded database requires $\mathsf{T}_{\mathsf{LDC}}(n)$ bit operations, and the length-M codeword is duplicated k times, so generating the codeword copies requires $\mathsf{T}_{\mathsf{LDC}}(n) + kM$ bit operations.

Updating the metadata contents requires writing $\frac{kM}{\mathsf{mB}} + kM$ blocks to mDB (since $\mathsf{Queried}^i$ can be updated by writing $\frac{kM}{\mathsf{mB}}$ $\vec{0}$-blocks, and each image of a permutation P^i has size $\log M \leq \mathsf{mB}$). Therefore, updating the metadata ORAM requires the server to perform $\left(\frac{kM}{\mathsf{mB}} + kM\right) \cdot \mathsf{mB} \cdot \mathsf{Ovh}(N)$ bit operations.

Finally, permuting the codeword copies requires the server to perform $k \cdot \mathsf{comp}(n, \mathsf{B})$ bit operations, and to communicated $k \cdot \mathsf{comp}(n, \mathsf{B})$ bits.

In summary, an execution of $\mathsf{OblSetup}$ requires a total of $\lambda + \mathsf{T}_m(N) + \mathsf{T}_{\mathsf{LDC}}(n) + kM + \left(\frac{kM}{\mathsf{mB}} + kM\right) \cdot \mathsf{mB} \cdot \mathsf{Ovh}(N) + k \cdot \mathsf{comp}(n, \mathsf{B})$ bit operations (performed by either the client or the server), and the communication complexity is $\mathsf{T}_m(N) + k \cdot \mathsf{comp}(n, \mathsf{B})$. □

4 Oblivious RAM Supporting Writes with $o(\log n)$ Read Complexity

In this section we extend the RO-ORAM scheme of Section 3 to support writes, while preserving the overhead of read operations. We instantiate our construction in several parameter regimes, obtaining the following results.

First, by instantiating our construction with "best possible" sorting circuits and LDCs, we prove Theorem 1.2:

Theorem 4.1 (ORAM, "dream" parameters; formal statement of Theorem 1.2)**.** *Assume the existence of LDCs and sorting circuits as in Corollary 3.2, where the LDC has the following additional properties:*

- $M = n^{1+\delta}$ *for some* $\delta \in (0,1)$*.*
- *Encoding is oblivious.*
- *Encoding requires* $M^{1+\gamma}$ *operations over size-*B *blocks, for some* $\gamma \in (0,1)$*.*

Then there exists an ORAM scheme for databases of size n *and blocks of size* $\mathsf{B} = \Omega\left(\lambda \cdot \log^3 n \log^7 \log n\right)$ *with* $O(1)$ *client storage, where* read *operations have* $O(\log\log n)$ *communication and computation overhead, and* write *operations have* $O(n^{\epsilon})$ *communication and computation overhead for any constant* $\epsilon \in (0,1)$ *such that* $\epsilon > \delta + \gamma + \delta\gamma$*.*

Using milder assumptions regarding the parameters of the underlying sorting circuit and LDC, we can prove the following:

Theorem 4.2 (ORAM, milder parameters)**.** *Assume the existence of LDCs and sorting circuits as in Corollary 3.3, where the LDC has the additional properties specified in Theorem 4.1. Then there exists an ORAM scheme for databases of size* n *and blocks of size* $\mathsf{B} = \Omega\left(\lambda \cdot \log^3 n \log^7 \log n\right)$

with $\mathrm{poly}\log\log n$ *client storage, where* read *operations have* $o(\log n)$ *communication and computation overhead, and* write *operations have* $O(n^{\epsilon})$ *communication and computation overhead for any constant* $\epsilon \in (0,1)$ *such that* $\epsilon > \delta + \gamma + \delta\gamma$*.*

Finally, we also obtain a scheme with improved write overhead, by somewhat strengthening the assumptions regarding the LDC.

Theorem 4.3 (ORAM, low write overhead; formal statement of Theorem 1.3)**.** *Assume the existence of LDCs and sorting circuits as in Corollary 3.2, where the LDC has the following additional properties:*

- $M = n^{1+o(1)}$.
- *Encoding is oblivious.*
- *Encoding requires* $M^{1+o(1)}$ *operations over size-*B *blocks.*

Then there exists an ORAM scheme for databases of size n *and blocks of size* $\mathsf{B} = \Omega\left(\lambda \cdot \log^3 n \log^7 \log n\right)$ *with* $O(1)$ *client storage, where* read *operations have* $o(\log n)$ *communication and computation overhead, and* write *operations have* $n^{o(1)}$ *communication and computation overhead.*

Construction Overview. As outlined in Section 1.2.2, the ORAM consists of ℓ levels of increasing size (growing from top to bottom), where initially the database is stored in the lowest level, and all other levels are empty. read operations look for the memory block in all levels, returning the top-most copy of the block, and write operations write the memory block to the top-most level, causing a reshuffle at predefined intervals to prevent levels from overflowing.

Transforming this high-level intuition into an actual scheme requires some adjustments. First, our RO-ORAM scheme[4] was designed for databases given as array data structures (namely, in which blocks can only be accessed by specifying the location of the block in the database), but upper levels are too small to contain the entire database, namely they require RO-ORAM schemes for *map data structure*.[5] To overcome this issue, we associate with each level i an array $\mathcal{DB}^i$ that contains the memory blocks of level i, and is stored in a RO-ORAM $\mathcal{O}^i$. Additionally, we store the metadata regarding which block appears in which array location in a (standard, polylogarithmic-overhead) ORAM $\mathcal{MO}^i$ for map structures. Thus, to look for block j in level i, the client first searches for j in $\mathcal{MO}^i$. If the j'th memory block appears in level i, then $\mathcal{MO}^i$ returns the location t in which it appears in $\mathcal{DB}^i$, and so the client can read the block by performing a read for address t on the RO-ORAM $\mathcal{O}^i$ of the level.

Second, to allow for efficient "reshuffling" of level i (which, in particular, requires a traversal of both $\mathcal{DB}^i$ and $\mathcal{DB}^{i+1}$), we also store $\mathcal{DB}^i$ in every level i. Thus, every level i contains the array $\mathcal{DB}^i$, the metadata ORAM $\mathcal{MO}^i$ which maps blocks to their locations in $\mathcal{DB}^i$, and the RO-ORAM $\mathcal{O}^i$ which stores $\mathcal{DB}^i$. We note that the metadata ORAM is not needed in the lowest level, because the structure will preserve the invariant that $\mathcal{DB}^\ell$ contains all the blocks "in order" (namely, the k'th database block is the k'th block of $\mathcal{DB}^\ell$). This structure is presented in Figure 1.

[4] The construction can use any RO-ORAM scheme, but the read overhead is no smaller than the overhead of the RO-ORAM scheme. Therefore, to obtain $o(\log n)$ overhead, we need to instantiate the ORAM with our RO-ORAM scheme.

[5] We note that several ORAM schemes (such as tree-based ORAM schemes, and in particular the ORAM of Theorem 3.7), though described for databases given as arrays, can actually support databases given as map data structures.

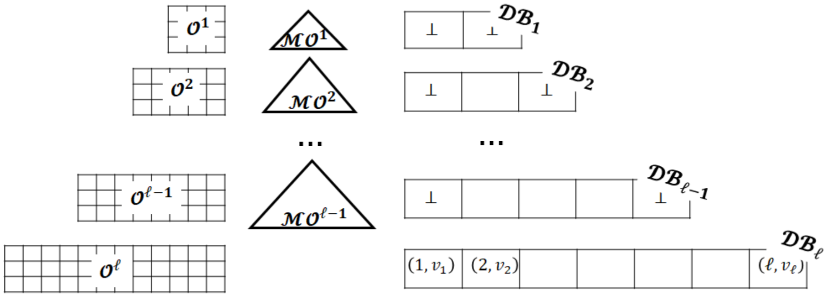

Figure 1: ORAM level structure at the onset of the computation

Finally, every "reshuffle" of level i into level $i+1$ requires re-generation of the RO-ORAM $\mathcal{O}^{i+1}$, since the contents of $\mathcal{DB}^{i+1}$ have changed. In general, re-generation cannot use the setup algorithm of the RO-ORAM due to two reasons. First, the setup is designed to be run by a trusted party, and so the server cannot run it, and since setup depends on the entire database, it is too costly for the client to run on his own. Second, while the setup of a RO-ORAM is only required to be polynomial-time (since it is only executed once, and so its cost is amortized over sufficiently many accesses to the RO-ORAM), when executed repeatedly as part of reshuffle, a more stringent efficiency requirement is needed. The first property is captured by the ORAM with oblivious setup primitive (Section 3.1). For the second property we use the fact that our RO-ORAM scheme described in Section 3 has a highly-efficient oblivious setup protocol (as described in Section 3.1).

Given these building blocks, the ORAM operates as follows. To read the block at address j from the database, the client looks for the block in every level. At the lowest level ℓ, which contains the entire database, this is done by reading the block at address j from $\mathcal{O}^\ell$. For all other levels $1 \leq i < \ell$, this is done by first reading j from $\mathcal{MO}^i$ to check whether the j'th memory block appears in $\mathcal{DB}^i$, and if so in which index t; and then using $\mathcal{O}^i$ to read the t'th block of $\mathcal{DB}^i$. (If the j'th block does not appear in $\mathcal{DB}^i$, a dummy read is performed on $\mathcal{O}^i$.) The output is the copy of block j from $\mathcal{DB}^{i^*}$ for the smallest level i^* such that $\mathcal{DB}^{i^*}$ contains the j'th memory block. This is the "correct" answer because the levels preserve the invariant that each level contains only a single copy, and the most recent copy appears in the top-most level that contains the block. An example of the execution of the read protocol is presented in Figure 2.

To write value v to the block at address j, the client asks the server to write a new copy of block j with value v to the top level. As noted above, this causes a reshuffle into lower levels at predefined intervals to prevent levels from overflowing. More specifically, every l_i write operations level i will be reshuffled into level $i+1$, where l_i denotes the size of level i. During reshuffle, all memory blocks from $\mathcal{DB}^i$ are copied into $\mathcal{DB}^{i+1}$, and multiple copies of the same memory block are consolidated by storing the level-i copy. Additionally, the ORAMs $\mathcal{MO}^{i+1}, \mathcal{O}^{i+1}$ of level $i+1$ are updated, and level i is emptied (that is, $\mathcal{DB}^i$ is replaced with an empty database, and $\mathcal{MO}^i, \mathcal{O}^i$ are updated accordingly). See Figures 4 (page 23) and 6 (page 25) for an example.

Instantiating this ORAM scheme with different values of the number of levels ℓ yields ORAM schemes with different tradeoffs between the read and write overhead. Concretely, Theorem 4.1 is obtained by setting ℓ to be constant, and Theorem 4.3 is obtained by setting $\ell = \frac{\log n}{\log^2 \log n}$.

We now formally describe the construction.

Construction 4.4 (ORAM with writes)**.** The scheme uses the following building blocks:

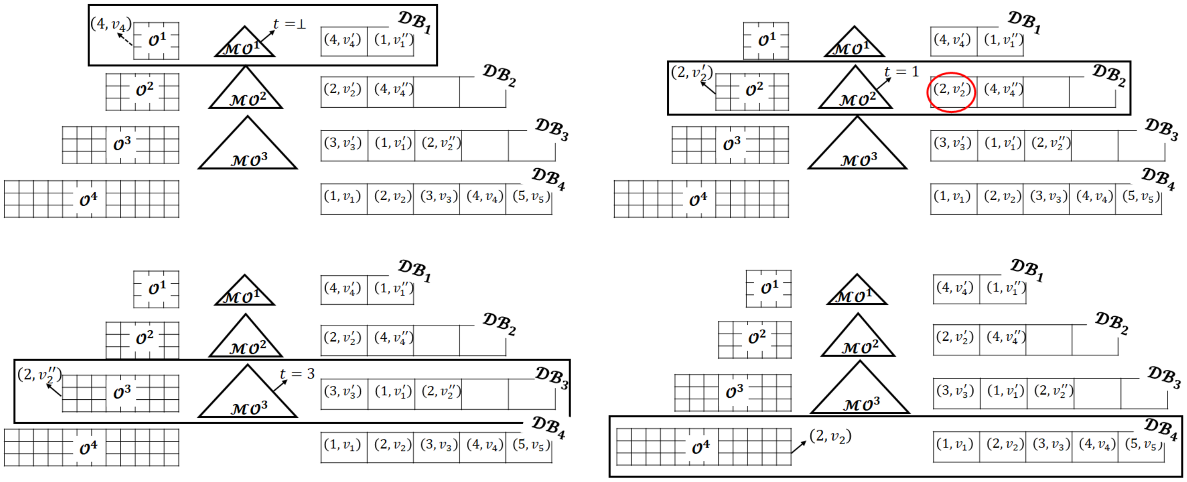

Figure 2: Read execution for addr $= 2$ in a toy-example ORAM with database size $n = 5$ and $\ell = 4$ levels. The frame indicates the level which is currently accessed, and the red circle denotes the output of the protocol. Arrows denote the output of the metadata and RO ORAMs, where dashes arrows denote dummy accesses. In level 1 (top left) $\mathcal{MO}^1$ is accessed and returns $t =\perp$ indicating that the second block is not in $\mathcal{DB}^1$, so a dummy access is performed on $\mathcal{O}^1$. In level 2 (top right), $\mathcal{MO}^2$ returns $t = 1$, indicating that block 2 appears as the first block of $\mathcal{DB}^2$. $\mathcal{O}^2$ is then accessed to retrieve the first block $(2, v_2')$ of $\mathcal{DB}^2$. This block is returned as the output, since higher levels (level 1) do not contain the block. To preserve obliviousness, lower levels are still accessed. In level 3 (bottom left), $\mathcal{MO}^3$ returns $t = 3$, indicating that block 2 appears as the third block of $\mathcal{DB}^3$. $\mathcal{O}^3$ is then accessed to retrieve the third block $(2, v_2'')$ of $\mathcal{DB}^3$, but this block is then discarded because a copy of the block already appears at a higher level. Finally, in level 4 (bottom right), $\mathcal{O}^4$ is accessed to retrieve the second block $(2, v_2)$ of $\mathcal{DB}^4$, which is also discarded.

- A RO-ORAM scheme $(\mathsf{Setup}_R, \mathsf{Read}_R)$ associated with an oblivious setup protocol $\mathsf{OblSetup}_R$.
- An ORAM scheme $(\mathsf{Setup}_m, \mathsf{Read}_m, \mathsf{Write}_m)$ for map data structures.

We define the following protocols.

- $\mathbf{Setup}(1^\lambda, \mathbf{DB})$**:** Recall that λ denotes the security parameter, and $\mathsf{DB} \in \left(\{0,1\}^{\mathsf{B}}\right)^n$. Setup does the following.
  - Initialize a writes counter. Initialize a writes counter count to 0.
  - Initialize lowest level.
    - $*$ Initialize $\mathcal{DB}^\ell = \mathsf{DB}$. We assume without loss of generality that the blocks in DB are of the form (j, b_j), namely each memory block contains its logical address.[6]
    - $*$ Generate a RO-ORAM scheme $\mathcal{O}^\ell$ for $\mathcal{DB}^\ell$ by running $\left(\mathsf{ck}_R^\ell, \mathsf{st}_R^\ell\right) \leftarrow \mathsf{Setup}_R\left(1^\lambda, \mathcal{DB}^\ell\right)$ to obtain a client key ck_R^ℓ and a server state st_R^ℓ for $\mathcal{O}^\ell$.
  - Initialize upper levels. For every level $1 \le i < \ell$:
    - $*$ Initialize $\mathcal{DB}^i$ to consist of l_i dummy memory blocks.

[6] This assumption is without loss of generality since for the block sizes we consider, concatenating the address to the block would cause at most a constant multiplicative increase in the block size.

    - ∗ Generate a RO-ORAM scheme $\mathcal{O}^i$ for $\mathcal{DB}^i$ by running $\left(\mathsf{ck}_R^i, \mathsf{st}_R^i\right) \leftarrow \mathsf{Setup}_R\left(1^\lambda, \mathcal{DB}^i\right)$ to obtain a client key ck_R^i and a server state st_R^i for $\mathcal{O}^i$.
    - ∗ Generate a map data structure $\mathcal{M}^i$ mapping each block (j, b_j) in $\mathcal{DB}^i$ to its index in $\mathcal{DB}^i$. (That is, if (j, b_j) is the t'th block of $\mathcal{DB}^i$ then the entry (t, j) is added to $\mathcal{M}^i$.)
    - ∗ Generate a metadata ORAM scheme $\mathcal{MO}^i$ for $\mathcal{M}^i$, by running $\left(\mathsf{ck}_m^i, \mathsf{st}_m^i\right) \leftarrow \mathsf{Setup}_m\left(1^\lambda, \mathcal{M}^i\right)$ to obtain the client key and server state for $\mathcal{MO}^i$.
  - <u>Output.</u> The long-term client key $\mathsf{ck} = \left(\mathsf{ck}_R^\ell, \left\{\mathsf{ck}_R^i, \mathsf{ck}_m^i\right\}_{i\in[\ell-1]}, \mathsf{count}\right)$ consists of the client keys for the RO-ORAMs $\mathcal{O}^i$ and the metadata ORAMs $\mathcal{MO}^i$ of all levels, as well as the counter count of the number of write operations performed. The server state $\mathsf{st}_S = \left(\mathsf{st}_R^\ell, \mathcal{DB}^\ell, \left\{\mathsf{st}_R^i, \mathsf{st}_m^i, \mathcal{DB}^i\right\}_{i\in[\ell-1]}\right)$ contains the server states in the RO-ORAMs $\mathcal{O}^i$ and the metadata ORAMs $\mathcal{MO}^i$ of all levels, as well as the memory contents $\mathcal{DB}^i$ of all levels.

The Read protocol. To read the memory block at location $\mathsf{addr} \in [n]$ from the server S, the client C with key $\left(\mathsf{ck}_R^\ell, \left\{\mathsf{ck}_R^i, \mathsf{ck}_m^i\right\}_{i\in[\ell-1]}, \mathsf{count}\right)$ operates as follows, where in all executions of the Read_R protocol on $\mathcal{O}^i$ (respectively, all executions of the Read_m or Write_m protocols on $\mathcal{MO}^i$) S plays the role of the server with state st_R^i (respectively, st_m^i) and C plays the role of the client with key ck_R^i (respectively, ck_m^i).

- <u>Determine block location in level i.</u> For every level $1 \leq i \leq \ell - 1$, run the Read_m protocol on $\mathcal{MO}^i$ to read the index l in which the block appears in $\mathcal{DB}^i$. (If block addr does not appear in level i, then $l =\perp$.)

- <u>Read block from level i.</u> For every level $1 \leq i \leq \ell - 1$, if $l =\perp$, set $l = 1$. Run the Read_R protocol on $\mathcal{O}^i$ to read the l'th block from $\mathcal{DB}^i$.

- <u>Read block from level ℓ.</u> Run the Read_R protocol on $\mathcal{O}^\ell$ to read the addr'th block from $\mathcal{DB}^\ell$.

- <u>Output.</u> Let i^* be the smallest such that block addr appears in $\mathcal{DB}^{i^*}$, and let (addr, v) denote the block returned by the execution of the Read_R protocol on $\mathcal{O}^{i^*}$. Output v to C. (All other memory blocks returned by the Read_R protocol executions are ignored.)

The Write protocol. To write value val at location $\mathsf{addr} \in [n]$ on the server S, the client C with key $\left(\mathsf{ck}_R^\ell, \left\{\mathsf{ck}_R^i, \mathsf{ck}_m^i\right\}_{i\in[\ell-1]}, \mathsf{count}\right)$ operates as follows.

- Generate a "dummy" level 0 which contains a single memory block $(\mathsf{addr}, \mathsf{val})$, and send it to the server.

- Update the server state and client key as follows:
  - $\mathsf{count} := \mathsf{count} + 1$.
  - If $l_{\ell-1}$ divides count, then reshuffle level $\ell - 1$ into level ℓ using the $\mathsf{ReShuffle}^\ell$ procedure of Figure 3, namely execute $\mathsf{ReShuffle}^\ell\left(\mathsf{ck}_R^{\ell-1}, \mathsf{ck}_R^\ell, \mathsf{ck}_m^{\ell-1}, \mathsf{st}_R^{\ell-1}, \mathsf{st}_R^\ell, \mathsf{st}_m^{\ell-1}\right)$.
  - For every i from $\ell - 2$ down to 0 for which l_i divides count, reshuffle level i into level $i + 1$ using the $\mathsf{ReShuffle}$ procedure of Figure 5, namely execute $\mathsf{ReShuffle}\left(i, \mathsf{ck}_R^i, \mathsf{ck}_R^{i+1}, \mathsf{ck}_m^i, \mathsf{ck}_m^{i+1}, \mathsf{st}_R^i, \mathsf{st}_R^{i+1}, \mathsf{st}_m^i, \mathsf{st}_m^{i+1}\right)$.

The $\mathsf{ReShuffle}^{\ell}$ procedure

Inputs:

$\mathsf{ck}_R^j, j \in \{\ell-1, \ell\}$**:** the client keys for the RO-ORAMs $\mathcal{O}^{\ell-1}, \mathcal{O}^{\ell}$ of levels $\ell-1, \ell$.

$\mathsf{ck}_m^{\ell-1}$**:** the client key for the metadata ORAM $\mathcal{MO}^{\ell-1}$ of level $\ell-1$.

$\mathsf{st}_R^j, j \in \{\ell-1, \ell\}$**:** the server states for the RO-ORAMs $\mathcal{O}^{\ell-1}, \mathcal{O}^{\ell}$ of levels $\ell-1, \ell$.

$\mathsf{st}_m^{\ell-1}$**:** the server state for the metadata ORAM $\mathcal{MO}^{\ell-1}$ of level $\ell-1$.

Operation:

- Updating contents of level ℓ. For every $1 \leq k \leq n$:
  - Read the k'th block (k, v_k) of $\mathcal{DB}^{\ell}$.
  - Run the Read_m protocol (with client key $\mathsf{ck}_m^{\ell-1}$ and server state $\mathsf{st}_m^{\ell-1}$) on $\mathcal{MO}^{\ell-1}$ to read the index t in which memory block k appears in $\mathcal{DB}^{\ell-1}$. (If memory block k does not appear in $\mathcal{DB}^{\ell-1}$ then Read_m returns $\perp$ to the client.)
  - Run the Read_R protocol (with client key $\mathsf{ck}_R^{\ell-1}$ and server state $\mathsf{st}_R^{\ell-1}$) on $\mathcal{O}^{\ell-1}$ to read the value v'_k of the t'th block in $\mathcal{DB}^{\ell-1}$. (If $t = \perp$, perform a dummy read of the block at index 1.)
  - If $t \neq \perp$, replace the k'th block in $\mathcal{DB}^{\ell}$ with (k, v'_k). Otherwise, replace the k'th block with (k, v_k) (this is a dummy write).
- Updating RO-ORAMs. Replace $\mathcal{DB}^{\ell-1}$ with a database consisting of $l_{\ell-1}$ dummy blocks. For $j = \ell-1, \ell$, run the $\mathsf{OblSetup}_R$ protocol to generate a new RO-ORAM $\mathcal{O}^j$ for $\mathcal{DB}^j$: $\left(\widetilde{\mathsf{ck}}_R^j, \widetilde{\mathsf{st}}_R^j\right) \leftarrow \mathsf{OblSetup}_R\left(1^{\lambda}, \mathcal{DB}^j\right)$. Replace $\mathsf{ck}_R^j, \mathsf{st}_R^j$ with $\widetilde{\mathsf{ck}}_R^j, \widetilde{\mathsf{st}}_R^j$, respectively.
- Updating metadata ORAM. For every $1 \leq k \leq l_{\ell-1}$:
  - Read the k'th block (j, v_j) of $\mathcal{DB}^{\ell-1}$.
  - Remove the entry corresponding to k from $\mathcal{M}^{\ell-1}$ by executing the Write_m protocol on $\mathcal{MO}^{\ell-1}$ (with client key $\mathsf{ck}_m^{\ell-1}$ and server state $\mathsf{st}_m^{\ell-1}$).

Figure 3: The $\mathsf{ReShuffle}^{\ell}$ protocol used in Construction 4.4

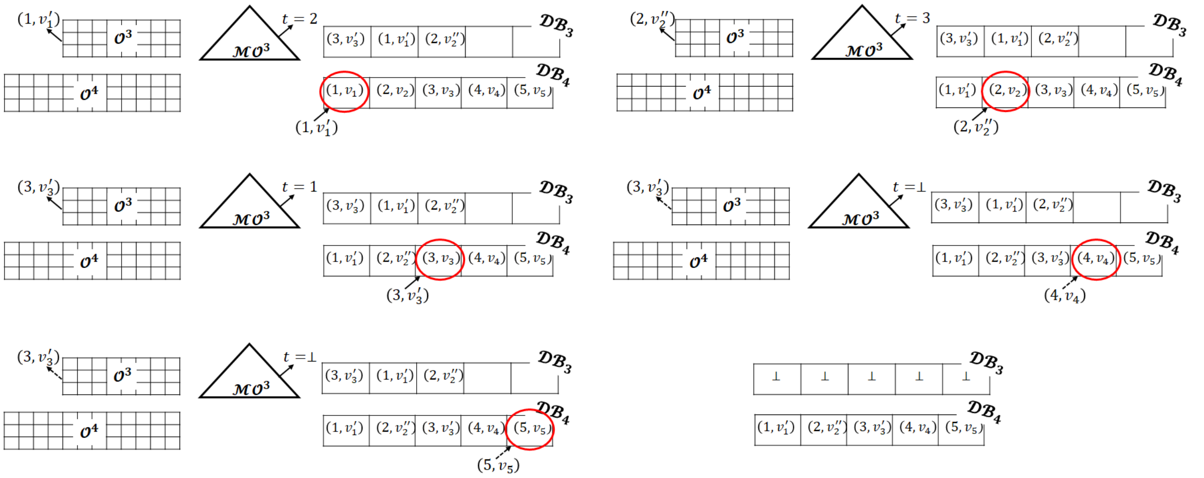

Figure 4: $\mathsf{ReShuffle}^{\ell}$ execution on the ORAM from Figure 2. The red circle indicates the block which is currently updated. Arrows denote the output of the metadata and RO ORAMs, where dashes arrows denote dummy accesses. Block 1 is updated first (top left), $\mathcal{MO}^3$ is accessed and returns $t = 2$ indicating that block 1 appears as the second block of $\mathcal{DB}^3$. The block $(1, v_1')$ is then read from $\mathcal{O}^3$, and updated in $\mathcal{DB}^4$. Block 2 is updated next (top right), $\mathcal{MO}^3$ is accessed and returns $t = 3$ indicating that block 2 appears as the third block of $\mathcal{DB}^3$. The block $(2, v_2'')$ is then read from $\mathcal{O}^3$, and updated in $\mathcal{DB}^4$. Block 3 is updated next (center left), $\mathcal{MO}^3$ is accessed and returns $t = 1$ indicating that block 3 appears as the first block of $\mathcal{DB}^3$. The block $(3, v_3')$ is then read from $\mathcal{O}^3$, and updated in $\mathcal{DB}^4$. Block 4 is updated next (center right), $\mathcal{MO}^3$ is accessed and returns $t = \perp$, indicating that block 4 does not appear in $\mathcal{DB}^3$. Therefore, a dummy read is performed on $\mathcal{O}^3$, and a dummy write is performed on $\mathcal{DB}^4$. Finally, block 5 is updated (bottom left), $\mathcal{MO}^3$ is accessed and returns $t = \perp$, indicating that block 5 does not appear in $\mathcal{DB}^3$. Therefore, a dummy read is performed on $\mathcal{O}^3$, and a dummy write is performed on $\mathcal{DB}^4$. The values of $\mathcal{DB}^3, \mathcal{DB}^4$ at the end of the $\mathsf{ReShuffle}^{\ell}$ execution are depicted at the bottom right (these values are used to generate new RO-ORAMs $\mathcal{O}^3, \mathcal{O}^4$, and update the metadata ORAMs $\mathcal{MO}^3, \mathcal{MO}^4$).

Remark on De-amortization. We note that using a technique of Ostrovsky and Shoup [OS97], the server complexity in Construction 4.4 can be de-amortized, by slightly modifying the Write protocol to allow the server to *spread-out* the reshuffling process over multiple accesses to the ORAM. The reason reshuffle operations can be "spread out" is that reshuffling is performed in a "bottom-up" fashion, namely when it is time to reshuffle level i into level $i+1$, that reshuffling is executed *before* level $i-1$ is reshuffled into level i. Thus, the memory blocks that are involved in the reshuffle of level i into level $i+1$ have been known for the last l_{i-1} time units, ever since level i was last updated due to a reshuffle of level $i-1$ into it. Therefore, the operations needed to perform the reshuffle of level i into level $i+1$ can be spread out over l_{i-1} operations.

We prove the following claims about Construction 4.4.

Claim 4.5 (ORAM security)**.** *Assuming the security of all of the building blocks, Construction 4.4 is a secure ORAM scheme.*

Claim 4.6 (ORAM complexity)**.** *Assume that:*

- *The database* DB*, and the underlying ORAM scheme, have block size* $\mathsf{B}, \mathsf{mB} \geq 2\log n$*, respectively.*

- *The underlying RO-ORAM scheme has computation and communication overhead* $\mathsf{Ovh}_R(N)$*, and* $\mathsf{OblSetup}_R$ *has runtime* $\mathsf{T}(N)$*, for databases of size* N*.*

The $\mathsf{ReShuffle}$ procedure

Inputs:

i**:** the index of a level to reshuffle.

$\mathsf{ck}_R^j, j \in \{i, i+1\}$**:** the client keys for the RO-ORAMs $\mathcal{O}^i, \mathcal{O}^{i+1}$ of levels $i, i+1$.

$\mathsf{ck}_m^j, j \in \{i, i+1\}$**:** the client keys for the metadata ORAMs $\mathcal{MO}^i, \mathcal{MO}^{i+1}$ of levels $i, i+1$.

$\mathsf{st}_R^j, j \in \{i, i+1\}$**:** the server states for the RO-ORAMs $\mathcal{O}^i, \mathcal{O}^{i+1}$ of levels $i, i+1$.

$\mathsf{st}_m^j, j \in \{i, i+1\}$**:** the server states for the metadata ORAMs $\mathcal{MO}^i, \mathcal{MO}^{i+1}$ of levels $i, i+1$.

Operation:

- Let $m = \mathsf{count} \mod l_{i+1}$. (Notice that level $i+1$ contains at most m elements.)
- Updating level-$(i+1)$ blocks. For every $1 \leq k \leq m$:
  1. Read the k'th block (j, v_j) from $\mathcal{DB}^{i+1}$.
  2. Run the Read_m protocol (with client key ck_m^i and server state st_m^i) on $\mathcal{MO}^i$ to read the index t in which memory block j appears in $\mathcal{DB}^i$. (If memory block j does not appear in $\mathcal{DB}^i$ then Read_m returns $\perp$ to the client.)
  3. Run the Read_R protocol (with client key ck_R^i and server state st_R^i) on $\mathcal{O}^i$ to read the value v_j' of the t'th block in $\mathcal{DB}^i$. (If $t = \perp$, perform a dummy read to the block at index 1.)
  4. If $t \neq \perp$, replace the k'th block in $\mathcal{DB}^{i+1}$ with $\left(j, v_j'\right)$. Otherwise, replace the k'th block with (j, v_j) (this is a dummy write).
  5. If $t \neq \perp$, remove the entry corresponding to t from $\mathcal{M}^i$ by executing the Write_m protocol on $\mathcal{MO}^i$. Otherwise, perform a dummy write to $\mathcal{MO}^i$, writing back the entry corresponding to t that was read in step 2.
- Copying level-i blocks that were not in $\mathcal{DB}^{i+1}$. Initialize a counter count' to $m+1$. For every $1 \leq k \leq l_i$:
  1. Read the k'th block (j, v_j) of $\mathcal{DB}^i$.
  2. Run the Read_m protocol (with client key ck_m^i and server state st_m^i) on $\mathcal{MO}^i$ to read the index t in which memory block j appears in $\mathcal{DB}^i$. (This step checks whether the k'th block has been deleted from $\mathcal{DB}^i$ in the previous step. If so, then Read_m returns $\perp$ to the client.)
  3. If $t \neq \perp$, write (j, v_j) as the count''th block of $\mathcal{DB}^{i+1}$. Otherwise, write a dummy block as the count''th block of $\mathcal{DB}^{i+1}$.
  4. If $t \neq \perp$, run the Write_m protocol (with client key ck_m^{i+1} and server state st_m^{i+1}) to write (count', j) to $\mathcal{MO}^{i+1}$. Otherwise, perform a dummy write to $\mathcal{MO}^{i+1}$.
  5. If $t \neq \perp$, remove the entry corresponding to t from $\mathcal{M}^i$ by executing the Write_m protocol on $\mathcal{MO}^i$. Otherwise, perform a dummy write to $\mathcal{MO}^i$.
  6. Update the counter: $\mathsf{count}' := \mathsf{count}' + 1$.
- Updating level ORAMs. Replace $\mathcal{DB}^i$ with a database consisting of l_i dummy blocks. For $j = i, i+1$, run the $\mathsf{OblSetup}_R$ protocol to generate a new RO-ORAM $\mathcal{O}^j$ for $\mathcal{DB}^j$: $\left(\widetilde{\mathsf{ck}}_R^j, \widetilde{\mathsf{st}}_R^j\right) \leftarrow \mathsf{OblSetup}_R\left(1^\lambda, \mathcal{DB}^j\right)$. Replace $\mathsf{ck}_R^j, \mathsf{st}_R^j$ with $\widetilde{\mathsf{ck}}_R^j, \widetilde{\mathsf{st}}_R^j$, respectively.

Figure 5: The $\mathsf{ReShuffle}$ protocol used in Construction 4.4

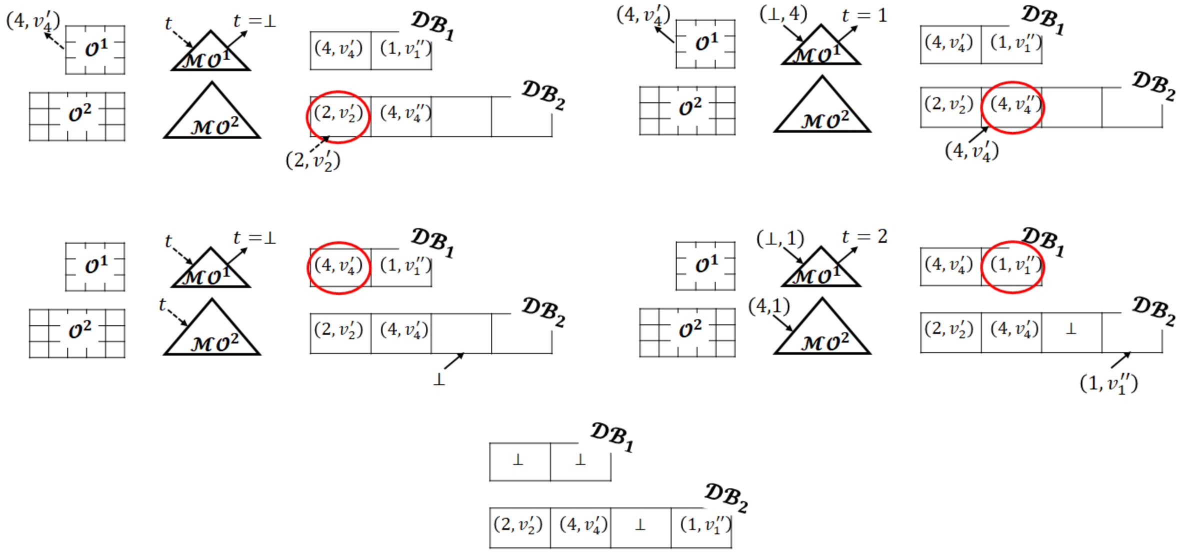

Figure 6: ReShuffle execution for $i = 1$ on the ORAM from Figure 2. The red circle indicates the block which is currently updated. Arrows denote the output of the metadata and RO ORAMs, where dashes arrows denote dummy accesses. The blocks of $\mathcal{DB}^2$ are updated first. The first block of $\mathcal{DB}^2$ is updated first (top left), $\mathcal{MO}^1$ is accessed and returns $t = \perp$ indicating that this block does not appear in $\mathcal{DB}^1$. Therefore, a dummy read is performed on $\mathcal{O}^1$, and dummy writes are performed on $\mathcal{MO}^1, \mathcal{DB}^2$. The second block of $\mathcal{DB}^2$ is updated next (top right), $\mathcal{MO}^1$ is accessed and returns $t = 1$ indicating that this block appears as the first block of $\mathcal{DB}^1$. The block $(4, v'_4)$ is then read from $\mathcal{O}^1$, and updated in $\mathcal{DB}^2$. Then, the block is deleted from $\mathcal{DB}^1$ by updating $\mathcal{MO}^1$ (replacing the entry $(1, 4)$ with $(\perp, 4)$). Next, the blocks of $\mathcal{DB}^1$ are copied into $\mathcal{DB}^2$. The first block of $\mathcal{DB}^1$ is copied first. $\mathcal{MO}^1$ is accessed and returns $t = \perp$, indicating that this block was already copied into $\mathcal{DB}^2$ (and removed from $\mathcal{DB}^1$). Therefore, a dummy block is written to $\mathcal{DB}^2$, and dummy writes are performed on $\mathcal{MO}^1, \mathcal{MO}^2$. Finally, the second block of $\mathcal{DB}^1$ is copied. $\mathcal{MO}^1$ is accessed and returns $t = 2$, indicating that the block has not been removed from $\mathcal{DB}^1$. The block is then written into $\mathcal{DB}^2$, $\mathcal{MO}^2$ is updated to reflect that block 1 appears as the fourth block of $\mathcal{DB}^2$, and the block is deleted from $\mathcal{DB}^1$ by updating $\mathcal{MO}^1$ accordingly. The values of $\mathcal{DB}^1, \mathcal{DB}^2$ at the end of the ReShuffle execution are depicted at the bottom (these values are used to generate new RO-ORAMs $\mathcal{O}^1, \mathcal{O}^2$).

- *The underlying ORAM scheme has computation and communication overhead* $\mathsf{Ovh}_m(N)$.

Then each execution of the Read *protocol of Construction 4.4 performs (communicates, respectively)*

$$\mathsf{B} \cdot \mathsf{Ovh}_R(n) + \sum_{i=1}^{\ell-1} \left(\mathsf{mB} \cdot \mathsf{Ovh}_m \left(\frac{2l_i \log n}{\mathsf{mB}} \right) + \mathsf{B} \cdot \mathsf{Ovh}_R(l_i) \right)$$

bit operations (bits, respectively), and each execution of the Write *protocol of Construction 4.4 performs (communicates, respectively)*

$$\mathsf{B} + \sum_{i=0}^{\ell-1} \frac{l_{i+1}}{l_i} \cdot \left(2\mathsf{B} + 2\mathsf{mB} \cdot \mathsf{Ovh}_m \left(\frac{2l_i \log n}{\mathsf{mB}} \right) + \mathsf{B} \cdot \mathsf{Ovh}_R(l_i) \right)$$

$$+ \sum_{i=0}^{\ell-1} \left(\log l_{i+1} + \mathsf{mB} \cdot \mathsf{Ovh}_m \left(\frac{2l_{i+1} \log n}{\mathsf{mB}} \right) + \frac{\mathsf{T}(l_i) + \mathsf{T}(l_{i+1})}{l_i} \right)$$

bit operations (bits, respectively).

Remark: Simplified Overhead Bounds. We note that the overheads in Construction 4.4 can be simplified when it is instantiated with a metadata ORAM with poly-logarithmic overhead, and for a large enough (poly-logarithmic) block size $\mathsf{B} = \operatorname{poly}\log n$ for which $\mathsf{mB} \cdot \mathsf{Ovh}_m(n) = O(\mathsf{B})$. Concretely, the overheads described in Claim 4.6 can be simplified as follows: the Read protocol has $O(\ell) + \sum_{i=1}^{\ell} \mathsf{Ovh}_R(l_i)$ communication and computational overhead, and the Write protocol has

$$\sum_{i=0}^{\ell-1}\left(\frac{l_{i+1}}{l_i}\cdot\left(\mathsf{Ovh}_R(l_i)+O(1)\right)+\frac{\mathsf{T}(l_i)+\mathsf{T}(l_{i+1})}{l_i\cdot\mathsf{B}}+O(1)\right)$$

communication and computational overhead.

Claims Imply Theorems. To prove Theorems 4.1-4.3, we instantiate Construction 4.4 with the RO-ORAM of Construction 3.4 (together with the $\mathsf{OblSetup}$ protocol of Construction 3.10), and the path ORAM scheme of Theorem 3.7 as the metadata ORAM.

Proof of Theorem 4.1. Security follows directly from Claim 4.5 since by Claim 3.5, Lemma 3.11 and Theorem 2.4, the assumptions imply the existence of a secure RO-ORAM scheme with oblivious setup.

As for the complexity of the construction, we choose $l_i = n^{i\cdot\mu}$ where $\mu = \frac{\epsilon-(\delta+\gamma+\delta\gamma)}{2(1+\delta)(1+\gamma)}$ is constant. Then the ORAM has a constant number $\ell = 1/\mu$ of levels. Recall that for every $i \in [\ell-1]$, $\left|\mathcal{MO}^i\right| \leq 2n\log n$, and let $N := \frac{2n\log n}{\log^2 n\log\log n}$. Therefore, the ORAM scheme of Theorem 3.7 satisfies $\mathsf{mB} = \log^2 N\log\log N$, and $\mathsf{Ovh}_m(N) = O(\log N)$. Moreover, by Theorem 3.1, and the assumptions of Theorem 4.1, $\mathsf{Ovh}_R(n) = O(\log\log n)$.

Regarding the overhead of the Read protocol, Claim 4.6 guarantees that every execution of the Read protocol consists of performing (sending, respectively)

$$\mathsf{B}\cdot O(\log\log n)+\sum_{i=1}^{\ell-1}\log^2 N\log\log N\cdot O\left(\log\left(\frac{2n^{i\cdot\mu}\log n}{\log^2 N\log\log N}\right)\right)+\sum_{i=1}^{\ell-1}\mathsf{B}\cdot O\left(\log\log n^{i\cdot\mu}\right)$$

bit operations (bits, respectively). For $\mathsf{B} = \Omega\left(\lambda\log^3 n\log^7\log n\right)$ as in the theorem statement, with a sufficiently large constant in the $\Omega(\cdot)$ notation, and since ℓ, μ are constants, the second summand is at most $O(\mathsf{B})$. The fact that $\ell, \mu = O(1)$ also implies that the third summand is $O(\mathsf{B}\cdot\log\log n)$. Therefore, the overall computation and communication overhead of read operations is $O(\log\log n)$.

As for the overhead of the Write protocol, Claim 4.6 guarantees that every execution of the Write protocol consists of performing (sending, respectively)

$$\begin{aligned}&\mathsf{B}+\sum_{i=0}^{\ell-1}\frac{l_{i+1}}{l_i}\cdot\left(2\mathsf{B}+2\mathsf{mB}\cdot\mathsf{Ovh}_m\left(\frac{2l_i\log n}{\mathsf{mB}}\right)+\mathsf{B}\cdot\mathsf{Ovh}_R(l_i)\right)\\&+\sum_{i=0}^{\ell-1}\left(\log l_{i+1}+\mathsf{mB}\cdot\mathsf{Ovh}_m\left(\frac{2l_{i+1}\log n}{\mathsf{mB}}\right)+\frac{\mathsf{T}(l_i)+\mathsf{T}(l_{i+1})}{l_i}\right)\end{aligned}$$

bit operations (bits, respectively). We analyze each summand separately. First, by Theorem 3.7, $\mathsf{Ovh}_m\left(\frac{2l_i\log n}{\mathsf{mB}}\right) \leq O(\log n)$ for every i, and as noted above $\mathsf{Ovh}_R(l_i) = O(\log\log n)$. Therefore,

$$\frac{l_{i+1}}{l_i}\cdot\left(2\mathsf{B}+2\mathsf{mB}\cdot\mathsf{Ovh}_m\left(\frac{2l_i\log n}{\mathsf{mB}}\right)+\mathsf{B}\cdot\mathsf{Ovh}_R(l_i)\right)=O\left(n^{\mu}\cdot\mathsf{B}\cdot\log\log n\right)$$

(here, we also use the fact that $\mathsf{mB}\cdot O(\log n)\leq O(\mathsf{B})$).

Moreover,

$$\mathsf{mB}\cdot\mathsf{Ovh}_m\left(\frac{2l_{i+1}\log n}{\mathsf{mB}}\right)\leq O(\mathsf{mB}\cdot\log n)=O(\mathsf{B})$$

and since ℓ is constant then

$$\sum_{i=0}^{\ell-1}\log l_{i+1}=O(\log n)\,.$$

Finally, to bound the term $\frac{\mathsf{T}(l_i)+\mathsf{T}(l_{i+1})}{l_i}$, we first bound $\mathsf{T}(\cdot)$. According to Lemma 3.12, the execution of $\mathsf{OblSetup}$ requires performing

$$\lambda+\mathsf{T}_m\left(\frac{k(M+\log M)}{\mathsf{mB}}\right)+\mathsf{T}_{\mathsf{LDC}}(n)+kM+\left(\frac{kM}{\mathsf{mB}}+kM\right)\cdot\mathsf{mB}\cdot\mathsf{Ovh}\left(\frac{k(M+\log M)}{\mathsf{mB}}\right)+k\cdot\mathsf{comp}(n,\mathsf{B})$$

bit operations, and communicating

$$\mathsf{T}_m\left(\frac{k(M+\log M)}{\mathsf{mB}}\right)+k\cdot\mathsf{comp}(n,\mathsf{B})$$

bits, where $\mathsf{T}_m(N)$ denotes the number of bit operations required to set-up the metadata ORAM on size-N databases, $\mathsf{Ovh}(N)$ denote the overhead of the metadata ORAM on size-N databases, $\mathsf{T}_{\mathsf{LDC}}(n)$ denotes the number of bit operations required to LDC-encode length-n messages, and $\mathsf{comp}(n,\mathsf{B})$ denotes the number of bit operations performed during the execution of the oblivious-access sort algorithm.

For the specific building blocks assumed in Theorem 4.1, and by Theorem 3.7, $\mathsf{T}_m(N)=O(N\cdot\mathsf{mB})$, so $\mathsf{T}_m\left(\frac{k(M+\log M)}{\mathsf{mB}}\right)=O(kM\log M)=O\left(n^{1+\delta}\log n\right)$, and $\mathsf{Ovh}(N)=O(\log N)$ so $\mathsf{Ovh}\left(\frac{k(M+\log M)}{\mathsf{mB}}\right)=O(\log n)$. Moreover, $\mathsf{T}_{\mathsf{LDC}}(n)=n^{(1+\delta)(1+\gamma)}$, and $\mathsf{comp}(n,\mathsf{B})=O(n\log\log n)$. Therefore, executing $\mathsf{OblSetup}$ requires $\lambda+n^{(1+\delta)(1+\gamma)}+O\left(n^{1+\delta}\log^4 n\right)=O\left(n^{(1+\delta)(1+\gamma)}\right)$ bit operations (for a large enough n), and communicating $O\left(n^{1+\delta}\log n\right)$ bits.

Therefore,

$$\frac{\mathsf{T}(l_i)+\mathsf{T}(l_{i+1})}{l_i}\leq\frac{2\mathsf{T}(l_{i+1})}{l_i}=2\cdot\frac{O\left(n^{(i+1)\mu(1+\delta)(1+\gamma)}\right)}{n^{i\mu}}=O\left(n^{i\mu(\delta+\gamma+\delta\gamma)+\mu(1+\delta)(1+\gamma)}\right)$$

so

$$\sum_{i=0}^{\ell-1}\frac{\mathsf{T}(l_i)+\mathsf{T}(l_{i+1})}{l_i}\leq 2\ell\cdot O\left(n^{\ell\mu(\delta+\gamma+\delta\gamma)+\mu(1+\delta)(1+\gamma)}\right)\leq O(n^\epsilon)$$

where the right-most inequality follows from the choice of μ.

In summary, since $\ell=O(1)$ then every execution of the Write protocol requires performing (sending, respectively)

$$O\left(\mathsf{B}+n^\mu\log\log n\cdot\mathsf{B}+n^\epsilon\right)=O\left(\mathsf{B}\cdot n^\epsilon\right)$$

bit operations (bits, respectively), for a large enough n (since for large enough n, $n^\mu\log\log n\leq n^{2\mu}\leq n^\epsilon$). Consequently, the communication and computation overhead of write operations is $O(n^\epsilon)$.

Finally, regarding client storage, emulating the LDC decoder requires storing $\mathsf{space}(k)=O(1)$ size-B blocks. By the choice of B, running $\mathsf{OblSetup}$ (as part of reshuffle operations) requires the client to store only $O(1)$ size-B blocks. Operations on mDB require (by Theorem 3.7) storing $O(\log N\log\log N)$ size-mB blocks which corresponds to a constant number of size-B blocks. □

The proof of Theorem 4.2 is similar to the proof of Theorem 4.1, so we only sketch the differences.

Proof of Theorem 4.2. As in the proof of Theorem 4.1, we choose $l_i = n^{i \cdot \mu}$ where $\mu = \frac{\epsilon - (\delta + \gamma + \delta\gamma)}{2(1+\delta)(1+\gamma)}$ is constant, so $\ell = 1/\mu$.

Regarding the overhead of the Read protocol, $\mathsf{Ovh}_R(n) = o(\log n)$ by Theorem 3.1 and the assumptions of Theorem 4.1. Therefore, Claim 4.6 guarantees that every execution of the Read protocol consists of performing (sending, respectively)

$$\mathsf{B} \cdot o(\log n) + \sum_{i=1}^{\ell-1} \log^2 N \log\log N \cdot O\left(\log\left(\frac{2n^{i\cdot\mu}\log n}{\log^2 N \log\log N}\right)\right) + \sum_{i=1}^{\ell-1} \mathsf{B} \cdot o\left(\log n^{i\cdot\mu}\right)$$

bit operations (bits, respectively). For our choice of B, μ, and ℓ, this is at most $o(\mathsf{B} \cdot \log n)$.

As for the overhead of the Write protocol, the analysis is identical to that of Theorem 4.1, except for the second summand (which depends on Ovh_R), and the last summand (which depends on the query complexity of the LDC decoder, and the complexity of the oblivious-access sort algorithm). Concretely, the second summand is:

$$\sum_{i=0}^{\ell-1} \frac{l_{i+1}}{l_i} \cdot \left(2\mathsf{B} + 2\mathsf{mB} \cdot \mathsf{Ovh}_m\left(\frac{2l_i \log n}{\mathsf{mB}}\right) + \mathsf{B} \cdot \mathsf{Ovh}_R(l_i)\right) = o\left(n^{\mu} \cdot \mathsf{B} \cdot \log n\right) + O(\mathsf{B}).$$

To bound the last summand, we bound the complexity of the $\mathsf{OblSetup}$ protocol. Recall that by Lemma 3.12, $\mathsf{OblSetup}$ performs

$$\lambda + \mathsf{T}_m\left(\frac{k(M + \log M)}{\mathsf{mB}}\right) + \mathsf{T}_{\mathsf{LDC}}(n) + kM + \left(\frac{kM}{\mathsf{mB}} + kM\right) \cdot \mathsf{mB} \cdot \mathsf{Ovh}\left(\frac{k(M + \log M)}{\mathsf{mB}}\right) + k \cdot \mathsf{comp}(n, \mathsf{B})$$

bit operations. For the parameters assumed in the theorem statement, $\mathsf{T}_m\left(\frac{k(M+\log M)}{\mathsf{mB}}\right) = O(kM \log M) = n^{1+\delta} \cdot \operatorname{poly}\log\log n$, and $\mathsf{Ovh}\left(\frac{k(M+\log M)}{\mathsf{mB}}\right) = O(\log n)$. Moreover, $\mathsf{T}_{\mathsf{LDC}}(n) = n^{(1+\delta)(1+\gamma)}$, and $\mathsf{comp}(n, \mathsf{B}) = O(n \log\log n) + o\left(\frac{n \cdot \log n}{\log^{2c} \log n}\right)$ (where $k = \log^c \log n$). Therefore, $\mathsf{OblSetup}$ performs $\lambda + n^{(1+\delta)(1+\gamma)} + n^{1+\delta} \operatorname{poly}\log\log n = O\left(n^{(1+\delta)(1+\gamma)}\right)$ bit operations (for a large enough n). Moreover, by Lemma 3.12 and the above analysis, $\mathsf{OblSetup}$ communicates $\mathsf{T}_m\left(\frac{k(M+\log M)}{\mathsf{mB}}\right) + k \cdot \mathsf{comp}(n, \mathsf{B}) = n^{1+\delta} \operatorname{poly}\log\log n + n \log n \operatorname{poly}\log\log n + o(n \log n) = n^{1+\delta} \operatorname{poly}\log\log n$ bits (for a large enough n).

Therefore,

$$\sum_{i=0}^{\ell-1} \frac{\mathsf{T}(l_i) + \mathsf{T}(l_{i+1})}{l_i} \leq O(n^{\epsilon})$$

similar to the proof of Theorem 4.1.

Consequently, every execution of the Write protocol performs (sends, respectively)

$$o\left(n^{\mu} \cdot \mathsf{B} \cdot \log n\right) + O\left(n^{\epsilon} \cdot \mathsf{B}\right) = O\left(n^{\epsilon} \cdot \mathsf{B}\right)$$

bit operations (bits, respectively) for a large enough n.

Finally, regarding client storage, emulating the LDC decoder requires storing $\mathsf{space}(k) = \operatorname{poly}\log\log n$ size-B blocks, running $\mathsf{OblSetup}$ requires the client to store only $O(1)$ size-B blocks, and operations on mDB require storing a constant number of size-B blocks. □

Proof of Theorem 4.3. Security follows directly from Claim 4.5 since by Claim 3.5, Lemma 3.11 and Theorem 2.4, the assumptions in the theorem statement imply the existence of a secure RO-ORAM scheme with oblivious setup.

As for the complexity of the construction, we choose $l_i = 2^{i \cdot \log^2 \log n}$, so the ORAM has $\ell = \frac{\log n}{\log^2 \log n}$ levels. Let $N := \frac{2n \log n}{\log^2 n \log \log n}$, and recall that $\mathsf{mB} = \log^2 N \log \log N$ and $\mathsf{Ovh}_m(N) = O(\log N)$ (by Theorem 3.7), and $\mathsf{Ovh}_R(n) = O(\log \log n)$ (by Theorem 3.1, and the assumptions in the theorem statement).

Regarding the overhead of the Read protocol, Claim 4.6 guarantees that every execution of the Read protocol consists of performing (sending, respectively)

$$\sum_{i=1}^{\ell-1} \log^2 N \log \log N \cdot O\left(\log\left(\frac{2l_i \log n}{\log^2 N \log \log N}\right)\right) + \sum_{i=1}^{\ell} \mathsf{B} \cdot O(\log \log l_i) = \sum_{i=1}^{\ell} \mathsf{B} \cdot O(\log \log l_i)$$

bit operations (bits, respectively), where the equality holds for $\mathsf{B} = \Omega\left(\lambda \log^3 n \log^7 \log n\right)$ as in the theorem statement, with a sufficiently large constant in the $\Omega(\cdot)$ notation. We can upper-bound this by

$$\mathsf{B} \cdot \ell \cdot O(\log \log n) = \mathsf{B} \cdot \frac{\log n}{\log^2 \log n} \cdot O(\log \log n) = o(\mathsf{B} \cdot \log n).$$

As for the overhead of the Write protocol, Claim 4.6 guarantees that every execution of the Write protocol consists of performing (sending, respectively)

$$\mathsf{B} + \sum_{i=0}^{\ell-1} \frac{l_{i+1}}{l_i} \cdot \left(2\mathsf{B} + 2\mathsf{mB} \cdot \mathsf{Ovh}_m\left(\frac{2l_i \log n}{\mathsf{mB}}\right) + \mathsf{B} \cdot \mathsf{Ovh}_R(l_i)\right)$$

$$+ \sum_{i=0}^{\ell-1} \left(\log l_{i+1} + \mathsf{mB} \cdot \mathsf{Ovh}_m\left(\frac{2l_{i+1} \log n}{\mathsf{mB}}\right) + \frac{\mathsf{T}(l_i) + \mathsf{T}(l_{i+1})}{l_i}\right)$$

bit operations (bits, respectively). We analyze each summand separately, using the fact that $\mathsf{mB} \cdot \mathsf{Ovh}_m\left(\frac{2l_i \log n}{\mathsf{mB}}\right) \leq O(\mathsf{B})$ for every i (by Theorem 3.7 and the choice of B), and $\mathsf{Ovh}_R(l_i) = O(\log \log n)$ (by Theorem 3.1 and the assumptions in the theorem statement). First,

$$\frac{l_{i+1}}{l_i} \cdot \left(2\mathsf{B} + 2\mathsf{mB} \cdot \mathsf{Ovh}_m\left(\frac{2l_i \log n}{\mathsf{mB}}\right) + \mathsf{B} \cdot \mathsf{Ovh}_R(l_i)\right) = O\left(\mathsf{B} \cdot 2^{\log^2 \log n} \cdot \log \log n\right)$$

so

$$\sum_{i=0}^{\ell-1} \frac{l_{i+1}}{l_i} \cdot \left(2\mathsf{B} + 2\mathsf{mB} \cdot \mathsf{Ovh}_m\left(\frac{2l_i \log n}{\mathsf{mB}}\right) + \mathsf{B} \cdot \mathsf{Ovh}_R(l_i)\right) = O\left(\mathsf{B} \cdot 2^{\log^2 \log n} \cdot \frac{\log n}{\log \log n}\right)$$

Moreover,

$$\sum_{i=0}^{\ell-1} \mathsf{mB} \cdot \mathsf{Ovh}_m\left(\frac{2l_{i+1} \log n}{\mathsf{mB}}\right) = O\left(\mathsf{B} \cdot \frac{\log n}{\log^2 \log n}\right)$$

and

$$\sum_{i=0}^{\ell-1} \log l_{i+1} \leq \ell \cdot \log n = \frac{\log^2 n}{\log^2 \log n}.$$

Finally, to bound the term $\frac{\mathsf{T}(l_i)+\mathsf{T}(l_{i+1})}{l_i}$, we first bound $\mathsf{T}(\cdot)$. According to Lemma 3.12, the execution of $\mathsf{OblSetup}$ requires performing

$$\lambda+\mathsf{T}_m\left(\frac{k\left(M+\log M\right)}{\mathsf{mB}}\right)+\mathsf{T}_{\mathsf{LDC}}\left(n\right)+kM+\left(\frac{kM}{\mathsf{mB}}+kM\right)\cdot\mathsf{mB}\cdot\mathsf{Ovh}\left(\frac{k\left(M+\log M\right)}{\mathsf{mB}}\right)+k\cdot\mathsf{comp}\left(n,\mathsf{B}\right)$$

bit operations, and communicating

$$\mathsf{T}_m\left(\frac{k\left(M+\log M\right)}{\mathsf{mB}}\right)+k\cdot\mathsf{comp}\left(n,\mathsf{B}\right)$$

bits, where $\mathsf{T}_m\left(N\right)$ denotes the number of bit operations required to set-up the metadata ORAM on size-N databases, $\mathsf{Ovh}\left(N\right)$ denote the overhead of the metadata ORAM on size-N databases, $\mathsf{T}_{\mathsf{LDC}}\left(n\right)$ denotes the number of bit operations required to LDC-encode length-n messages, and $\mathsf{comp}\left(n,\mathsf{B}\right)$ denotes the number of bit operations performed during the execution of the oblivious-access sort algorithm.

For the specific building blocks assumed in Theorem 4.3, and by Theorem 3.7, $\mathsf{T}_m\left(N\right)=O\left(N\cdot\mathsf{mB}\right)$, so $\mathsf{T}_m\left(\frac{k(M+\log M)}{\mathsf{mB}}\right)=O\left(kM\log M\right)=n^{1+o(1)}$ (for a large enough n), and $\mathsf{Ovh}\left(N\right)=O\left(\log N\right)$ so $\mathsf{Ovh}\left(\frac{k(M+\log M)}{\mathsf{mB}}\right)=O\left(\log n\right)$. Moreover, $\mathsf{T}_{\mathsf{LDC}}\left(n\right)=n^{1+o(1)}$, and $\mathsf{comp}\left(n,\mathsf{B}\right)=O\left(n\log\log n\right)$. Therefore, executing $\mathsf{OblSetup}$ requires $\lambda+n^{1+o(1)}=n^{1+o(1)}$ bit operations (for a large enough n), and communicating $n^{1+o(1)}$ bits.

Therefore,

$$\sum_{i=0}^{\ell-1}\frac{\mathsf{T}\left(l_i\right)+\mathsf{T}\left(l_{i+1}\right)}{l_i}\leq\sum_{i=0}^{\ell-1}\frac{2\mathsf{T}\left(l_{i+1}\right)}{l_i}\leq\ell\cdot2\cdot\frac{n^{1+o(1)}}{n/2^{\log^2\log n}}=\frac{\log n}{\log^2\log n}\cdot2\cdot n^{o(1)}\cdot2^{\log^2\log n}=n^{o(1)}.$$

In summary, every execution of the Write protocol performs (sends, respectively)

$$O\left(\mathsf{B}\cdot2^{\log^2\log n}\cdot\frac{\log n}{\log\log n}\right)+n^{o(1)}=n^{o(1)}$$

bit operations (bits, respectively), for a large enough n.

Finally, regarding client storage, emulating the LDC decoder requires storing $\mathsf{space}\left(k\right)=O\left(1\right)$ size-B blocks. By the choice of B, running $\mathsf{OblSetup}$ (as part of reshuffle operations) requires the client to store only $O\left(1\right)$ size-B blocks. Operations on mDB require (by Theorem 3.7) storing $O\left(\log N\log\log N\right)$ size-mB blocks which corresponds to a constant number of size-B blocks. □

Security Analysis: Proof of Claim 4.5. We show that Construction 4.4 is secure.

Proof of Claim 4.5. The correctness of the scheme follows from the correctness of the underling RO-ORAM (with oblivious setup) and metadata ORAM schemes, and the description of Construction 4.4. Indeed, the reshuffling procedures preserve the invariant that for every level i, $\mathcal{DB}^i$ contains at most a single copy of each memory block, where upper levels contain more recent copies, so every read operation returns the correct value.

We now argue security. Let $\mathsf{DB}^0,\mathsf{DB}^1$ be two databases consisting of n size-B blocks, and let Q^0,Q^1 be two sequences of $q=\mathrm{poly}\left(\lambda\right)$ read and write operations. We proceed via a sequence of hybrids.

$\mathcal{H}_{\mathbf{0}}^{\mathbf{b}}$: Hybrid $\mathcal{H}_0^b$ is the visible access pattern $\mathsf{vAP}\left(\mathsf{DB}^b,Q^b\right)$ in an execution of the sequence Q^b on the ORAM generated for database DB^b.

$\mathcal{H}_{\mathbf{1}}^{\mathbf{b}}$: In hybrid $\mathcal{H}_1^b$, for every level $1 \leq i < \ell$, we replace all entries of $\mathcal{M}^i$ with dummy values of (e.g.,) the all-0 string. Moreover, we replace all read and write accesses to the metadata ORAM $\mathcal{MO}^i$ with dummy operations that (e.g.,) read and write the all-0 string to the first location in $\mathcal{M}^i$. (We note that the accesses to the databases $\mathcal{DB}^i$, and the RO-ORAM $\mathcal{O}^i$, remain unchanged.)

Hybrids $\mathcal{H}_0^b$ and $\mathcal{H}_1^b$ are statistically indistinguishable by the security of the underlying ORAM scheme, using a standard hybrid argument in which we replace $\mathcal{M}^i$ and the accesses to $\mathcal{MO}^i$ one level at a time.

$\mathcal{H}_{\mathbf{2}}^{\mathbf{b}}$: In hybrid $\mathcal{H}_2^b$, for every level $1 \leq i < \ell$, we initialize its RO-ORAM $\mathcal{O}^i$ (during Setup and $\mathsf{ReShuffle}, \mathsf{ReShuffle}^{\ell}$ calls) with a dummy database whose blocks all contain (e.g.,) the all-0 string. Moreover, we replace all read and write accesses to $\mathcal{O}^i$ with dummy operations that (e.g.,) read the first block in $\mathcal{O}^i$ and write the all-0 string to the first block in $\mathcal{O}^i$. (As in $\mathcal{H}_1^b$, the accesses to the databases $\mathcal{DB}^i$ remain unchanged.)

Hybrids $\mathcal{H}_1^b$ and $\mathcal{H}_2^b$ are statistically indistinguishable by the security of the underlying RO-ORAM scheme with oblivious setup, using a standard hybrid argument in which we replace the contents and accesses to $\mathcal{O}^i$ of one level at a time.

Notice that $\mathcal{H}_2^0, \mathcal{H}_2^1$ may differ only in the parts of the visible access pattern caused by the accesses to $\mathcal{DB}^i, i \in [\ell]$. Since those visible accesses are independent of the access patterns Q^0, Q^1 (respectively), or the contents of the databases $\mathsf{DB}^0, \mathsf{DB}^1$ (respectively), we conclude that $\mathcal{H}_2^0 \equiv \mathcal{H}_2^1$. □

Complexity Analysis: Proof of Claim 4.6. We now analyze the complexity of Construction 4.4, proving Claim 4.6. Notice that since $\mathsf{mB} \geq 2\log n$, each entry in $\mathcal{M}^i, i \in [\ell - 1]$ is contained in a single block of $\mathcal{MO}^i, i \in [\ell - 1]$. Notice also that for every level $i \in [\ell - 1]$, the map $\mathcal{M}^i$ contains at most $l_i \cdot 2\log n$ bits, and let $N_m := \frac{2n \log n}{\mathsf{mB}}$ denote an upper bound on $\left|\mathcal{M}^i\right|$ in terms of size-mB blocks.

Proof of Claim 4.6. We first analyze the complexity of read operations. Every execution of the Read protocol consists of the following operations:

- For every level $1 \leq i \leq \ell - 1$, a single access to the metadata ORAM $\mathcal{MO}^i$, which results in at most $\mathsf{mB} \cdot \mathsf{Ovh}_m \left(\frac{2l_i \log n}{\mathsf{mB}}\right)$ bit operations.
- For every level $1 \leq i \leq \ell$, a single access to the RO-ORAM $\mathcal{O}^i$, which results in at most $\mathsf{B} \cdot \mathsf{Ovh}_R\left(l_i\right)$ bit operations.

In total, these operations require performing

$$\mathsf{B} \cdot \mathsf{Ovh}_R\left(n\right) + \sum_{i=1}^{\ell-1} \left(\mathsf{mB} \cdot \mathsf{Ovh}_m \left(\frac{2l_i \log n}{\mathsf{mB}}\right) + \mathsf{B} \cdot \mathsf{Ovh}_R\left(l_i\right)\right)$$

bit operations.

Next, we analyze the complexity of write operations. Every execution of the Write protocol consists of writing a single block of size B to the server (writing the "dummy" level 0). In addition, each Write execution performs its "share" of the operations needed to execute the reshuffle procedures $\mathsf{ReShuffle}^{\ell}$, and $\mathsf{ReShuffle}$ for every level $1 \leq i \leq \ell - 2$. More specifically, for every level

$0 \leq i < \ell$, an execution of the Write protocol entails performing a $\frac{1}{l_i}$-fraction of the *total* number of bit operations needed to reshuffle level i into level $i+1$. We now analyze these operations.

The $\mathsf{ReShuffle}$ procedure, when applied to level i, consists of the following operations:

- Every block in levels $i, i+1$ (the total number of such blocks is at most l_{i+1}) causes:
  - Two accesses (one for reading, one for writing) to $\mathcal{DB}^{i+1}$,[7] resulting in $2\mathsf{B}$ bit operations.
  - Two accesses (one for reading, one for writing) to $\mathcal{MO}^i$, resulting in $2\mathsf{mB} \cdot \mathsf{Ovh}_m\left(\frac{2l_i \log n}{\mathsf{mB}}\right)$ bit operations.
- Every level-$(i+1)$ block (there are at most l_{i+1} such blocks) causes a single access to $\mathcal{O}^i$, resulting in $\mathsf{B} \cdot \mathsf{Ovh}_R(l_i)$ bit operations.
- Every level-i block (there are at most l_i such blocks) causes a single update of the counter, and a single access to $\mathcal{MO}^{i+1}$, causing $\log l_{i+1} + \mathsf{mB} \cdot \mathsf{Ovh}_m\left(\frac{2l_{i+1} \log n}{\mathsf{mB}}\right)$ bit operations.
- The RO-ORAMs $\mathcal{O}^j, j \in \{i, i+1\}$ are regenerated by running $\mathsf{OblSetup}_R$ on the new database $\mathcal{DB}^j, j \in \{i, i+1\}$, causing $\mathsf{T}(l_i) + \mathsf{T}(l_{i+1})$ bit operations.

The $\mathsf{ReShuffle}^\ell$ procedure causes at most this number of operations.

In summary, the update operations needed to reshuffle level $i, 0 \leq i < \ell - 1$ into level $i+1$ add a total of at most

$$\frac{l_{i+1}}{l_i} \cdot \left(2\mathsf{B} + 2\mathsf{mB} \cdot \mathsf{Ovh}_m\left(\frac{2l_i \log n}{\mathsf{mB}}\right) + \mathsf{B} \cdot \mathsf{Ovh}_R(l_i)\right) + \log l_{i+1} + \mathsf{mB} \cdot \mathsf{Ovh}_m\left(\frac{2l_{i+1} \log n}{\mathsf{mB}}\right) + \frac{\mathsf{T}(l_i) + \mathsf{T}(l_{i+1})}{l_i}$$

bit operations to each execution of the Write protocol. Therefore, each execution of the Write protocol requires performing (communicating, respectively) at most

$$\mathsf{B} + \sum_{i=0}^{\ell-1} \frac{l_{i+1}}{l_i} \cdot \left(2\mathsf{B} + 2\mathsf{mB} \cdot \mathsf{Ovh}_m\left(\frac{2l_i \log n}{\mathsf{mB}}\right) + \mathsf{B} \cdot \mathsf{Ovh}_R(l_i)\right)$$

$$+ \sum_{i=0}^{\ell-1} \left(\log l_{i+1} + \mathsf{mB} \cdot \mathsf{Ovh}_m\left(\frac{2l_{i+1} \log n}{\mathsf{mB}}\right) + \frac{\mathsf{T}(l_i) + \mathsf{T}(l_{i+1})}{l_i}\right)$$

bit operations (bits, respectively). □

References

[AFN⁺17] Ittai Abraham, Christopher W. Fletcher, Kartik Nayak, Benny Pinkas, and Ling Ren. Asymptotically tight bounds for composing ORAM with PIR. In *Public-Key Cryptography - PKC 2017 - 20th IACR International Conference on Practice and Theory in Public-Key Cryptography, Amsterdam, The Netherlands, March 28-31, 2017, Proceedings, Part I*, pages 91–120, 2017.

[AKS83] Miklós Ajtai, János Komlós, and Endre Szemerédi. An $O(n \log n)$ sorting network. In *Proceedings of the 15th Annual ACM Symposium on Theory of Computing, 25-27 April, 1983*, pages 1–9, 1983.

[7] More accurately, blocks from level i cause one access to $\mathcal{DB}^i$ and one access to $\mathcal{DB}^{i+1}$, but these operations have the same complexity since they entail reading or writing a size-B block.

[AKST14] Daniel Apon, Jonathan Katz, Elaine Shi, and Aishwarya Thiruvengadam. Verifiable oblivious storage. In *Public-Key Cryptography - PKC 2014 - 17th International Conference on Practice and Theory in Public-Key Cryptography, Buenos Aires, Argentina, March 26-28, 2014. Proceedings*, pages 131–148, 2014.

[BCP15] Elette Boyle, Kai-Min Chung, and Rafael Pass. Large-scale secure computation: Multi-party computation for (parallel) RAM programs. In *Advances in Cryptology - CRYPTO 2015 - 35th Annual Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2015, Proceedings, Part II*, pages 742–762, 2015.

[BIPW17] Elette Boyle, Yuval Ishai, Rafael Pass, and Mary Wootters. Can we access a database both locally and privately? In *Theory of Cryptography - 15th International Conference, TCC 2017, Baltimore, MD, USA, November 12-15, 2017, Proceedings, Part II*, pages 662–693, 2017.

[Blu84] Norbert Blum. A boolean function requiring $3n$ network size. *Theor. Comput. Sci.*, 28:337–345, 1984.

[BN16] Elette Boyle and Moni Naor. Is there an oblivious RAM lower bound? In *Proceedings of the 2016 ACM Conference on Innovations in Theoretical Computer Science, Cambridge, MA, USA, January 14-16, 2016*, pages 357–368, 2016.

[CFL$^+$13] Yeow Meng Chee, Tao Feng, San Ling, Huaxiong Wang, and Liang Feng Zhang. Query-efficient locally decodable codes of subexponential length. *Computational Complexity*, 22(1):159–189, 2013.

[CHR17] Ran Canetti, Justin Holmgren, and Silas Richelson. Towards doubly efficient private information retrieval. In *Theory of Cryptography - 15th International Conference, TCC 2017, Baltimore, MD, USA, November 12-15, 2017, Proceedings, Part II*, pages 694–726, 2017.

[CKW13] David Cash, Alptekin Küpçü, and Daniel Wichs. Dynamic proofs of retrievability via oblivious RAM. In *Advances in Cryptology - EUROCRYPT 2013, 32nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Athens, Greece, May 26-30, 2013. Proceedings*, pages 279–295, 2013.

[Dur64] Richard Durstenfeld. Algorithm 235: Random permutation. *Commun. ACM*, 7(7):420, 1964.

[DvDF$^+$16] Srinivas Devadas, Marten van Dijk, Christopher W. Fletcher, Ling Ren, Elaine Shi, and Daniel Wichs. Onion ORAM: A constant bandwidth blowup oblivious RAM. In *Theory of Cryptography - 13th International Conference, TCC 2016-A, Tel Aviv, Israel, January 10-13, 2016, Proceedings, Part II*, pages 145–174, 2016.

[Efr09] Klim Efremenko. 3-query locally decodable codes of subexponential length. In *Proceedings of the 41st Annual ACM Symposium on Theory of Computing, STOC 2009, Bethesda, MD, USA, May 31 - June 2, 2009*, pages 39–44, 2009.

[FGHK16] Magnus Gausdal Find, Alexander Golovnev, Edward A. Hirsch, and Alexander S. Kulikov. A better-than-$3n$ lower bound for the circuit complexity of an explicit function. In *IEEE 57th Annual Symposium on Foundations of Computer Science, FOCS 2016, 9-11 October 2016, Hyatt Regency, New Brunswick, New Jersey, USA*, pages 89–98, 2016.

[FNR$^+$15] Christopher W. Fletcher, Muhammad Naveed, Ling Ren, Elaine Shi, and Emil Stefanov. Bucket ORAM: single online roundtrip, constant bandwidth oblivious RAM. *IACR Cryptology ePrint Archive*, 2015:1065, 2015.

[GGH$^+$13] Craig Gentry, Kenny A. Goldman, Shai Halevi, Charanjit S. Jutla, Mariana Raykova, and Daniel Wichs. Optimizing ORAM and using it efficiently for secure computation. In *Privacy Enhancing Technologies - 13th International Symposium, PETS 2013, Bloomington, IN, USA, July 10-12, 2013. Proceedings*, pages 1–18, 2013.

[GHJR15] Craig Gentry, Shai Halevi, Charanjit S. Jutla, and Mariana Raykova. Private database access with HE-over-ORAM architecture. In *Applied Cryptography and Network Security - 13th International Conference, ACNS 2015, New York, NY, USA, June 2-5, 2015, Revised Selected Papers*, pages 172–191, 2015.

[GKK$^+$12] S. Dov Gordon, Jonathan Katz, Vladimir Kolesnikov, Fernando Krell, Tal Malkin, Mariana Raykova, and Yevgeniy Vahlis. Secure two-party computation in sublinear (amortized) time. In *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012*, pages 513–524, 2012.

[GMOT12] Michael T. Goodrich, Michael Mitzenmacher, Olga Ohrimenko, and Roberto Tamassia. Privacy-preserving group data access via stateless oblivious RAM simulation. In *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2012, Kyoto, Japan, January 17-19, 2012*, pages 157–167, 2012.

[GO96] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious RAMs. *J. ACM*, 43(3):431–473, 1996.

[Gol87] Oded Goldreich. Towards a theory of software protection and simulation by oblivious rams. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing, 1987, New York, New York, USA*, pages 182–194, 1987.

[Goo14] Michael T. Goodrich. Zig-zag sort: a simple deterministic data-oblivious sorting algorithm running in $O(n \log n)$ time. In *Symposium on Theory of Computing, STOC 2014, New York, NY, USA, May 31 - June 03, 2014*, pages 684–693, 2014.

[HO08] Brett Hemenway and Rafail Ostrovsky. Public-key locally-decodable codes. In *Advances in Cryptology - CRYPTO 2008, 28th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2008. Proceedings*, pages 126–143, 2008.

[HOSW11] Brett Hemenway, Rafail Ostrovsky, Martin J. Strauss, and Mary Wootters. Public key locally decodable codes with short keys. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques - 14th International Workshop, APPROX 2011, and 15th International Workshop, RANDOM 2011, Princeton, NJ, USA, August 17-19, 2011. Proceedings*, pages 605–615, 2011.

[HOWW18] Ariel Hamlin, Rafail Ostrovsky, Mor Weiss, and Daniel Wichs. Private anonymous data access. *IACR Cryptology ePrint Archive*, 2018:363, 2018.

[IM02] Kazuo Iwama and Hiroki Morizumi. An explicit lower bound of $5n - o(n)$ for boolean circuits. In *Mathematical Foundations of Computer Science 2002, 27th International Symposium, MFCS 2002, Warsaw, Poland, August 26-30, 2002, Proceedings*, pages 353–364, 2002.

[IS10] Toshiya Itoh and Yasuhiro Suzuki. Improved constructions for query-efficient locally decodable codes of subexponential length. *IEICE Transactions*, 93-D(2):263–270, 2010.

[KLO12] Eyal Kushilevitz, Steve Lu, and Rafail Ostrovsky. On the (in)security of hash-based oblivious RAM and a new balancing scheme. In *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2012, Kyoto, Japan, January 17-19, 2012*, pages 143–156, 2012.

[KM18] Eyal Kushilevitz and Tamer Mour. Sub-logarithmic distributed oblivious RAM with small block size. *CoRR*, abs/1802.05145, 2018.

[KS14] Marcel Keller and Peter Scholl. Efficient, oblivious data structures for MPC. In *Advances in Cryptology - ASIACRYPT 2014 - 20th International Conference on the Theory and Application of Cryptology and Information Security, Kaoshiung, Taiwan, R.O.C., December 7-11, 2014, Proceedings, Part II*, pages 506–525, 2014.

[KT00] Jonathan Katz and Luca Trevisan. On the efficiency of local decoding procedures for error-correcting codes. In *Proceedings of the Thirty-Second Annual ACM Symposium on Theory of Computing, May 21-23, 2000, Portland, OR, USA*, pages 80–86, 2000.

[LHS$^+$14] Chang Liu, Yan Huang, Elaine Shi, Jonathan Katz, and Michael W. Hicks. Automating efficient RAM-model secure computation. In *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014*, pages 623–638, 2014.

[LN18] Kasper Green Larsen and Jesper Buus Nielsen. Yes, there is an oblivious RAM lower bound! Cryptology ePrint Archive, Report 2018/423 (to appear in CRYPTO‘18), 2018. https://eprint.iacr.org/2018/423.

[LO13] Steve Lu and Rafail Ostrovsky. Distributed oblivious RAM for secure two-party computation. In *Theory of Cryptography - 10th Theory of Cryptography Conference, TCC 2013, Tokyo, Japan, March 3-6, 2013. Proceedings*, pages 377–396, 2013.

[LPM$^+$13] Jacob R. Lorch, Bryan Parno, James W. Mickens, Mariana Raykova, and Joshua Schiffman. Shroud: ensuring private access to large-scale data in the data center. In *Proceedings of the 11th USENIX conference on File and Storage Technologies, FAST 2013, San Jose, CA, USA, February 12-15, 2013*, pages 199–214, 2013.

[MBC14] Travis Mayberry, Erik-Oliver Blass, and Agnes Hui Chan. Efficient private file retrieval by combining ORAM and PIR. In *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014*, 2014.

[MLS$^+$13] Martin Maas, Eric Love, Emil Stefanov, Mohit Tiwari, Elaine Shi, Krste Asanovic, John Kubiatowicz, and Dawn Song. PHANTOM: practical oblivious computation in a secure processor. In *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, pages 311–324, 2013.

[OS97] Rafail Ostrovsky and Victor Shoup. Private information storage (extended abstract). In *Proceedings of the Twenty-Ninth Annual ACM Symposium on the Theory of Computing, El Paso, Texas, USA, May 4-6, 1997*, pages 294–303, 1997.

[Ost90] Rafail Ostrovsky. Efficient computation on oblivious RAMs. In *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing, May 13-17, 1990, Baltimore, Maryland, USA*, pages 514–523, 1990.

[PD06] Mihai Patrascu and Erik D. Demaine. Logarithmic lower bounds in the cell-probe model. *SIAM J. Comput.*, 35(4):932–963, 2006.

[Rag07] Prasad Raghavendra. A note on Yekhanin's locally decodable codes. *Electronic Colloquium on Computational Complexity (ECCC)*, 14(016), 2007.

[RFK$^+$15] Ling Ren, Christopher W. Fletcher, Albert Kwon, Emil Stefanov, Elaine Shi, Marten van Dijk, and Srinivas Devadas. Constants count: Practical improvements to oblivious RAM. In *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015.*, pages 415–430, 2015.

[SCSL11] Elaine Shi, T.-H. Hubert Chan, Emil Stefanov, and Mingfei Li. Oblivious RAM with $O((\log N)^3)$ worst-case cost. In *Advances in Cryptology - ASIACRYPT 2011 - 17th International Conference on the Theory and Application of Cryptology and Information Security, Seoul, South Korea, December 4-8, 2011. Proceedings*, pages 197–214, 2011.

[SS13] Emil Stefanov and Elaine Shi. ObliviStore: High performance oblivious distributed cloud data store. In *20th Annual Network and Distributed System Security Symposium, NDSS 2013, San Diego, California, USA, February 24-27, 2013*, 2013.

[SvDS$^+$13] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher W. Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: an extremely simple oblivious RAM protocol. In *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, pages 299–310, 2013.

[WCS15] Xiao Wang, T.-H. Hubert Chan, and Elaine Shi. Circuit ORAM: on tightness of the Goldreich-Ostrovsky lower bound. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-6, 2015*, pages 850–861, 2015.

[WGK18] Xiao Wang, S. Dov Gordon, and Jonathan Katz. Simple and efficient two-server ORAM. *IACR Cryptology ePrint Archive*, 2018:5, 2018.

[WHC$^+$14] Xiao Shaun Wang, Yan Huang, T.-H. Hubert Chan, Abhi Shelat, and Elaine Shi. SCORAM: oblivious RAM for secure computation. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, pages 191–202, 2014.

[Woo07] David P. Woodruff. New lower bounds for general locally decodable codes. *Electronic Colloquium on Computational Complexity (ECCC)*, 14(006), 2007.

[WS12] Peter Williams and Radu Sion. Single round access privacy on outsourced storage. In *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012*, pages 293–304, 2012.

[Yek07] Sergey Yekhanin. Towards 3-query locally decodable codes of subexponential length. In *Proceedings of the 39th Annual ACM Symposium on Theory of Computing, San Diego, California, USA, June 11-13, 2007*, pages 266–274, 2007.

[YFR$^+$13] Xiangyao Yu, Christopher W. Fletcher, Ling Ren, Marten van Dijk, and Srinivas Devadas. Generalized external interaction with tamper-resistant hardware with bounded information leakage. In *CCSW'13, Proceedings of the 2013 ACM Cloud Computing Security Workshop, Co-located with CCS 2013, Berlin, Germany, November 4, 2013*, pages 23–34, 2013.

[ZMZQ16] Jinsheng Zhang, Qiumao Ma, Wensheng Zhang, and Daji Qiao. MSKT-ORAM: A constant bandwidth ORAM without homomorphic encryption. *IACR Cryptology ePrint Archive*, 2016:882, 2016.