Is There an Oblivious RAM Lower Bound for Online Reads?

Mor Weiss(✉) and Daniel Wichs

Department of Computer Science, Northeastern University, Boston, MA, USA
m.weiss@northeastern.edu, wichs@ccs.neu.edu

Abstract. 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.

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 his data stored on untrusted storage, while entirely hiding the access pattern, i.e., which operations were performed and at which locations. More precisely, we think of the client's data as "logical memory" which the ORAM scheme encodes and stores in "physical memory". Whenever the client wants

A. Beimel and S. Dziembowski (Eds.): TCC 2018, LNCS 11240, pp. 603–635, 2018.
https://doi.org/10.1007/978-3-030-03810-6_22

to read or write to logical memory, the ORAM scheme translates this operation into several accesses to the physical memory. Security ensures that for any two (equal length) sequences of access to logical memory, the resultant distributions over the physical accesses performed by the ORAM 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 logical memory. 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; and works [PPRY] achieving $O(\log n \log\log n)$ overhead for $\Omega(\log n)$ block size, assuming one-way functions. 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]. However, in this work we focus on the standard ORAM setting with a single server and no server-side computation.

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 logical memory. We stress that the scheme is read-only in the sense that it only supports programs that do not write to the *logical* memory. However, to emulate such programs in the ORAM, the client might write to the *physical* memory stored on the server. 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. In all our constructions, the server is only used as remote storage, and does not perform any computations.

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 logical memory of size-n, 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 (Informal statement of Corollary 1). *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 memory of size n and blocks of size* $\mathrm{poly}\log n$*, with $O(1)$ client storage and $O(\log \log n)$ overhead.*

In Sect. 3, we also show a read-only ORAM scheme with $o(\log n)$ overhead based on milder assumptions – concretely, smooth LDCs with $O(\log \log n)$ query complexity, and the existence of sorting circuits of size $o\left(\frac{n \log n}{\log^2 \log n}\right)$; see Corollary 2. We note that under the (strong) assumption that the LDC has linear-size codewords, our constructions achieve linear-size server storage. We also 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, as well as the existence of One-Way Functions (OWFs). (We elaborate on why OWFs are needed in Sect. 1.2.) 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 2 (Informal statement of Theorem 7). *Assume the existence of OWFs, as well as LDCs and sorting circuits as in Theorem 1. Then for every constant $\epsilon \in (0, 1)$, there exists a constant $\gamma \in (0, 1)$ such that if LDC encoding* *requires* $n^{1+\gamma}$ *operations then there is a computationally-secure ORAM scheme for memory of size* n *and blocks of size* $\mathrm{poly}\log n$ *with* $O(1)$ *client storage,* $O(\log\log n)$ *read overhead, and* $O(n^{\epsilon})$ *write overhead.*

Similar to the read-only setting, we also instantiate (Sect. 4, Theorem 8) 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 3 (Informal statement of Theorem 9). *Assume the existence of OWFs, as well as LDCs and sorting circuits as in Theorem 1, where LDC encoding requires* $n^{1+o(1)}$ *operations. Then there exists a computationally-secure ORAM scheme for memory of size* n *and blocks of size* $\mathrm{poly}\log n$ *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). More specifically, words, which are the basic unit of physical memory on the server, consist of w bits; and blocks, which are the basic unit of logical memory on the client, consist of B words. We measure the overhead as the number of words the client accesses on the server to read or write to a single logical block, divided by B. We note that it is generally easier to construct schemes with *smaller* word size. (Indeed, it allows the client more fine-grained access to the physical memory; a larger word size might cause the client to access unneeded bits on the server, simply because they are part of a word containing bits that do interest the client.) Consequently, we would generally like to support *larger* word size, ideally having words and blocks of equal size. Our constructions can handle any word size,[1] as long as blocks are poly-logarithmically larger (for a sufficiently large poylogarithmic factor). A similar differentiation between block and word size was used in some previous works as well (e.g., to get $O(\log N)$ overhead in Path ORAM [SvDS+13]).

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 Sects. 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 Sect. 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].

[1] Similar to previous works (e.g., [SCSL11, SvDS+13, SS13]), we assume words are of at least logarithmic size.

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.

A Note on the Connection to Private Information Retrieval (PIR) and Doubly-Efficient PIR (DEPIR). The notions of PIR and DEPIR, which support reads from memory stored on a remote server, are closely related to read-only ORAM, but differ from it significantly in some respects. We now discuss these primitives in more detail. In a (single-server) PIR scheme [KO97], there is no initial setup, and anybody can run a protocol with the server to retrieve an arbitrary location in the logical memory. The server is *not* used solely as remote storage, and in fact the main goal, which is to minimize the communication between the client and server, *inherently* requires the server to perform computations. One additional significant difference from ORAM is that the PIR privacy guarantee *inherently* requires the server runtime to be linear in the size of the logical memory, whereas a main ORAM goal is to have the server touch only a sublinear number of blocks (which the client reads from it to retrieve the block he is interested in). In a DEPIR scheme [BIM00, BIPW17, CHR17], there is a setup phase (as in ORAM), following which the server(s) stores an encoded version of the logical memory, and the logical memory can be accessed either with no key (in multi-server DEPIR [BIM00]), with a public key (in public-key DEPIR [BIPW17]) or with a secret key (in secret-key DEPIR [BIPW17, CHR17]). First proposed by Beimel, Ishai and Malkin [BIM00], who showed how to construct information-theoretic DEPIR schemes in the multi-server setting (i.e., with several non-colluding servers), two recent works [BIPW17, CHR17] give the first evidence that this notion may be achievable in the single-server setting. These works achieve sublinear server runtime, with a server that is only used as remote storage. Thus, these single-server DEPIR schemes satisfy all the required properties of a RO-ORAM scheme, with the added "bonus" of having a stateless server (namely, whose internal memory does not change throughout the execution of the scheme). However, these (secret-key) constructions are based on new, previously unstudied, computational hardness assumptions relating to Reed-Muller codes, and the public-key DEPIR scheme of [BIPW17] additionally requires a heuristic use of obfuscation. Unfortunately, both of the above assumptions are non-standard, poorly understood, and not commonly accepted. Additionally, these constructions do not achieve $o(\log n)$ overhead (at least not with polynomial server storage).

A Note on Statistical vs. Computational Security. Our RO-ORAM achieves *statistical* security under the assumption that the server does not see the memory *contents*, namely the server only sees which memory locations are accessed. Hiding memory contents from the server can be generically achieved by encrypting the logical memory, in which case security holds against *computationally-bounded* servers. We note that our ORAM scheme supporting writes requires encrypting the logical memory *even if the server does not see the memory contents*. Consequently, our ORAM with writes scheme achieve computational security even in the setting where the server does not see the memory contents. Alternatively, our construction can achieve statistical security if the underlying LDC has the additional property that the memory accesses during encoding are independent of the data. (This property is satisfied by, e.g., linear codes.) We elaborate on this further in Sects. 3.1 and 4.

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.

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 memory. 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 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 at most $M/2k$, 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 logical memory is read-only, the refreshing operations can be spread-out across the $M/2k$ read operations of the epoch.

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 logical memory on the server in a sequence of ℓ levels of increasing size, each containing an RO-ORAM.[2] 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 Sect. 4 for details.

We note that our construction differs from Hierarchical ORAM in two main points. First, in Hierarchical ORAM level i is reshuffled into level $i + 1$ every l_i read *or* write *operations*, whereas in our scheme only write operations are "counted" towards reshuffle (in that respect, read operations are "free"). This is because the data is stored in each level using an RO-ORAM which already guarantees privacy for read operations. Second, Hierarchical ORAM uses $\Omega(\log n)$ levels, whereas to preserve $o(\log n)$ read overhead, we must use $o(\log n)$ levels. In particular, the ratio between consecutive levels in our scheme is no longer constant, leading to a higher reshuffle cost (which is the reason write operations have higher overhead in our scheme).

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. Recall that words, the basic unit of physical memory on the server, consist of w bits; and blocks, the basic unit of logical memory on the client, consist of B words. The client may *locally* perform bit operations on the bit representation of blocks, but can only access full words on the server. We will usually measure complexity in terms of logical blocks (namely, in terms of the basic memory unit on the client). More specifically, unless explicitly stated otherwise, client and server storage are measured as the number of *blocks* they store (even though the basic storage unit on the server side is a word), and overhead measures the number of blocks one needs to read or write to implement a read or write operation on a single block. Formally:

[2] This is reminiscent of a construction of [OS97], which also instantiated the levels of a hierarchical ORAM with a primitive guaranteeing read privacy (specifically, they use PIR). However, our goals, and the details of our construction, differs significantly from [OS97].

Definition 1 (Overhead). *For a block size* B *and input length* n*, we say that a protocol between client* C *and server* S *has overhead* Ovh *for a function* $\mathsf{Ovh} : \mathbb{N} \to \mathbb{N}$*, if implementing a* read *or* write *operation on a single logical memory block requires the client to access* $\mathsf{B} \cdot \mathsf{Ovh}(n)$ *words on the server.*

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 (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* $(\mathsf{Enc}, \mathit{Query}, \mathsf{Dec})$ *of PPT algorithms with the following properties.*

– ***Syntax.*** Enc *is given a message* $\mathit{msg} \in \Sigma^n$ *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* $\mathit{msg} \in \Sigma^n$*, and every index* $\ell \in [n]$*,*

$$\Pr\left[\mathbf{r} \leftarrow \mathit{Query}(\ell) \ : \ \mathsf{Dec}\left(\mathsf{Enc}\left(\mathit{msg}\right)_{\mathbf{r}}\right) = \mathit{msg}_{\ell}\right] = 1.$$

– ***Smoothness.*** *For every index* $\ell \in [n]$*, every query in the output of* $\mathit{Query}(\ell)$ *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 its input, such that the access patterns of the algorithm on any two inputs 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 $\{C(n, \mathsf{B})\}_{n,\mathsf{B}}$ such that each $C(n, \mathsf{B})$ takes as input n size-B tagged blocks, and outputs the blocks in sorted order according to their tags.)

Definition 3 (Oblivious-Access Sort Algorithm, [BN16]**).** *An* Oblivious-Access Sort *algorithm for input size n and block size* B*, with overhead* $\mathsf{Ovh}_{\mathsf{Sort}}(n, \mathsf{B})$*, is a (possibly randomized) algorithm* Sort *run by a client C on an input stored remotely on a server S, with the following properties:*

- ***Operation:*** *The input consists of n tagged blocks which are represented as length-B bit strings (the tag is a substring of the block) and stored on the server.*[3] *The client can perform local bit operations, but can only read and write full blocks from the server.*
- ***Overhead:*** *The overhead of* Sort *is* $\mathsf{Ovh}_{\mathsf{Sort}}(n, \mathsf{B})$.
- ***Correctness:*** *With overwhelming probability in n, at the end of the algorithm the server stores the blocks in sorted order according to their tags.*
- ***Oblivious Access:*** *For a logical memory* DB *consisting of n blocks of size* B*, let* $\mathsf{AP}_{n,\mathsf{B}}(\mathsf{Sort}, \mathsf{DB})$ *denote the random variable consisting of the list of addresses accessed in a random execution of the algorithm* Sort *on* DB*. Then for every pair* $\mathsf{DB}, \mathsf{DB}'$ *of inputs with n size-*B *blocks,* $\mathsf{AP}_{n,\mathsf{B}}(\mathsf{Sort}, \mathsf{DB}) \approx^s \mathsf{AP}_{n,\mathsf{B}}(\mathsf{Sort}, \mathsf{DB}')$*, where $\approx^s$ denotes* $\mathsf{negl}(n)$ *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 4 (Oblivious-access sort from sorting circuits, [BN16]**).** *If there exist boolean sorting circuits* $\{C(n, \mathsf{B})\}_{n,\mathsf{B}}$ *of size* $s(n, \mathsf{B})$*, then there exists an oblivious-access sort algorithm for n distinct elements with $O(1)$ client storage,* $O\left(n \cdot \log \mathsf{B} + s\left(\frac{2n}{\mathsf{B}}, \mathsf{B}\right)\right)$ *overhead, and* $e^{-n^{\Omega(1)}}$ *probability of error.*

Remark on the Existence of Oblivious-Access Sort Algorithms with Small Overhead. We note that for blocks of poly-logarithmic size $\mathsf{B} = \operatorname{poly}\log n$, the existence of sorting circuits of size $s(n, \mathsf{B}) = O(n \cdot \mathsf{B} \cdot \log \log n)$ guarantees (through Theorem 4) the existence of oblivious-access sort algorithms with $O(n \cdot \log \log n)$ overhead.

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 input, and outputs them in sorted order. We note that a simple information-theoretic lower bound of $\Omega(n \log n)$ 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.

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

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*.

Notation 1 (Access pattern)**.** A length-q *access pattern* Q consists of a list $(\mathsf{op}_l, \mathsf{val}_l, \mathsf{addr}_l)_{1 \leq l \leq q}$ 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 4 (Oblivious RAM (ORAM)). *An* Oblivious RAM (ORAM) *scheme with block size* B *consists of procedures* $(\mathit{Setup}, \mathit{Read}, \mathit{Write})$*, with the following syntax:*

– $\mathit{Setup}(1^\lambda, \mathit{DB})$ *is a function that takes as input a security parameter* λ*, and a logical memory* $\mathit{DB} \in \left(\{0,1\}^{\mathsf{B}}\right)^n$*, 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* $|\mathit{DB}|$*.*
– Read *is a protocol between the server* S *and the client* C*. The client holds as input an address* $\mathit{addr} \in [n]$ *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* $\mathit{addr} \in [n]$*, 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*.*

Throughout the execution of the Read *and* Write *protocols, the server is used only as remote storage, and does not perform any computations.*

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 logical memory in the corresponding* read *operation, if the prefix of* V *performed before the* Read *protocol was performed directly on the logical memory.*
– ***Security:*** *For a logical memory* DB*, and an access pattern* Q*, let* $\mathit{AP}(\mathit{DB}, Q)$ *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* $\mathit{DB}^0, \mathit{DB}^1 \in \left(\{0,1\}^{\mathsf{B}}\right)^n$ *of inputs, and any pair* $Q^0 = \left(\mathit{op}_l, \mathsf{val}_l^0, \mathit{addr}_l^0\right)_{1 \leq l \leq q}, Q^1 = \left(\mathit{op}_l, \mathsf{val}_l^1, \mathit{addr}_l^1\right)_{1 \leq l \leq q}$ *of access patterns of length* $q = \mathrm{poly}(\lambda)$*,* $\mathit{AP}\left(\mathit{DB}^0, Q^0\right) \approx^s \mathit{AP}\left(\mathit{DB}^1, Q^1\right)$*, where* $\approx^s$ *denotes* $\mathsf{negl}(\lambda)$ *statistical distance.*
If $\mathit{AP}\left(\mathit{DB}^0, Q^0\right), \mathit{AP}\left(\mathit{DB}^1, Q^1\right)$ *are only* computationally *indistinguishable, then we say the scheme is computationally secure.*

Definition 4 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 on Hiding the Type of Operation. Notice that Definition 4 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 Physical Memory Contents. The security property of Definition 4 implicitly assumes that the server does not see the *contents* of the physical memory: if the server is allowed to see it, he might be able to learn some non-trivial information regarding the access pattern, and thus violate the security property. As noted in Sect. 1.1, hiding the physical memory contents from the server can be achieved by encrypting the physical memory blocks, 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 (which also allows for cleaner constructions).

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 5 (Read-Only Oblivious RAM (RO-ORAM)). *A* Read-Only Oblivious RAM (RO-ORAM) *scheme consists of procedures* (Setup, Read) *with the same syntax as in Definition 4, 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 5. *Suppose there exist:*

- (k, n, M)*-smooth LDCs with* $M = \mathrm{poly}(n)$.
- *An oblivious-access sort algorithm* Sort *with* $s(n, \mathsf{B})$ *overhead for input size* n *and block size* B.

Then there exists an RO-ORAM scheme for logical memory 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* $k + \frac{2k^2}{M} \cdot s(M, \mathsf{B}) + O(1)$ *overhead, and* $O(k)$ *client storage.*

Theorem 1 now follows from Theorem 5 (using also Theorem 4) for an appropriate instantiation of the sorting algorithm and LDC.

Corollary 1 (RO-ORAM, "dream" parameters; formal statement of Theorem 1). *Suppose there exist:*

- (k, n, M)*-smooth LDCs with* $k = O(1)$ *and* $M = \text{poly}(n)$*.*
- *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 an RO-ORAM scheme for logical memory of size n *and blocks of size* $\Omega(\lambda \cdot \log^4 n)$ *with* $O(\log \log n)$ *overhead, and* $O(1)$ *client storage.*

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

Corollary 2 (RO-ORAM, milder parameters). *Suppose there exist:*

- (k, n, M)*-smooth LDCs with* $k = \text{poly} \log \log n$ *and* $M = \text{poly}(n)$*.*
- *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 an RO-ORAM scheme for memory of size n *and blocks of size* $\Omega(\lambda \cdot \log^4 n)$ *with* $= o(\log n)$ *overhead, and* $\text{poly} \log \log n$ *client storage.*

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 logical memory, 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 overhead of the oblivious-access sort algorithm.

Construction 1 (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 $(\mathsf{Setup}_{\mathrm{in}}, \mathsf{Read}_{\mathrm{in}}, \mathsf{Write}_{\mathrm{in}})$.

The scheme consists of the following procedures:

- **$\mathsf{Setup}(1^\lambda, \mathsf{DB})$:** 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.
  1. Counter initialization. Initializes a *step counter* $\mathsf{count} = 0$.
  2. Data storage generation.
     (a) Generate the codeword $\widetilde{\mathsf{DB}} = \mathsf{Enc}_{\mathsf{LDC}}(\mathsf{DB})$ with $\widetilde{\mathsf{DB}} \in \Sigma^M$.
     (b) For every $1 \le i \le k$:
       - Generate a random permutation $P^i : [M] \to [M]$.
       - Let $\widetilde{\mathsf{DB}}^i \in \Sigma^M$ be a permuted version of the codeword which satisfies $\widetilde{\mathsf{DB}}^i_{P^i(j)} = \widetilde{\mathsf{DB}}_j$ for all $j \in [M]$.
  3. Metadata storage generation.
     (a) For every $1 \le i \le k$:
       - Initialize a length-M bit-array $\mathsf{Queried}^i$ to $\mathbf{0}$.
       - Initialize a length-M array Perm^i over $\{0,1\}^{\log M}$ such that $\mathsf{Perm}^i(j) = P^i(j)$.

     (b) Let mDB denote the logical memory obtained by concatenating $\mathsf{Queried}^1, \ldots, \mathsf{Queried}^k$ and $\mathsf{Perm}^1, \ldots, \mathsf{Perm}^k$. Run $(\mathsf{ck}_m, \mathsf{st}_m) \leftarrow \mathsf{Setup}_{\mathrm{in}}\left(1^\lambda, \mathsf{mDB}\right)$ to obtain the client key and server state for the metadata ORAM.
  4. Output. The long-term client key $\mathsf{ck} = \mathsf{ck}_m$ consists of the client key for the metadata ORAM. The server state $\mathsf{st}_S = \left(\left\{\widetilde{\mathsf{DB}}^i : i \in [k]\right\}, \mathsf{st}_m, \mathsf{count}\right)$ contains the k permuted codewords, the server state for the metadata ORAM, and the step counter.

- **The Read protocol.** To read the logical memory block at location $\mathsf{addr} \in [n]$ from the server S, the client C with key $\mathsf{ck} = \mathsf{ck}_m$ operates as follows, where in all executions of the $\mathsf{Read}_{\mathrm{in}}$ or $\mathsf{Write}_{\mathrm{in}}$ 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.
  1. 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 $1 \le i \le k$, run the $\mathsf{Read}_{\mathrm{in}}$ 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.)

2. 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.
3. Decoding logical memory blocks. Read $\widetilde{\mathsf{DB}}^1_{q'_1}, \ldots, \widetilde{\mathsf{DB}}^k_{q'_k}$ from the server, and set the client output to $\mathsf{Dec}_{\mathsf{LDC}}\left(\widetilde{\mathsf{DB}}^1_{q'_1}, \ldots, \widetilde{\mathsf{DB}}^k_{q'_k}\right)$.
4. Updating counter and server state. Let $\ell = \frac{M}{2k}$. Read count from the server.
   - 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 $\mathbf{0}$ 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 client performs 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 1.

Proposition 1 (ORAM security). *Assuming the security of all of the building blocks, Construction 1 is a secure RO-ORAM scheme.*

Proposition 2 (ORAM overhead). *Assume that:*

– *The logical memory DB has block size* B*, and the metadata ORAM has block size* mB*, satisfying* $\mathsf{B} > \mathsf{mB} \geq \log M$*.*
– *The metadata ORAM has overhead* $\mathsf{Ovh}(N)$ *for memory of size* N*.*
– *The oblivious-access sort algorithm has* $\mathsf{Ovh}_{\mathsf{Sort}}(n, \mathsf{B})$ *overhead when operating on inputs consisting of* n *size-*B *blocks.*

Then every execution of the Read *protocol in Construction 1 requires accessing*

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

words on the server.

Claims Imply Theorem. To prove Theorem 5, we instantiate the metadata ORAM of Construction 1 with the following variant of path ORAM [SvDS+13]:

Theorem 6 (Statistical ORAM with polylog overhead, implicit in [SvDS+13]). *Let λ be a security parameter. Then there exists a statistical*

ORAM scheme with $\mathsf{negl}(\lambda)$ *error for logical memory consisting of* N *blocks of size* $\mathsf{mB} = \log^2 N \log\log N$ *with* $O(\log N)$ *overhead, in which the client stores* $O(\log N(\lambda + \log\log N))$ *blocks.*

Moreover, initializing the scheme requires accessing $O(N \cdot \mathsf{mB})$ *words, and the server stores* $O(N)$ *blocks.*

Proof of Theorem 5. Security follows directly from Proposition 1 since (as noted in Sect. 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 overhead of the construction, let $N_m = k(M + M \log M)$ denote the size (in bits) of the metadata ORAM. Substituting $\mathsf{mB} = \log^2 N_m \log\log N_m$, and $\mathsf{Ovh}(N) = O(\log N)$ (according to Theorem 6), Proposition 2 guarantees that every execution of the Read protocol requires accessing

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

words on the server. The first summand can be upped bounded by

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

For $\mathsf{B} = \Omega\left(\lambda \cdot k^2 \cdot \log^3(kn) \log^7 \log(kn)\right)$ (as in the theorem statement) with a sufficiently large constant in the $\Omega(\cdot)$ notation, and since $M = \mathrm{poly}(n)$, this corresponds to accessing $O(\mathsf{B})$ words on the server, so the overhead is $k + \frac{2k^2}{M} \cdot s(M, \mathsf{B}) + O(1)$.

Finally, regarding client storage, emulating the LDC decoder requires storing k size-B blocks (i.e, the answers to the decoder queries). Operations on mDB require (by Theorem 6) storing $O(\log N_m(\lambda + \log\log N_m))$ size-mB blocks which corresponds to a constant number of size-B blocks. □

Security Analysis: Proof of Proposition 1. The proof of Proposition 1 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 1. *Let* $k, M \in \mathbb{N}$, *and let* $X = (X_1, \ldots, X_k)$ *be a random variable over* $[M]^k$ *such that for every* $1 \leq i \leq k$, X_i *is uniformly distributed over* $[M]$. *Let* $S_1, \ldots, S_k \subseteq [M]$ *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* $(x_1, \ldots, x_k)$ *such that* $x_i \notin S_i$ *for every* $1 \leq i \leq k$.

In particular, if $\ell = \frac{M}{2k}$ *and* $l = \Omega(\lambda)$ *then except with probability* $\mathsf{negl}(\lambda)$, *there exists a sample* $(x_1, \ldots, x_k)$ *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 $\Pr[x_i \in S_i] \leq \frac{\ell}{M}$, 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 Proposition 1.

Proof of Proposition 1. 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 logical memories consisting of n size-B blocks, and let R^0, R^1 be two sequences of read operations of length $q = \text{poly}(\lambda)$. 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 1, this happens with overwhelming probability.

$\mathcal{H}_\mathbf{0}^\mathbf{b}$: Hybrid $\mathcal{H}_0^b$ is the access pattern $\mathsf{AP}\left(\mathsf{DB}^b, R^b\right)$ in an execution of read sequence R^b on the RO-ORAM generated for logical memory 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 codewords 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 codeword 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 codeword 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 Proposition 2. We now analyze the complexity of Construction 1, proving Proposition 2. Notice that since $\mathsf{mB} \geq \log M$, an image of any random permutation $P^i : [M] \rightarrow [M]$ 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. Recall that a word (i.e., the basic unit of the physical memory stored on the server) consists of w bits.

Proof of Proposition 2. 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. Reading each bit requires reading a different block from mDB, which requires accessing $k\lambda \cdot \mathsf{mB} \cdot \mathsf{Ovh}\left(N_m\right)$ words on the server.
- Reading k images from $\mathsf{Perm}^1, \ldots, \mathsf{Perm}^k$ to permute the chosen decoding queries. This requires reading k blocks from mDB, which requires accessing $k \cdot \mathsf{mB} \cdot \mathsf{Ovh}\left(N_m\right)$ words on the server.
- Reading k blocks from the permuted codewords $\widetilde{\mathsf{DB}}^1, \ldots, \widetilde{\mathsf{DB}}^k$ to answer the decoder queries, which requires accessing $\frac{\mathsf{B}}{w} \cdot k$ words on the server.
- 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 accessing $k \cdot \mathsf{mB} \cdot \mathsf{Ovh}\left(N_m\right)$ words on the server. (This operation is only performed when $\mathsf{count} < \ell - 1$, but counting it in *every* Read execution will not increase the overall asymptotic complexity.)
- Updating the counter, which requires accessing $\frac{\lambda}{w}$ words on the server.

In total, these operations require accessing $O\left(k\lambda\right) \cdot \mathsf{mB} \cdot \mathsf{Ovh}\left(N_m\right) + k \cdot \frac{\mathsf{B}}{w}$ words on the server.

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 accessing $k \cdot M \cdot \left(\frac{1}{\mathsf{mB}} + 4\right) \cdot \mathsf{mB} \cdot \mathsf{Ovh}\left(N_m\right)$ words on the server.
- Running k executions of Sort on an input of M blocks of size B to re-permute the codeword copies, which requires accessing $k \cdot \mathsf{Ovh}_{\mathsf{Sort}}\left(M, \mathsf{B}\right)$ words on the server.

So these update operations require accessing $O\left(k^2\right) \cdot \mathsf{mB} \cdot \mathsf{Ovh}\left(N_m\right) + \frac{2k^2}{M} \cdot \mathsf{B} \cdot \mathsf{Ovh}_{\mathsf{Sort}}\left(M, \mathsf{B}\right)$ words on the server per execution of the Read protocol.

In summary, reading a single logical block from DB requires accessing $O\left(k\lambda + k^2\right) \cdot \mathsf{mB} \cdot \mathsf{Ovh}\left(\frac{k \cdot (M + M \log M)}{\mathsf{mB}}\right) + \left(\frac{k}{w} + \frac{2k^2}{M} \cdot \mathsf{Ovh}_{\mathsf{Sort}}\left(M, \mathsf{B}\right)\right) \cdot \mathsf{B}$ words on the server.

3.1 Read-Only ORAM with Oblivious Setup

In this section we generalize the notion of an RO-ORAM scheme to allow the client to run the ORAM Setup algorithm, using the server as remote storage, when the logical memory is already stored at the server. We call this primitive an *RO-ORAM scheme with oblivious setup.* This primitive will be used in the next section to construct an ORAM scheme supporting writes with low read overhead.

At a high level, an RO-ORAM scheme with oblivious setup is an RO-ORAM scheme $(\mathsf{Setup}, \mathsf{Read})$ associated with an additional protocol $\mathsf{OblSetup}$ which allows the client to execute the Setup algorithm using the server as remote storage when the logical memory is already stored on the server, 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.

In the full version [WW18] we formalize this notion, and show that the RO-ORAM scheme of Construction 1 has oblivious setup. The oblivious setup protocol relies on the building blocks of Construction 1, and additionally uses a CPA-secure symmetric encryption scheme (whose existence follows from the existence of OWFs). The high-level idea is conceptually simple. The client first encrypts the logical memory, then generates the codeword copies by encoding the *encrypted* logical memory. This can be done by running the encoding procedure of the LDC "in the clear" (using the server as remote storage), because by the CPA-security of the encryption scheme, the access pattern of the encoding procedure reveals no information on the logical memory. (Indeed, the access pattern might depend on the *values of the ciphertexts*, but those are computationally indistinguishable from encryptions of 0.) Then, the client can use an "empty" metadata (initialized to $\mathbf{0}$) 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 full version [WW18], where we prove the following:

Lemma 2 (RO-ORAM with oblivious setup). *Assuming OWFs, and assuming the security of the building blocks of Construction 1, there exists a computationally-secure RO-ORAM scheme with oblivious setup. Moreover, if:*

- *the logical memory DB has block size B, and the metadata ORAM has block size mB, satisfying $\mathsf{B} > \mathsf{mB} \geq \log M$,*
- *the metadata ORAM has $\mathsf{Ovh}(N)$ overhead for memories of size N, and its setup algorithm can be executed using the server as remote storage by accessing $\mathsf{T}_m(N)$ words on the server, where the client (server) stores s_C (s_S) size-mB blocks,*
- *the oblivious-access sort algorithm has $\mathsf{Ovh}_{\mathsf{Sort}}(n, \mathsf{B})$ overhead when operating on inputs consisting of n size-B blocks,*
- *the LDC has query complexity k, codeword length M, and on messages of length n its encoding procedure performs $\mathsf{T}_{\mathsf{LDC}}(n)$ operations (i.e., touches $\mathsf{T}_{\mathsf{LDC}}(n)$ message symbols),*

then the $\mathsf{OblSetup}$ protocol accesses

$$\lambda + \mathsf{T}_m\left(\frac{k(M + M\log M)}{\mathsf{mB}}\right) + 2n \cdot \frac{\mathsf{B}}{w} + \mathsf{T}_{\mathsf{LDC}}(n) \cdot \frac{\mathsf{B}}{w} + kM \cdot \frac{\mathsf{B}}{w}$$

$$+ \left(\frac{kM}{\mathsf{mB}} + kM\right) \cdot \mathsf{mB} \cdot \mathsf{Ovh}\left(\frac{k(M + M\log M)}{\mathsf{mB}}\right) + k \cdot \mathsf{B} \cdot \mathsf{Ovh}_{\mathsf{Sort}}(n, \mathsf{B})$$

words on the server, where w denotes the word size. Moreover, the client stores $s_C \cdot \frac{\mathsf{mB}}{\mathsf{B}}$ size-B blocks, and the server stores $n + kM + s_S \cdot \frac{\mathsf{mB}}{\mathsf{B}} + \lambda$ size-B blocks.

A Note on Statistically-Secure RO-ORAM with Oblivious Setup. Our RO-ORAM with oblivious setup scheme is computationally-secure, even assuming the server does not see the memory contents. This is due to the fact that the access pattern during LDC-encoding might depend on the contents of the message being encoded, which in our case is the encrypted contents of the logical memory. Since the encryptions of two logical memories are only computationally indistinguishable, the resultant security is computational. We note that using an LDC with additional properties, we can obtain a *statistically-secure* RO-ORAM scheme with oblivious setup. Concretely, if the LDC encoding procedure is oblivious in the sense that its access pattern is independent of the contents of the message being encrypted (a property satisfied by, e.g., linear codes) then one can run the LDC encoding procedure on the logical memory itself, and encryption is not needed. Similarly, if the LDC has a sufficiently small encoding *circuit*, then encoding can be performed directly on the (un-encrypted) logical memory.

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

In this section we extend the RO-ORAM scheme of Sect. 3 to support writes, while preserving the overhead of read operations. We instantiate our construction in several parameter regimes, obtaining the following results (see the full version [WW18] for the proofs).

First, by instantiating our construction with "best possible" sorting circuits and LDCs, we prove Theorem 2:

Theorem 7 (ORAM, "dream" parameters; formal statement of Theorem 2). *Assume the existence of OWFs, as well as LDCs and sorting circuits as in Corollary 1, where the LDC has the following additional properties:*

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

Then there exists an ORAM scheme for memories 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 $O(\log\log n)$ overhead, and write operations have $O(n^{\epsilon})$ 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 8 (ORAM, milder parameters). *Assume the existence of OWFs, as well as LDCs and sorting circuits as in Corollary 2, where the LDC has the additional properties specified in Theorem 7. Then there exists an ORAM scheme for memories 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* $o(\log n)$ *overhead, and* write *operations have* $O(n^{\epsilon})$ *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 9 (ORAM, low write overhead; formal statement of Theorem 3). *Assume the existence of OWFs, as well as LDCs and sorting circuits as in Corollary 1, where the LDC has the following additional properties:*

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

Then there exists an ORAM scheme for memories 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* $o(\log n)$ *overhead, and* write *operations have* $n^{o(1)}$ *overhead.*

Construction Overview. As outlined in Sect. 1.2, the ORAM consists of ℓ levels of increasing size (growing from top to bottom), where initially the logical memory 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 logical memories given as array data structures (namely, in which blocks can only be accessed by specifying the location of the block in the logical memory), but upper levels are too small to contain the entire logical memory, 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 an RO-ORAM $\mathcal{O}^i$ (for array data structures). 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

[4] The construction can use any RO-ORAM scheme, but the read overhead is at least 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 6), though described for logical memories given as arrays, can actually support logical memories given as map data structures.

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 block of the logical memory is the k'th block of $\mathcal{DB}^\ell$).

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 logical memory, it is too costly for the client to run on his own. Second, while the setup of an 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 (Sect. 3.1). For the second property we use the fact that our RO-ORAM scheme described in Sect. 3 has a highly-efficient oblivious setup protocol.

Given these building blocks, the ORAM operates as follows. To read the j'th logical memory block, the client looks for the block in every level. At the lowest level ℓ, which contains the entire logical memory, 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 at most one copy of each logical memory block, and the most recent copy appears in the top-most level that contains the block.

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 array, and $\mathcal{MO}^i, \mathcal{O}^i$ are updated accordingly). See Figs. 2 and 4 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, Theorems 7 and 8 are obtained by setting ℓ to be constant, and Theorem 9 is obtained by setting $\ell = \frac{\log n}{\log^2 \log n}$.

We now formally describe the construction.

Construction 2 (ORAM with writes). The scheme uses the following building blocks:

- An RO-ORAM scheme with oblivious setup $(\mathsf{Setup}_R, \mathsf{Read}_R, \mathsf{OblSetup}_R)$.
- 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 logical memory block contains its logical address.[6]
    - Generate an 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 $_i$ dummy memory blocks.
    - Generate an 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 $\mathcal{MO}^i$.
  - Output. The long-term client key $\mathsf{ck} = \left(\mathsf{ck}_R^\ell, \left\{\mathsf{ck}_R^i, \mathsf{ck}_m^i\right\}_{i\in[\ell-1]}\right)$ consists of the client keys for the RO-ORAMs $\mathcal{O}^i$ and the metadata ORAMs $\mathcal{MO}^i$ of all levels. The server state $\mathsf{st}_S = \left(\mathsf{count}, \mathsf{st}_R^\ell, \mathcal{DB}^\ell, \left\{\mathsf{st}_R^i, \mathsf{st}_m^i, \mathcal{DB}^i\right\}_{i\in[\ell-1]}\right)$ contains the counter count of the number of write operations performed, 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.

[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.

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 $1 \le k \le n$:
  - 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 an array 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 $1 \le k \le l_{\ell-1}$:
  - 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 $\mathsf{st}_m^{\ell-1}$).

Fig. 1. The $\mathsf{ReShuffle}^{\ell}$ protocol used in Construction 2

The Read protocol. To read the logical memory block at location $\mathsf{addr} \in [n]$ from the server S, the client C with key $\left(\mathsf{ck}_R^{\ell}, \left\{\mathsf{ck}_R^i, \mathsf{ck}_m^i\right\}_{i \in [\ell-1]}\right)$ 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).

- Determine block location in level i. For every level $1 \le i \le \ell-1$, 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 $l = \perp$.)
- Read block from level i. For every level $1 \le i \le \ell-1$, if $l = \perp$, set $l = 1$. Run the Read_R protocol on $\mathcal{O}^i$ to read the l'th block from $\mathcal{DB}^i$.
- Read block from level ℓ. Run the Read_R protocol on $\mathcal{O}^{\ell}$ to read the addr'th block from $\mathcal{DB}^{\ell}$.

– Output. 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 to block $\mathsf{addr} \in [n]$ in the logical memory, the client C with key $\left(\mathsf{ck}_R^\ell, \left\{\mathsf{ck}_R^i, \mathsf{ck}_m^i\right\}_{i\in[\ell-1]}\right)$ 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 Fig. 1, namely execute $\mathsf{ReShuffle}^\ell\left(\mathsf{ck}_R^{\ell-1}, \mathsf{ck}_R^\ell, \mathsf{ck}_m^{\ell-1}, \mathsf{st}_R^{\ell-1}, \mathsf{st}_R^\ell, \mathsf{st}_m^{\ell-1}\right)$.

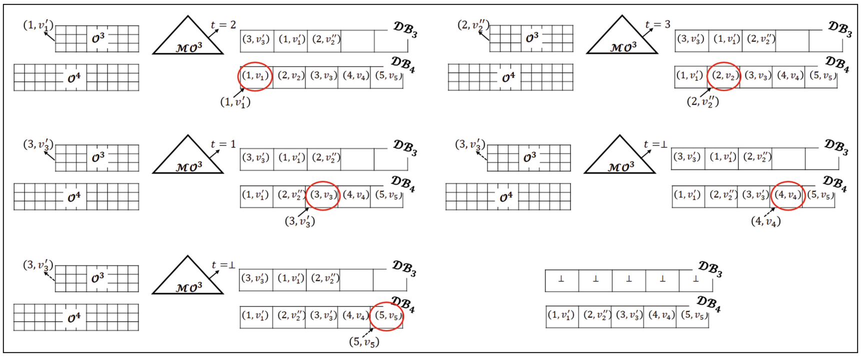

Fig. 2. $\mathsf{ReShuffle}^\ell$ execution on a toy-example ORAM with logical memory size $n = 5$ and $\ell = 4$ levels. The red circle indicates the block which is currently updated. Arrows denote the output of the metadata and RO ORAMs, where dashes arrows denote dummy accesses. Block 1 is updated 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 $\mathcal{MO}^3, \mathcal{MO}^4$).

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 (j, v_j'). 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 an array 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.

Fig. 3. The $\mathsf{ReShuffle}$ protocol used in Construction 2

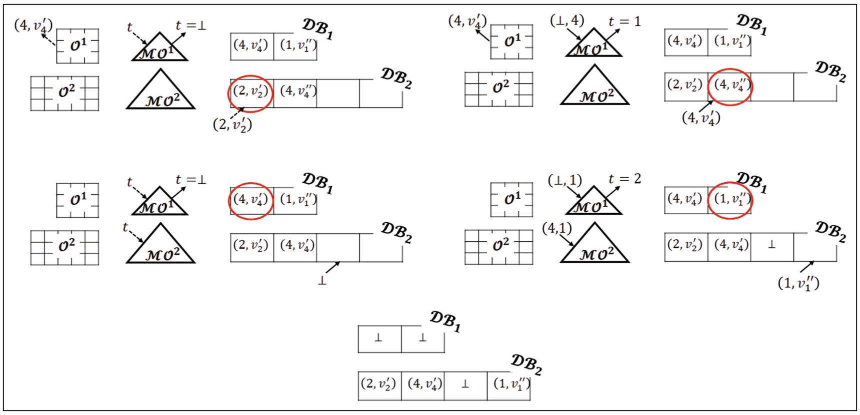

Fig. 4. ReShuffle execution for $i = 1$ on the ORAM from Fig. 2. The red circle indicates the block which is currently updated. Arrows denote the output of the metadata and RO ORAMs, where dashes arrows denote dummy accesses. The blocks of $\mathcal{DB}^2$ are updated first. The first block of $\mathcal{DB}^2$ 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 $\mathcal{O}^1, \mathcal{O}^2$).

- For every i from $\ell - 2$ down to 0 for which l_i divides count, reshuffle level i into level $i + 1$ using the ReShuffle procedure of Fig. 3, namely execute $\mathsf{ReShuffle}\left(i, \mathsf{ck}_R^i, \mathsf{ck}_R^{i+1}, \mathsf{ck}_m^i, \mathsf{ck}_m^{i+1}, \mathsf{st}_R^i, \mathsf{st}_R^{i+1}, \mathsf{st}_m^i, \mathsf{st}_m^{i+1}\right)$.

Remark on De-amortization. We note that using a technique of Ostrovsky and Shoup [OS97], the server complexity in Construction 2 can be de-amortized, by slightly modifying the Write protocol to allow the reshuffling process to be *spread-out* 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.

A Note on Statistically-Secure ORAM with Writes. Our ORAM with writes constructions (Theorems 7–9) are computationally-secure due to the use of a computationally-secure RO-ORAM with oblivious setup. However, given a *statistically-secure* RO-ORAM with oblivious setup the resultant ORAM with writes would also be statistically secure. As noted in Sect. 3.1, such a scheme can be obtained assuming an LDC with a small encoding circuit, or with an oblivious encoding procedure. Thus, given an LDC with one of these additional properties we can get a statistically-secure ORAM with writes (with the parameters stated in Theorems 7–9).

Acknowledgements. Research supported by NSF grants CNS-1314722, CNS-1413964, CNS-1750795 and the Alfred P. Sloan Research Fellowship. The first author was supported in part by The Eric and Wendy Schmidt Postdoctoral Grant for Women in Mathematical and Computing Sciences.

References

[AFN+17] Abraham, I., Fletcher, C.W., Nayak, K., Pinkas, B., Ren, L.: Asymptotically tight bounds for composing ORAM with PIR. In: Fehr, S. (ed.) PKC 2017. LNCS, vol. 10174, pp. 91–120. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54365-8_5

[AKS83] Ajtai, M., Komlós, J., Szemerédi, E.: An $O(n \log n)$ sorting network. In: Proceedings of the 15th Annual ACM Symposium on Theory of Computing, 25–27 April 1983, pp. 1–9 (1983)

[AKST14] Apon, D., Katz, J., Shi, E., Thiruvengadam, A.: Verifiable oblivious storage. In: Krawczyk, H. (ed.) PKC 2014. LNCS, vol. 8383, pp. 131–148. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54631-0_8

[BCP15] Boyle, E., Chung, K.-M., Pass, R.: Large-scale secure computation: multiparty computation for (parallel) RAM programs. In: Gennaro, R., Robshaw, M. (eds.) CRYPTO 2015. LNCS, vol. 9216, pp. 742–762. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-48000-7_36

[BIM00] Beimel, A., Ishai, Y., Malkin, T.: Reducing the servers computation in private information retrieval: PIR with preprocessing. In: Bellare, M. (ed.) CRYPTO 2000. LNCS, vol. 1880, pp. 55–73. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-44598-6_4

[BIPW17] Boyle, E., Ishai, Y., Pass, R., Wootters, M.: Can we access a database both locally and privately? In: Kalai, Y., Reyzin, L. (eds.) TCC 2017. LNCS, vol. 10678, pp. 662–693. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-70503-3_22

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

[BN16] Boyle, E., Naor, M.: Is there an oblivious RAM lower bound? In: Proceedings of the 2016 ACM Conference on Innovations in Theoretical Computer Science, Cambridge, MA, USA, 14–16 January 2016, pp. 357–368 (2016)

[CFL+13] Chee, Y.M., Feng, T., Ling, S., Wang, H., Zhang, L.F.: Query-efficient locally decodable codes of subexponential length. Comput. Complex. **22**(1), 159–189 (2013)

[CHR17] Canetti, R., Holmgren, J., Richelson, S.: Towards doubly efficient private information retrieval. In: Kalai, Y., Reyzin, L. (eds.) TCC 2017. LNCS, vol. 10678, pp. 694–726. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-70503-3_23

[CKW13] Cash, D., Küpçü, A., Wichs, D.: Dynamic proofs of retrievability via oblivious RAM. In: Johansson, T., Nguyen, P.Q. (eds.) EUROCRYPT 2013. LNCS, vol. 7881, pp. 279–295. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38348-9_17

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

[DvDF+16] Devadas, S., van Dijk, M., Fletcher, C.W., Ren, L., Shi, E., Wichs, D.: Onion ORAM: a constant bandwidth blowup oblivious RAM. In: Kushilevitz, E., Malkin, T. (eds.) TCC 2016. LNCS, vol. 9563, pp. 145–174. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49099-0_6

[Efr09] Efremenko, K.: 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, 31 May–2 June 2009, pp. 39–44 (2009)

[FGHK16] Find, M.G., Golovnev, A., Hirsch, E.A., Kulikov, A.S.: 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, pp. 89–98 (2016)

[FNR+15] Fletcher, C.W., Naveed, M., Ren, L., Shi, E., Stefanov, E.: Bucket ORAM: single online roundtrip, constant bandwidth oblivious RAM. IACR Cryptology ePrint Archive 2015:1065 (2015)

[GGH+13] Gentry, C., Goldman, K.A., Halevi, S., Julta, C., Raykova, M., Wichs, D.: Optimizing ORAM and using it efficiently for secure computation. In: De Cristofaro, E., Wright, M. (eds.) PETS 2013. LNCS, vol. 7981, pp. 1–18. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39077-7_1

[GHJR15] Gentry, C., Halevi, S., Jutla, C., Raykova, M.: Private database access with HE-over-ORAM architecture. In: Malkin, T., Kolesnikov, V., Lewko, A.B., Polychronakis, M. (eds.) ACNS 2015. LNCS, vol. 9092, pp. 172–191. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-28166-7_9

[GKK+12] Gordon, S.D., et al.: Secure two-party computation in sublinear (amortized) time. In: The ACM Conference on Computer and Communications Security, CCS 2012, Raleigh, NC, USA, 16–18 October 2012, pp. 513–524 (2012)

[GMOT12] Goodrich, M.T., Mitzenmacher, M., Ohrimenko, O., Tamassia, R.: 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, 17–19 January 2012, pp. 157–167 (2012)

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

[Gol87] Goldreich, O.: 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, NY, USA, pp. 182–194 (1987)

[Goo14] Goodrich, M.T.: 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, 31 May–03 June 2014, pp. 684–693 (2014)

[HO08] Hemenway, B., Ostrovsky, R.: Public-key locally-decodable codes. In: Wagner, D. (ed.) CRYPTO 2008. LNCS, vol. 5157, pp. 126–143. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-85174-5_8

[HOSW11] Hemenway, B., Ostrovsky, R., Strauss, M.J., Wootters, M.: Public key locally decodable codes with short keys. In: Goldberg, L.A., Jansen, K., Ravi, R., Rolim, J.D.P. (eds.) APPROX/RANDOM - 2011. LNCS, vol. 6845, pp. 605–615. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22935-0_51

[HOWW18] Hamlin, A., Ostrovsky, R., Weiss, M., Wichs, D.: Private anonymous data access. IACR Cryptology ePrint Archive 2018:363 (2018)

[IM02] Iwama, K., Morizumi, H.: An explicit lower bound of $5n-o(n)$ for boolean circuits. In: Diks, K., Rytter, W. (eds.) MFCS 2002. LNCS, vol. 2420, pp. 353–364. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45687-2_29

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

[KLO12] Kushilevitz, E., Lu, S., Ostrovsky, R.: 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, 17–19 January 2012, pp. 143–156 (2012)

[KM18] Kushilevitz, E., Mour, T.: Sub-logarithmic distributed oblivious RAM with small block size. CoRR, abs/1802.05145 (2018)

[KO97] Kushilevitz, E., Ostrovsky, R.: Replication is not needed: single database, computationally-private information retrieval. In: 38th Annual Symposium on Foundations of Computer Science, FOCS 1997, Miami Beach, Florida, USA, 19–22 October 1997, pp. 364–373 (1997)

[KS14] Keller, M., Scholl, P.: Efficient, oblivious data structures for MPC. In: Sarkar, P., Iwata, T. (eds.) ASIACRYPT 2014. LNCS, vol. 8874, pp. 506–525. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-45608-8_27

[KT00] Katz, J., Trevisan, L.: On the efficiency of local decoding procedures for error-correcting codes. In: Proceedings of the Thirty-Second Annual ACM Symposium on Theory of Computing, 21–23 May 2000, Portland, OR, USA, pp. 80–86 (2000)

[LHS+14] Liu, C., Huang, Y., Shi, E., Katz, J., Hicks, M.W.: Automating efficient RAM-model secure computation. In: 2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, 18–21 May 2014, pp. 623–638 (2014)

[LN18] Larsen, K.G., Nielsen, J.B.: Yes, there is an oblivious RAM lower bound!. In: Shacham, H., Boldyreva, A. (eds.) CRYPTO 2018. LNCS, vol. 10992, pp. 523–542. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96881-0_18

[LO13] Lu, S., Ostrovsky, R.: Distributed oblivious RAM for secure two-party computation. In: Sahai, A. (ed.) TCC 2013. LNCS, vol. 7785, pp. 377–396. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36594-2_22

[LPM+13] Lorch, J.R., Parno, B., Mickens, J.W., Raykova, M., Schiffman, J.: Shroud: ensuring private access to large-scale data in the data center. In: Proceedings of the 11th USENIX Conference on File and Storage Technologies, FAST 2013, San Jose, CA, USA, 12–15 February 2013, pp. 199–214 (2013)

[MBC14] Mayberry, T., Blass, E.-O., Chan, A.H.: Efficient private file retrieval by combining ORAM and PIR. In: 21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, 23–26 February 2014 (2014)

[MLS+13] Maas, M., Love, E., Stefanov, E., Tiwari, M., Shi, E., Asanovic, K., Kubiatowicz, J., Song, D.: PHANTOM: practical oblivious computation in a secure processor. In: 2013 ACM SIGSAC Conference on Computer and Communications Security, CCS 2013, Berlin, Germany, 4–8 November 2013, pp. 311–324 (2013)

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

[Ost90] Ostrovsky, R.: Efficient computation on oblivious RAMs. In: Proceedings of the 22nd Annual ACM Symposium on Theory of Computing, 13–17 May 1990, Baltimore, Maryland, USA, pp. 514–523 (1990)

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

[PPRY] Patel, S., Persiano, G., Raykova, M., Yeo, K.: Panorama: Oblivious RAM with logarithmic overhead. In: FOCS, (2018, to appear)

[Rag07] Raghavendra, P.: A note on Yekhanin's locally decodable codes. In: Electronic Colloquium on Computational Complexity (ECCC), vol. 14, no. 16 (2007)

[RFK+15] Ren, L., et al.: Constants count: practical improvements to oblivious RAM. In: 24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, 12–14 August 2015, pp. 415–430 (2015)

[SCSL11] Shi, E., Chan, T.-H.H., Stefanov, E., Li, M.: Oblivious RAM with $O((\log N)^3)$ worst-case cost. In: Lee, D.H., Wang, X. (eds.) ASIACRYPT 2011. LNCS, vol. 7073, pp. 197–214. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-25385-0_11

[SS13] Stefanov, E., Shi, E.: ObliviStore: high performance oblivious distributed cloud data store. In: 20th Annual Network and Distributed System Security Symposium, NDSS 2013, San Diego, California, USA, 24–27 February 2013 (2013)

[SvDS+13] Stefanov, E., et al.: Path ORAM: an extremely simple oblivious RAM protocol. In: 2013 ACM SIGSAC Conference on Computer and Communications Security, CCS 2013, Berlin, Germany, 4–8 November 2013, pp. 299–310 (2013)

[WCS15] Wang, X., Chan, T.-H.H., Shi, E.: 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, 12–16 October 2015, pp. 850–861 (2015)

[WGK18] Wang, X., Gordon, S.D., Katz, J.: Simple and efficient two-server ORAM. IACR Cryptology ePrint Archive, 2018:5 (2018)

[WHC+14] Wang, X.S., Huang, Y., Chan, T.-H.H., Shelat, A., Shi, E.: SCORAM: oblivious RAM for secure computation. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, 3–7 November 2014, pp. 191–202 (2014)

[Woo07] Woodruff, D.P.: New lower bounds for general locally decodable codes. In: Electronic Colloquium on Computational Complexity (ECCC), vol. 14, no. 6 (2007)

[WS12] Williams, P., Sion, R.: Single round access privacy on outsourced storage. In: The ACM Conference on Computer and Communications Security, CCS 2012, Raleigh, NC, USA, 16–18 October 2012, pp. 293–304 (2012)

[WW18] Weiss, M., Wichs, D.: Is there an oblivious RAM lower bound for online reads? IACR Cryptology ePrint Archive 2018:619 (2018)

[Yek07] Yekhanin, S.: 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, 11–13 June 2007, pp. 266–274 (2007)

[YFR+13] Yu, X., Fletcher, C.W., Ren, L., van Dijk, M., Devadas, S.: Generalized external interaction with tamper-resistant hardware with bounded information leakage. In: CCSW 2013, Proceedings of the 2013 ACM Cloud Computing Security Workshop, Co-located with CCS 2013, Berlin, Germany, 4 November 2013, pp. 23–34 (2013)

[ZMZQ16] Zhang, J., Ma, Q., Zhang, W., Qiao, D.: MSKT-ORAM: a constant bandwidth ORAM without homomorphic encryption. IACR Cryptology ePrint Archive 2016:882 (2016)