J Cryptol (2022)35:6
https://doi.org/10.1007/s00145-022-09419-1

Journal of
CRYPTOLOGY

Research Article

Locality-Preserving Oblivious RAM[*]

Gilad Asharov
Department of Computer Science, Bar-Ilan University, 5290002 Ramat-Gan, Israel
Gilad.Asharov@biu.ac.il

T.-H. Hubert Chan
Department of Computer Science, The University of Hong Kong, Pokfulam Road, Hong Kong, SAR, China
hubert@cs.hku.hk

Kartik Nayak
Department of Computer Science, Duke University, Durham, USA
kartik@cs.duke.edu

Rafael Pass
Department of Computer Science, Cornell Tech, New York, USA
rafael@cs.cornell.edu

Ling Ren
Computer Science Department, University of Illinois at Urbana-Champaign, Urbana, USA
renling@illinois.edu

Elaine Shi
Computer Science and Electrical and Computer Engineering Departments, Carnegie Mellon University, Pittsburgh, USA
runting@gmail.com

Communicated by Stefano Tessaro

Received 20 August 2020 / Revised 31 December 2021 / Accepted 3 January 2022
Online publication 24 January 2022

Abstract. Oblivious RAMs, introduced by Goldreich and Ostrovsky [JACM'96], compile any RAM program into one that is "memory oblivious," i.e., the access pattern to the memory is independent of the input. All previous ORAM schemes, however, completely break the *locality* of data accesses (for instance, by shuffling the data to pseudorandom positions in memory). In this work, we initiate the study of *locality-preserving ORAMs*—ORAMs that preserve locality of the accessed memory regions, while leaking only the lengths of contiguous memory regions accessed. Our main results demonstrate the existence of a locality-preserving ORAM with polylogarithmic overhead both in terms of bandwidth and locality. We also study the trade-off between locality, bandwidth and leakage, and show that any scheme that preserves locality and does not leak the lengths of the contiguous memory regions accessed, suffers from prohibitive bandwidth. To further improve the parameters, we also consider a weaker notion of a File ORAM, which

[*]A preliminary version of this paper appeared in IACR-EUROCRYPT 2019

supports accesses to predefined non-overlapping regions. Assuming one-way functions, we present a computationally secure File ORAM that has a work overhead and locality of roughly $O(\log^2 N)$, while ignoring $\log\log N$ factors. To the best of our knowledge, before our work, the only works combining locality and obliviousness were for symmetric searchable encryption [e.g., Cash and Tessaro (EUROCRYPT'14), Asharov et al. (STOC'16)]. Symmetric search encryption ensures obliviousness if each keyword is searched only once, whereas ORAM provides obliviousness to any input program. Thus, our work generalizes that line of work to the much more challenging task of preserving locality in ORAMs.

Keywords. Oblivious RAM, Locality, Randomized algorithms.

1. Introduction

Oblivious RAM, originally proposed in the seminal work by Goldreich and Ostrovsky [25,26], allows a client to outsource encrypted data to an untrusted server, and access the data in a way such that the access patterns observed by the server are provably obfuscated.

Thus far, the primary metric used to analyze ORAM schemes has been bandwidth which is the number of memory blocks accessed for every logical access. After a long sequence of works (e.g., [25,35,37,38,42]), it is now understood that ORAM schemes can be constructed incurring only *logarithmic* bandwidth [3], and moreover, this is asymptotically optimal [25,34].

An important performance metric that has been traditionally overlooked in the ORAM literature is *data locality*. The majority of real-world applications and programs exhibit a high degree of data locality, i.e., if a program or application accesses some address it is very likely to access also a neighboring address. This observation has profoundly influenced the design of storage systems—for example, commodity hard drive and SSD disks support sequential accesses faster than random accesses.

Unfortunately, existing ORAM schemes (e.g., [3,17,25,26,37,38,42]) are not locality-friendly. Randomization in ORAMs is inherent due to the requirement to hide the access pattern of the program, and ORAM schemes (pseudo-)randomly permute blocks and shuffle them in the memory. As a result, if a client wants to read a large file consisting of $\Theta(N)$ *contiguous* blocks, all known ORAM schemes would have to access more than $\Omega(N \log N)$ random (i.e., discontiguous) disk locations, introducing significant delays due to lack of locality.

In this paper, we ask the question: Can we design ORAM schemes with data locality? At first sight, this seems impossible. Intuitively, an ORAM scheme must hide whether the client requests N random locations or a single contiguous region of size N. As a result, such a scheme cannot preserve locality, and indeed, we formalize this intuition and formally show that any ORAM scheme that hides the differences between the above two extreme cases must necessarily suffer from either high bandwidth or bad locality.

However, this does not mean that providing oblivious data accesses and preserving locality simultaneously is a hopeless cause. In particular, in many practical applications, it may already be public knowledge that a user is accessing contiguous regions; e.g., consider the following two motivating scenarios:

- *Outsourced file server.* Imagine that a client outsources encrypted files to a server, and then repeatedly queries the server to retrieve various files. In this case, each file captures a contiguous region in logical memory. Note that unless we pad all files to the maximum size possible (which can be very expensive if files sizes vary greatly), we would already leak the file size (i.e., length of contiguous memory region visited) on each request.
- *Outsourced range query database.* Consider an outsourced (encrypted) database system where a client makes range queries on a primary search key, e.g., an IoT database that allows a client to retrieve all sensor readings during a specified time range. We would like to protect the client's access patterns from the server. As previous works argued [13,30], in this case one can leverage differential privacy to hide the number of matching records and it may be safe to reveal a noisy version of the length of the contiguous region accessed.

Note that in both of the above scenarios, some length leakage seems unavoidable unless we always pad to the maximum with every request—and this is true even if we employ ORAM to outsource the files/database! Further, disk IO may be more costly than network bandwidth depending on the deployment scenario: For example, if the server is serving many clients simultaneously (e.g., serving many users from the same organization sharing a secret key, or if the server has a trusted CPU such as Intel SGX and is serving multiple mutually distrustful clients), the system's bottleneck may well be the server's disk I/O rather than the server's aggregate bandwidth.

Motivated by these practical scenarios, we ask the following question.

Can we construct a bandwidth-efficient ORAM that preserves data locality while leaking only the lengths of contiguous regions accessed?

We answer the question in the affirmative and prove the following result:

Theorem 1.1. (Informal) *Let N be the size of the logical address space. There is an ORAM scheme that makes use of only 2 disks and $O(1)$ client storage, such that upon receiving a sufficiently long request sequence containing T logical addresses, the ORAM can correctly answer the requests paying only $T \cdot \mathsf{poly}\log N$ bandwidth, and moreover, if the T addresses requested contains ℓ discontiguous regions, the ORAM server visits only $\ell \cdot \mathsf{poly}\log N$ discontiguous regions on its 2 disks. The construction only leaks the length of contiguous regions accessed.*

To the best of our knowledge, we are the first to consider and formulate the problem of locality-friendly ORAM. Even formulating the problem turns out to be non-trivial, since it requires teasing out the boundaries between theoretical feasibility and impossibility, and capturing what kind of leakage is reasonable in practical applications and yet does not rule out constructions that are both bandwidth-efficient and locality-friendly. Besides the conceptual definitional contributions, we also describe novel algorithmic techniques that result in the first non-trivial locality-friendly ORAM construction.

To help the reader understand the technical nature of our work, we point out that our problem formulation in fact generalizes a line of work on optimizing locality in searchable symmetric encryption (SSE) schemes. The issue of locality was encountered in recent implementations [15] of searchable symmetric encryption in real-world databases, showing that the practical performance of known schemes that overlook the issue of locality do not scale well to large data sizes. The problem of optimizing locality in searchable symmetric schemes has received considerable attention recently (see, e.g., [6,7,20–22]). Our problem generalizes this line of work, and achieving good locality in oblivious RAM is significantly more challenging due to the following reasons: (1) In SSE, obliviousness is guaranteed only if each "file" is accessed at most once (and the length of the file is also leaked in SSE);[1] and (2) SSE assumes that rebuilding the "server-side oblivious data structure" happens on a powerful client with linear storage, and thus, the rebuilding comes "for free." We show, for the first time, how to remove both of these above restrictions, and provide a generalized, full-fledged oblivious memory abstraction that supports unbounded polynomial accesses and yet preserves both bandwidth and locality.

In some cases, such as a motivating scenario of an outsourced file server, it suffices to consider a weaker primitive than locality-preserving ORAM. In particular, instead of storing and accessing arbitrary range data, files of fixed lengths are stored and accessed. Each memory block is a part of exactly one file, and the sizes of the files are leaked on accessing them. We investigate a primitive called File ORAM, which addresses this specific motivating application. We achieve a computationally secure File ORAM that has a work overhead and locality of $O(\log^2 N)$ ignoring $\log\log N$ factors.

2. Technical Roadmap

In the following, we provide a summary of results and techniques. In Sect. 2.1, we discuss our modeling of locality. In Sect. 2.2, we discuss our lower bounds, providing trade-offs between the locality of a program, leakage and bandwidth. Toward introducing our construction, we start in Sect. 2.3 with a warmup– oblivious sort with "good" locality. In Sect. 2.4, we introduce range ORAM, our core building block for achieving locality, in which in Sect. 2.5 we overview its construction. In Sect. 2.6, we overview a variant of Range ORAM, called "Online Range ORAM," which can also be viewed as a locality-preserving ORAM, i.e., our main new primitive. We then present the overview of File ORAM in Sect. 2.7.

2.1. *A Generalized Model of Locality*

How do we model locality of an algorithm (e.g., an ORAM or SSE algorithm)? A natural option is to use the well-accepted approach adopted by the SSE line of work [6,7,20,22]. Imagine that every time an algorithm (e.g., SSE or ORAM) needs to read an item from disk, it has two choices: (1) Read the next contiguous address and (2) jump to a new address (often called "seek" in the systems literature). While both types of operations contribute to the *bandwidth* measure, only the latter type contributes to the *locality* measure [6,7,20,22] since seeks are significantly more expensive than sequential reads on real-world disks. We point out that locality alone is not a meaningful measure since we can always achieve better locality and minimize jumps by scanning through the entire

[1] Intuitively, a file stores the identifiers of the documents matching a keyword search in SSE schemes.

memory extracting the values we want along the way. Thus, we always use locality in conjunction with a *bandwidth* metric too, i.e., how many blocks we must fetch from the disk upon each request.

Motivating the multi-disk model The aforementioned model was adopted by the SSE line of work; however, it is very constraining in the sense that they assume that the server has access to only 1 disk. In practice, cloud-hosting services such as EC2 and Azure provision servers with multiple disks. For example, the white paper by Brewer et al. [10] provides an overview of the disk technology for modern data centers. Constraining to such a single-disk model might rule out interesting cryptographic algorithms of practical value. For example, even the simple task of making a copy of an array would require jumping around linear number of times on a single disk, assuming that both the source and destination arrays must be stored in contiguous regions. However, with just 2 disks, we can accomplish the above without requiring any jumps. More examples are provided in Sect. 3.1. Therefore, we will consider a model that allows multiple disks, and in fact, all the algorithms proposed in this paper only need 2 or 3 disks to achieve good locality and bandwidth efficiency simultaneously. More concretely, we generalize the locality definition as follows.

Defining (D, ℓ)-locality We consider the scenario where the ORAM server may have multiple (but ideally a small number) of disks, where each disk still supports the aforementioned two types of instructions: "read the next contiguous address" and "jump to a new address." Henceforth, we say that an ORAM scheme satisfies (D, ℓ)-locality and β bandwidth cost iff for a sufficiently long input sequence containing B requests spanning L non-contiguous regions, the ORAM server, with access to D disks, may access at most $\beta \cdot B$ blocks and issue at most $\ell \cdot L$ jump instructions. Of course, the adversary can observe all disks and all movement operations in these disks. We refer the readers to Sect. 3.1 for the formal definition.

Under these new definitions, our result can be stated technically as "an ORAM scheme with $(2, \mathsf{poly}\log N)$-locality and $\mathsf{poly}\log N$ bandwidth (amortized) cost" where N is the total number of logical blocks. Moreover, as mentioned, our ORAM scheme leaks only the length of each contiguous region in the request sequence and nothing else (and as mentioned, some leakage is inherent if we desire efficiency).

Open questions Given our new modeling techniques and results, we also suggest several exciting open questions, e.g., is it possible to have an ORAM scheme that achieves $(1, \ell)$-locality and β bandwidth cost where ℓ and β are small? Can we compile source programs that exhibit (D, ℓ)-locality where $D > 1$ with meaningful leakage? For the former question, if there is a lower bound that shows a sharp separation between 1 and 2 disks, it would be technically really intriguing. For the latter question, the constructions in this paper directly imply that if one is willing to leak the disk each request wants to access, such schemes are possible. However, depending on the practical application such leakage varies from reasonable to extremely harmful. Thus, the challenge is to understand the feasibility/infeasibility of achieving such compilation while hiding which disk each request wants to access. We refer the reader to Sect. 8 for other open problems.

2.2. *Locality with No Leakage*

As we already discussed, preserving both bandwidth and locality with no leakage is impossible. We formalize this claim and study trade-offs between leakage profiles and performance. We consider schemes that leak only the total number of accesses (just as in standard ORAM)[2] and show that a scheme with good locality must incur a high bandwidth, even when allowing large client-side space blowup. We prove the following:

Theorem 2.1. *For any $\ell, c \leq \frac{N}{10}$, any (D, ℓ)-local ORAM scheme with c blocks of client storage that leaks no information (besides the total number of requests) must incur $\Omega(\frac{N}{\mathsf{D}})$ bandwidth.*

To intuitively understand the lower bound, consider a simplified case where the O-RAM must satisfy (1, 1)-locality. Consider the following two scenarios: (1) requesting contiguous blocks at addresses $1, 2, \ldots N$; and (2) requesting blocks at random addresses. By the locality constraint, in the former scenario the ORAM scheme can access only 1 contiguous region on 1 disk. Now the oblivious requirement says that the address distributions under these two scenarios must be indistinguishable, and thus, even for the second scenario the ORAM server can only access a single contiguous region too. Now, if each request's address is generated at random, in expectation the desired block is at least $N/2$ far from where the disk's head currently is—and this holds no matter how one arranges the contents stored on the disk, and even when the server's disk may be unbounded! Since the ORAM scheme must perform a single linear scan even in the second scenario, it must read in expectation $N/2$ locations to serve each randomized request. Note that one key idea in this lower bound proof is that we generate the request sequence at random in the second scenario, such that even if the ORAM scheme is allowed to perform arbitrary, possibly randomized setup, informally speaking it does not help. In Sect. 6, we make non-trivial generalizations to the above intuition and prove a lower bound for generalized choices of D and ℓ.

On leaking the lengths Given our lower bound, our constructions presented next leak the lengths of the accessed regions to achieve good locality. Before proceeding with our construction, we remark the following points regarding this leakage: (1) The input program can always break locality (say, via fictitious non-contiguous accesses), and therefore, our scheme can be viewed as a strict generalization of ordinary ORAM schemes. In other words, the user can choose to opt out of the locality feature. (2) As we mentioned above, in many applications it is already public knowledge that the client accesses contiguous regions. In those cases, the leakage is the same had we used an ordinary ORAM [29]. (3) Finally, we stress that just like the case of ordinary ORAM, our locality-friendly ORAM can be combined with differential privacy techniques as Kellaris et al. [30] suggested to offer strengthened privacy guarantees.

Despite these arguments, in some applications with good locality, such leakage might be harmful. For example, a program may access several regions of different lengths and which regions are accessed depend on some sensitive data. Whether the locality

[2]We emphasize that many practical applications leak some more information even when using standard ORAM, e.g., in the form of communication volume. See discussion in below.

feature of our scheme should be used or not is application dependent, and we encourage using the locality feature only in places where the leakage pattern is clear and is public information to begin with.

2.3. *Warmup: Locality-Friendly Oblivious Sort*

Before describing our main construction, we first introduce a new building block called *locality-friendly oblivious sort* which we will repeatedly use. First, we observe that not all known oblivious sorting algorithms are “locality-friendly.” For example, algorithms such as AKS sort [4] and Zig-zag sort [28] are described with a sorting circuit whose wiring has good randomness-like properties (e.g., in AKS the wiring involve expander graphs, which have proven random-walk properties), thus making these algorithms difficult to implement with small locality consuming a small number of disks (while preserving the algorithm’s runtime).

Fortunately, we observe that there is a particular method to implement the Bitonic sort [8] algorithm such that with only 2 disks, the algorithm can be accomplished using $O(\log^2 n)$ “jumps” (note also that “natural” implementations of the Bitonic sort circuit do not seem to have such locality friendliness).

We defer the details of this specific locality-friendly implementation of Bitonic sort to “Appendix A,” stating only the theorem here:

Theorem 2.2. (Locality-friendly oblivious sort) *Bitonic sort (when implemented as in “Appendix A”) is a perfectly oblivious sorting algorithm that sorts n elements using $O(n \log^2 n)$ bandwidth and $(2, O(\log^2 n))$-locality.*

For achieving better asymptotic complexity (albeit, with statistical security as opposed to perfect as in Bitonic), we show that the Bucket oblivious sort, proposed in [1] also enjoys good locality. See “Appendix B.”

2.4. *Range ORAM: An Intermediate, Relaxed Abstraction*

We now start to give an informal exposition of our upper bound results. This is perhaps the most technically sophisticated part of our work.

To achieve the final result, we will do it in two steps. In our final ORAM scheme (henceforth called *Online Range ORAM*), the ORAM client receives the requests one by one in an online fashion, and it is not informed a priori when a contiguous scan would occur in the request sequence. That is, it has exactly the same syntax as an ordinary ORAM, but when the client accesses contiguous addresses, the online range ORAM has to recognize this fact, and fetch contiguous regions from the memory. To reach this final goal, however, we need an intermediate stepping stone called *Range ORAM*, which is an “offline” version of Online Range ORAM. In a Range ORAM, imagine that the ORAM client receives a request sequence that can look ahead into the future, i.e., the client is informed that the next len requests will scan contiguously through the logical memory.

More formally, in a Range ORAM, the ORAM client receives requests of the form $\mathsf{Access}(\mathsf{op}, [s, t], \mathsf{data})$, where $\mathsf{op} \in \{\mathtt{read}, \mathtt{write}\}$, $s, t \in [N]$, $s < t$, and $\mathsf{data} \in$

$(\{0,1\}^b)^{(t-s+1)}$ where b is the block size. Upon each request, the client interacts with the server to update the server-side data structure and fetch the data it needs:

- If $\mathsf{op} = \texttt{read}$, at the end of the request, all blocks whose logical addresses belong to the range $[s, t]$ are written down in server memory starting at a designated address; the server may then return the blocks to the client one by one in a single contiguous scan.
- If $\mathsf{op} = \texttt{write}$, then imagine that the client has already written down a data array consisting of $t - s + 1$ blocks on the server in a designated, contiguous region; the client and the server then perform interactions to update the server-side data structure to reflect that the logical address range $[s, t]$ should now store the contents of data.

Note that as described above, a Range ORAM is well defined even for a client that has only $O(1)$ blocks of storage—and indeed we give a more general formulation by assuming $O(1)$ client storage.

As for obliviousness, we require that the distribution of memory addresses accessed by the Range ORAM can be simulated from the lengths of the accessed ranges only, which implies that there is no other leakage other than these lengths. We prove the following theorem:

Theorem 2.3. *There exists a perfectly secure Range ORAM construction consuming $O(N \log N)$ space with amortized $\mathsf{len} \cdot \mathsf{poly}\log N$ bandwidth and $(2, \mathsf{poly}\log N)$-locality, for accessing a range of length len, while leaking only the length len of the accessed ranges.*

We remark bandwidth is in an amortized sense: For accessing contiguous regions of lengths $\mathsf{len}_1, \ldots, \mathsf{len}_m$, the total bandwidth is $\left(\sum_{i=1}^{m} \mathsf{len}_i\right) \cdot \mathsf{poly}\log N$. Locality is worst case, for each access of a continguous region, of length $\mathsf{len}_1, \ldots,$ or len_m, the locality is $(2, \mathsf{poly}\log N)$, i.e., the total locality of the m accesses is $(2, m \cdot \mathsf{poly}\log N)$.

In comparison, for all existing ORAM schemes, accessing a single region of len contiguous blocks involves accessing $\Omega(\mathsf{len} \cdot \log N)$ blocks residing at discontiguous physical locations. We now overview the high level ideas behind our range ORAM construction.

Strawman scheme: read-only Range ORAM Assuming that the CPU sends only read instructions, we can achieve locality and obliviousness as follows. The idea is to make replications of a set of superblocks that form contiguous memory regions. Specifically, let N be a power of 2 that bounds the size of the logical memory. A size-2^i superblock consists of 2^i consecutive blocks with the starting address being a multiple of 2^i. We call size-1 blocks as "primitive blocks." We store $\log N$ different ORAMs, where the i-th ORAM (for $i = 0, \ldots, \log N - 1$) stores all size-$2^i$ (super)blocks (exactly $N/2^i$ blocks of size 2^i each). Since any contiguous memory region of length 2^i is "covered" by two superblocks of that length, reading any contiguous memory of length 2^i region would boil down to making two accesses to the i-th ORAM.

However, this approach breaks down once we also need to support writes. The main challenge is to achieve data coherency in different ORAMs. Since there are multiple

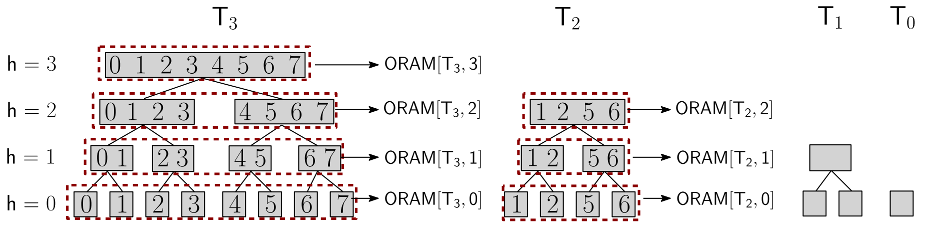


Fig. 1. Hierarchy of range trees. Logically, data is divided into trees of exponentially increasing sizes. In each tree block, a parent superblock stores the contents of both its children. If a block appears in more than one tree, the smallest tree contains the freshest copy. The above figure shows the state of the data structure after two accesses (read, 5, 2, ⊥) and (read, 1, 2, ⊥). h denotes height of a node in the Range Tree .

replicas of each data block, either a write must update all replicas, or a read must fetch all replicas to retrieve the latest copy. Both strategies break data locality.

2.5. *Constructing Range ORAM*

Range Trees The aforementioned strawman scheme demonstrates the challenges we face if we want a Range ORAM supporting both reads and writes. To achieve this we need more sophisticated data structures.

We first describe a *logical* data structure called a *Range Tree* (without specifying at this point how to actually store this logical Range Tree on physical memory). A Range Tree of size 2^i is the following (logical) data structure: the leaves store 2^i primitive blocks sorted by their (possibly non-contiguous) addresses, whereas each internal node replicates and stores all blocks contained in the leaves of its subtree. For example, in Fig. 1, each of T_0, T_1, T_2 and T_3 is a logical Range Tree of sizes 1, 2, 4, 8, respectively. In such a Range Tree, each node at height j stores a superblock of size 2^j (leaves have height 0 and store primitive blocks).

Range ORAM's data structure As shown in Fig. 1, our full Range ORAM (supporting both reads and writes) will *logically* contain a hierarchy of such Range Trees of sizes $1, 2, 4, 8, \ldots, N$, denoted $\mathsf{T}_0, \mathsf{T}_1, \ldots, \mathsf{T}_L$, respectively, where $L = O(\log N)$. These trees form a hierarchy of stashes just like in hierarchical ORAM [25,26], i.e., each T_i is a stash for T_{i+1} which is twice as large. Thus, if a block at some logical address is replicated multiple times in multiple range trees, *the copy in a smaller Range Tree is always more fresh* (e.g., in Fig. 1, notice that the block at logical address 1 appears in both T_3 and T_2). Within each Range Tree, a logical block also appears multiple times within superblocks (or primitive blocks) of different sizes, but all these copies within the same tree contain the same value.

We now specify how these logical Range Trees are stored in the physical memory. Basically, in each Range Tree, all superblocks at the same height will be stored in a separate ORAM—thus an ORAM at height j of the tree stores superblocks of size 2^j.

Besides the ORAMs storing each height of each Range Tree, we also need an auxiliary data structure that facilitates lookup. The client can access this data structure to figure out, for a requested range $[s, t]$, which superblocks in a specific tree height intersect the request. This auxiliary data structure is stored on the server in an ORAM, and it can be viewed as a variant of "oblivious binary search tree."

Fetch phase of the Range ORAM Let us now consider how to read and write contiguous ranges of blocks (i.e., implement the read and write operations of Range ORAM). Each request, no matter read or write requests, proceed in two phases: a fetch phase and a maintain phase. We first describe the fetch phase whose goal is to write down the requested range in a designated contiguous space on the server.

Suppose that the range $[s, t]$ is requested. Without loss of generality, assume that the length of the range $t - s + 1 = 2^i$ (otherwise round it up to the nearest power of 2). Roughly speaking, we would like to achieve the following effect:

- For every Range Tree at least 2^i in size, we would like to fetch all size-2^i superblocks that intersect the range requested—it is not difficult to see that there are at most *two* such superblocks.
- For every Range Tree smaller than 2^i in size, we simply fetch the root.
- Write down all these superblocks fetched in a contiguous region on the server, and then obliviously reconstruct the freshest value of each logical address (using locality-friendly oblivious sort).

Henceforth, we focus only on the Range Trees that are at least 2^i in size since for the smaller trees it is trivial to read the entire root. To achieve the above, roughly speaking, the client may proceed in the following steps. For each Range Tree that is not too small,

1. Look up the auxiliary data structure (stored on the server) to figure out which *two* superblocks to request in the desired height that stores superblocks of size 2^i;
2. Fetch these two desired superblocks from the corresponding ORAM and write down the fetched superblock in a contiguous region (starting at a designated position) on the server's memory.

All these fetched superblocks are written down on the server's memory contiguously (including the root nodes for the smaller Range Trees which we have ignored above). The client now relies on oblivious sorting to reconstruct the freshest copy of each logical address requested, and the result is stored in a designated contiguous region on the server.

Notice that the entire read procedure reads only polylogarithmically many contiguous memory regions:

- Queries to the oblivious auxiliary data structure accesses polylogarithmically many "small" metadata blocks using ordinary oblivious data structures;
- There are only logarithmically many requests to per-height ORAMs storing superblocks of size 2^i. Using an ordinary ORAM scheme, this step requires reading polylogarithmically many regions of size 2^i. Here, since every superblock of size 2^i is bundled together, we do not need to read 2^i separate small blocks from an ORAM, and this is inherently why the algorithm's *locality is independent of the length of the range requested*.
- The oblivious sorting needed for reconstruction also consumes polylogarithmic locality as mentioned in Sect. 2.3.

Maintain phase of the Range ORAM Inspired by the hierarchial ORAM [25,26], here a superblock fetched will be written to the smallest Range Tree that is large enough to fit this superblock. If this Range Tree is full, we will then perform a cascading merge to merge consecutive, full Range Trees into the next empty Range Tree.

During this rebuilding process, we must also maintain correctness, including but not restricted to the following:

- for duplicated copies of each block, figure out the freshest copy and suppress duplicates; and
- correctly rebuild the oblivious auxiliary data structure in the process.

Without going into algorithmic details at this point, most of this rebuilding process can be accomplished through a locality-friendly oblivious sorting procedure as mentioned earlier in Sect. 2.3. However, technically instantiating all the details and making everything work together is non-trivial. To enable this, we in fact introduce a new algorithmic abstraction, that is, *an ordinary ORAM scheme with a locality-friendly initialization procedure* (see Sect. 4.3). We will use this new building block to instantiate both the oblivious auxiliary data structure and each tree height's ORAM. In comparison with a traditional ORAM where rebuilding can be supported by writing the blocks one by one (which will consume super-linear locality), here we would like to rebuild the server-side ORAM data structure using a special locality-friendly algorithm upon receiving a possibly large input array of the blocks. In subsequent technical sections, we show how to have such a special ORAM scheme where initializing the server-side data structure can be accomplished using locality-friendly oblivious sorting as a building block. We refer the reader to Sect. 5 for the algorithmic details. We also refer the reader to Example 5.5 for a demonstration how the levels are updated after few accesses.

2.6. *Online Range ORAM*

Given our Range ORAM abstraction, we are now ready to construct Online Range ORAM. The difference is that now, when the client receives request, it is unaware whether the future requests will be contiguous. In fact, Online Range ORAM provides the same interface as an ordinary ORAM: each request the client receives is of the form $(\mathsf{op}, \mathsf{addr}, \mathsf{data})$ where $\mathsf{op} \in \{\mathtt{read}, \mathtt{write}\}$, and $\mathsf{addr} \in [N]$ specifies a single address to read or write (with data). Yet the Online Range ORAM must preserve the locality that is available in the request sequence up to polylogarithmic factors.

Roughly speaking, we can construct Online Range ORAM from Range ORAM as follows, by using a predictive prefetching idea: When a request (containing a single address) comes in, the client first requests that singe address. When a new request comes in, it checks whether the request is consecutive to the address of the previous request. If so, it requests 2 contiguous blocks—the specified address and also its next address. This can be done by requesting a range in Range ORAM. If the next 2 requests happen to be contiguous, then the client prefetches the next 4 blocks with Range ORAM; if the requests are still contiguous, it will next prefetch 8 blocks with Range ORAM. At any time if the contiguous pattern stops, back off and start requesting a region of size 1 again. It is not hard to see that the Online Range ORAM still preserves polylogarithmic bandwidth blowup; moreover, if the request sequence contains a contiguous region of length len, it will be separated into at most $\log(\mathsf{len})$ Range ORAM requests. Thus, the Online Range ORAM's locality is only a logarithmic factor worse than the Range ORAM. The reader is referred to Sect. 5.5 for further details.

2.7. *File ORAM*

Compared to a Range ORAM which supports accesses to any contiguous memory region, a File ORAM provides a more constrained functionality that supports accesses to a set of files of predefined ranges. More specifically, while in Range ORAM every pair of $[s, t]$ is a valid request (as long as $t > s$), in File ORAM, a client requests files whose boundaries are known a priori; and further, distinct files do not overlap in memory.

Concretely, the primitive is defined as follows: The primitive gets as input a file structure $\mathcal{F} = (F_{\mathsf{fid}_1}, \ldots, F_{\mathsf{fid}_k})$, where each file F_{fid_i} has some length len_i and identifier fid_i. The total size of all lengths is N. It arranges the files in the memory such that later it can support accesses $(\mathsf{op}_i, \mathsf{fid}_i, \mathsf{data}_i)$ with $\mathsf{op}_i \in \{\texttt{read}, \texttt{write}\}$. It then has to update the content of the file fid_i in case $\mathsf{op}_i = \texttt{write}$, or retrieve its content when $\mathsf{op}_i = \texttt{read}$.

Obliviously, we can use Range ORAM to realize File ORAM—but we are interested in a more practical and asymptotically more efficient primitive. We show a construction that has linear space (as opposed to $O(N \log N)$ in our Range ORAM construction), and $\tilde{O}(\log^2 N)$ work overhead, and is $(3, \tilde{O}(\log^2 N))$-locality.[3] Toward that end, we make use of the Bucket Oblivious sort of [1], and we show how it can be implemented with good locality (see "Appendix B").

Naïve construction Given a file structure $\mathcal{F} = (F_{\mathsf{fid}_1}, \ldots, F_{\mathsf{fid}_k})$ of a total size N, we build $\log N$ different ORAMs where the ith ORAM holds up to $N/2^i$ files of size 2^i. If we use Circuit ORAM [42] with a merged stash across all recursion levels [19], accessing each file of 2^i boils down to accessing $O(\log^2 N)$ superblocks of length 2^i. It is not hard to see that this naïve scheme achieves a $O(\log^2 N)$ work, but it consumes $O(N \log N)$ space.

Non-recurrent file hashing scheme To show our linear space File ORAM, we start with constructing File ORAM for non-recurrent accesses, namely, when the same file is not accessed at most once. At a high level, to achieve this, we define a new primitive which we call "Non-Recurrent File Hashing Scheme," build upon the two-dimensional balanced allocation scheme [6]. However, we show how to perform this balanced allocation obliviously.

Given a sequence of files $\mathcal{F} = (F_{\mathsf{fid}_1}, \ldots, F_{\mathsf{fid}_k})$ of various sizes and with total size N, we allocate an array of bins and hash the files to the memory as follows. For each file F_{fid}, we evaluate a pseudo random function on the file identifier fid to receive a (pseudo-)random starting bin g, and place the ith block of the file in the bin $g + i$. That is, the first block of the file is placed in the bin g, the second bin is placed in $g + 1$, and so on. Once this process is completed for all files, each bin contains blocks from different files. We will show that if we allocate $N/\tilde{O}(\log N)$ bins of size $\tilde{O}(\log N)$, while we make sure that the overall space is $O(N)$, then with all but a negligible probability no bin overflows. We pad each bin to contain exactly $\tilde{O}(\log N)$ blocks by adding dummy block when needed.

To access file fid, we compute the starting bin g by applying the pseudorandom function on fid and then read the $\mathsf{len}(\mathsf{fid})$ consecutive bins $g, \ldots, g + \mathsf{len}(\mathsf{fid}) - 1$ to retrieve all the blocks that correspond to the file fid. This guarantees good locality, as

[3]For a function $f(n)$, we let $\tilde{O}(f(n))$ denote $O(f(n) \log f(n))$.

all the data that is associated with the file fid is stored in len consecutive bins, i.e., a contiguous region of length $\mathsf{len} \cdot \tilde{O}(\log N)$. As no file is accessed more than once, this range of memory locations is pseudorandom. As such, we can also support accesses of fictitious files: When accessing a file with $\mathsf{fid} = \perp$ and some length len, we choose a random bin g uniformly at random, and read the bins $g, \ldots, g + \mathsf{len} - 1$.

As for the initialization, we store next to each block some metadata that includes its file identifier and its offset within the file, the destination bin of the block can be computed directly. We then show how to implement the allocation procedure using locality-friendly oblivious sort, from an array containing the data of the files in arbitrary locations. Moreover, we have a metadata ORAM to let the user retrieve the length of each file from its identifier.

Achieving obliviousness for recurrent memory requests To make recurrent requests, we make the following observation: The hierarchical ORAM framework originally proposed by Goldreich and Ostrovsky [25] is, in fact, a method for constructing a recurrent ORAM form a non-recurrent ORAM. Thus, we will apply the hierarchical ORAM framework atop our oblivious non-recurrent File Hashing scheme, somewhat similarly to the way we converted Range Trees to Range ORAM. Here, however, the ith table T_i is a non-recurrent file hashing of size 2^i which consumes space $O(2^i$, whereas Range Tree of size 2^i consumes space $O(i \cdot 2^i)$. As a result, the total space consumption of our File ORAM is $O(N)$ and not $O(N \log N)$ as in our Range ORAM.

2.8. *Related and Subsequent Work*

Related work on locality. Algorithmic performance with data stored on the disk has been studied in the external memory models (e.g., [2,36,39,40] and references within). Fundamental problems in this area include scanning, permuting, sorting, range searching, where there are known lower bounds and matching upper bounds.

Relationship to locality-preserving SSE Searchable symmetric encryption (SSE) enables a client to encrypt an index of record/keyword pairs and later retrieve all records matching a keyword. The typical approach (e.g., [14,16,32,33,41], and references within) is to store an inverted index. Our work is inspired by recent works that study locality in SSE schemes [6,7,20–22]. While locality-preserving oblivious RAM is formally incomparable to the SSE line of work (it allows access to any regions in the memory and not just predefined ones), our new File ORAM formulation can be viewed as a generalization of the onetime ORAM (with free rebuild) construction adopted in recent SSE constructions. File ORAM provides a much stronger security guarantee: Whenever the same file (keyword) is accessed more than once, our FileORAM does not reveal this fact, whereas the SSE schemes do reveal this fact. Table 1 compares the different SSE schemes and our File ORAM.

In a concurrent work, Demertzis, Papadopoulos and Papamanthou [22] also consider the SSE application. In their construction, they leverage as a building block a multi-use ORAM with $O(1)$-locality, by blowing up the bandwidth to $O(\sqrt{N})$ and the client storage to $O(N^{2/3})$. This construction fails to preserve the locality of the input program, and when accessing a region of size len will result in $O(\mathsf{len})$-locality, and $O(\mathsf{len} \cdot \sqrt{N})$-

Table 1. Comparison between FileORAM and local SSE schemes.

Scheme	Space	Bandwidth	Locality	Leakage	Remarks
[14]	$O(N)$	$\mathsf{len} \cdot O(1)$	$(1, O(\mathsf{len}))$	AP, Sizes	
[22]	$O(N)$	$\mathsf{len} \cdot O(\log^{2/3+\delta} N)$	$(1, O(1))$	AP, Sizes	For any constant $\delta > 0$ (assumes large client storage)
[7]	$O(N)$	$\mathsf{len} \cdot O(\log\log\log N)$	$(1, O(1))$	AP, Sizes	Assumes maximal list length $< N/\log^3 N$
[6]	$O(N \log N)$	$\mathsf{len} \cdot O(1)$	$(1, O(1))$	AP, Sizes	
Our FileORAM	$O(N)$	$\mathsf{len} \cdot \tilde{O}(\log^2 N)$	$(3, \tilde{O}(\log N))$	Sizes	$O(1)$ client storage

AP stands for Access Pattern (reveals whether the same file/keyword is accessed more than once). File ORAM comes to address this leakage. $\tilde{O}$ is defined as in Footnote 3

bandwidth. In contrast, we achieve $\mathsf{poly}\log N$-locality and $\mathsf{len} \cdot \mathsf{poly}\log N$-bandwidth when accessing a region of size len, and with $O(1)$-client space.

Oblivious RAM (ORAM) Numerous works [3,24,31,35,37,38,42,43,45–47] construct ORAMs in different settings. Most of the ORAM constructions follow one of two frameworks: the hierarchical framework, originally proposed by Goldreich and Ostrovsky [25,26], or the tree-based framework proposed by Shi et al. [37].

Up until recently, the asymptotically most efficient scheme was given by [31], providing $O(\log^2 N/\log\log N)$ bandwidth. A recent improvement was given by Patel et al. [35], reducing the bandwidth to $O(\log N \cdot \mathsf{poly}\log\log N)$. The scheme of Asharov et al. [3] achieves $O(\log N)$ bandwidth and matches the lower bounds given by Goldreich and Ostrovsky [25,26] and Larsen and Nielsen [34]. Further, the Goldreich–Ostrovsky lower bound is also known not to hold when the memory (i.e., ORAM server) is capable of performing computation [5,23], which is beyond the scope of this paper.

Subsequent work Chakraborti et al. [11] show an ORAM called rORAM with good locality and with $O(\log^2 N)$ bandwidth assuming $\Omega(\log^2 N)$ block size. Their scheme is based on tree-based ORAM. The construction works with large client storage (i.e., linear in the sequential data to be read/write), and reducing this client storage to $O(1)$ would incur multiplicative $\mathsf{poly}\log N$ factors in locality and bandwidth in addition to using more disks to achieve locality. The construction requires $O(N \log N)$ storage.

3. Definitions

Notations and conventions We let $[n]$ denote the set $\{1, \ldots, n\}$. We denote by p.p.t. probabilistic polynomial time Turing machines. A function $\mathsf{negl}(\cdot)$ is called $\mathsf{negligible}$ if for any constant $c > 0$ and all sufficiently large λ's, it holds that $\mathsf{negl}(\lambda) < \lambda^{-c}$. We let λ denote the security parameter. For an ensemble of distributions $\{D_\lambda\}$ (parametrized with λ), we denote by $x \leftarrow D_\lambda$ a sampling of an instance according to the distribution D_λ. Given two ensembles of distributions $\{X_\lambda\}$ and $\{Y_\lambda\}$, we use the notation $\{X_\lambda\} \overset{\epsilon(N)}{\equiv} \{Y_\lambda\}$

to say that the two ensembles are statistically (resp. computationally) indistinguishable if for any unbounded (resp. p.p.t.) adversary $\mathcal{A}$,

$$\left| \Pr_{x \leftarrow X_\lambda} \left[\mathcal{A}(1^\lambda, x) = 1\right] - \Pr_{y \leftarrow Y_\lambda} \left[\mathcal{A}(1^\lambda, y) = 1\right] \right| \leq \epsilon(\lambda)$$

Throughout this paper, for underlying building blocks, we will use n to denote the size of the instance and use λ to denote the security parameter. For our final ORAM constructions, we use N to denote the size of the total logical memory size as well as the security parameter—note that this follows the convention of most existing works on ORAMs [24–26,31,37,38,42].

3.1. *Memory with Multiple Disks and Data Locality*

To understand the notion of data locality, it may be convenient to view the memory as D rotational hard drives or other storage mediums where sequential accesses are faster than random accesses. The program interacting with the memory has to specify which disk to access. Each disk is equipped with one read/write head. In order to serve a `read` or `write` request with address addr in some disk $\mathsf{d} \in [\mathsf{D}]$, the memory has to move the read/write head of the disk d to the physical location addr to perform the operation. Any such movement of the head introduces cost and delays, and the machine that interacts with the memory would like to minimize the number of move head operations. Traditionally, the latter can be improved by ensuring that the program accesses contiguous regions of the memory. However, this poses a great challenge for oblivious computation in which data is often continuously shuffled across memory.

More formally, a memory is denoted as $\mathsf{mem}[N, b, \mathsf{D}]$, consisting of D disks, indexed by the address space $[N] = \{1, 2, \ldots, N\}$, where $\mathsf{D} \cdot N$ is the size of the logical memory. We refer to each memory word also as a *block* and we use b to denote the bit length of each block. The memory supports the following two types of instructions.

- **Move head operation** $(\mathsf{move}, \mathsf{d}, \mathsf{addr})$ moves the head of the d-th disk ($\mathsf{d} \in [\mathsf{D}]$) to point to address addr within that disk.
- **A read/write operation** $(\mathsf{op}, \mathsf{d}, \mathsf{data})$, where $\mathsf{op} \in \{\texttt{read}, \texttt{write}\}$, $\mathsf{d} \in [\mathsf{D}]$ and $\mathsf{data} \in \{0,1\}^b \cup \{\perp\}$. If $\mathsf{op} = \texttt{read}$, then $\mathsf{data} = \perp$ and mem should return the content of the block pointed to by the d-th disk; If $\mathsf{op} = \texttt{write}$, the block pointed to by the d-th disk is updated to data. The d-th head is then incremented to point to the next consecutive address, and wrapped around when the end of the disk is reached.

Locality A sequence of memory operations has (D, ℓ) worst-case locality if it contains ℓ move operations to a memory that is equipped with D disks.

Examples The above formalism enables us to distinguish between different degrees of locality, such that:

- An algorithm that just accesses an array sequentially can be described using a program that is $(1, O(1))$-local.
- An algorithm that computes the inner product of two vectors can be implemented with $(2, O(1))$-local (but cannot be implemented with $O(1)$ locality with 1 disk).

- An algorithm that merges two sorted arrays is $(3, O(1))$-local (and cannot be implemented with $O(1)$ locality with only 2 disks).
- An algorithm that makes N random accesses to an array is $(\mathsf{D}, \Theta(N))$-local for any constant number of D disks with overwhelming probability.

Relation to the standard memory definition Instead of specifying which disk to read from/write to, we can define a memory of range $[\mathsf{D} \cdot N] = \{1, \ldots, \mathsf{D} \cdot N\}$. The address space determines the disk index, and therefore also whether or not to move the read/write head. Thus, one can consider the regular notion of a RAM program, and our definition provides a way to measure the locality of the program. Different implementations of the same functionality can have different locality, similarly to other metrics.

3.2. *Oblivious Machines*

In this section, we define oblivious simulation of functionalities, either stateless (non-reactive) or stateful (reactive). As most prior works, we consider oblivious simulation of deterministic functionalities only. We capture a stronger notion than what is usually considered, in which the adversary is adaptive and can issue request as a function of previously observed access pattern.

Warmup: Oblivious simulation of a stateless deterministic functionality We consider machines that interact with the memory via move and `read/write` operations. In case of a stateless (non-reactive) functionality, the machine M receives one instruction I as input, interacts with the memory, computes the output and halts. Formally, we say that the stateless algorithm M obliviously simulates a stateless, deterministic functionality f w.r.t. to the leakage function $\mathsf{leakage} : \{0, 1\}^* \to \{0, 1\}^*$, iff

- **Correctness:** there exists a negligible function $\mu(\cdot)$ such that for every λ and I, $M(1^\lambda, I) = f(I)$ except with $\mu(\lambda)$ probability.
- **Obliviousness:** there exists a stateless p.p.t. simulator Sim, such that for any λ and I, $\mathsf{Addr}(M(1^\lambda, I)) \overset{\epsilon(\lambda)}{\equiv} \mathsf{Sim}(1^\lambda, \mathsf{leakage}(I))$, where $\mathsf{Addr}(M(1^\lambda, I))$ is a random variable denoting the addresses incurred by an execution of M over the input I.

Depending on whether $\overset{\epsilon(\lambda)}{\equiv}$ refers to computational or statistical indistinguishability, we say M is computationally or statistically oblivious. If $\epsilon(\cdot) = 0$, we say M is perfectly oblivious. For example, an oblivious sorting algorithm is an oblivious simulation of the functionality that receives an array and sorts it (according to some specified preference function), where the leakage function contains only the length of the array being sorted.

Oblivious simulation of a stateful functionality We often care about oblivious simulation of stateful functionalities. For example, the ordinary ORAM is an oblivious simulation of a logical memory abstraction. We define a composable notion of security for oblivious simulation of a stateful functionality below. This time, the machine M, the simulator Sim, the functionality f and the leakage function $\mathsf{leakage}$ are all interactive machines that might receive instructions as long as they are activated, and each might maintain a secret state. Moreover, we explicitly introduce the distinguisher $\mathcal{A}$, which is now also an interactive machine. In each step, the distinguisher $\mathcal{A}$ observes the access pattern and selects the next command to perform. We write $(\mathsf{out}_i, \mathsf{addr}_i) \leftarrow M(I_i)$, where out_i denotes the intermediate output of M for the instruction I_i, and addr_i denote the memory addresses accessed by M when answering the instruction I_i. We have:

Definition 3.1. (*Adaptively secure oblivious simulation of stateful functionalities*) Let M, $\mathsf{leakage}$, f be interactive machines. We say that M obliviously simulates a possibly randomized, stateful functionality f w.r.t. to the leakage function $\mathsf{leakage}$ iff there exists an (interactive) p.p.t. simulator Sim, such that for any non-uniform (interactive) p.p.t. adversary $\mathcal{A}$, $\mathcal{A}$'s view in the following two experiments, $\mathsf{Expt}_{\mathcal{A}}^{\mathrm{real},M}$ and $\mathsf{Expt}_{\mathcal{A},\mathsf{Sim}}^{\mathrm{ideal},f}$ are computationally indistinguishable.

```
$\underline{\mathsf{Expt}_{\mathcal{A}}^{\mathrm{real},M}(1^\lambda)}$:
$\mathsf{out}_0 = \mathsf{addr}_0 = \perp$
For $i = 1, 2, \ldots \mathsf{poly}(\lambda)$:
    $I_i \leftarrow \mathcal{A}(1^\lambda, \mathsf{out}_{i-1}, \mathsf{addr}_{i-1})$
    $\mathsf{out}_i, \mathsf{addr}_i \leftarrow M(I_i)$
```

```
$\underline{\mathsf{Expt}_{\mathcal{A},\mathsf{Sim}}^{\mathrm{ideal},f}(1^\lambda)}$:
$\mathsf{out}_0 = \mathsf{addr}_0 = \perp$
For $i = 1, 2, \ldots \mathsf{poly}(\lambda)$:
    $I_i \leftarrow \mathcal{A}(1^\lambda, \mathsf{out}_{i-1}, \mathsf{addr}_{i-1})$
    $\mathsf{out}_i \leftarrow f(I_i)$
    $\mathsf{addr}_i \leftarrow \mathsf{Sim}(\mathsf{leakage}(I_i))$
```

In the above definition, if we replace computational indistinguishability with statistical indistinguishability (or identically distributed resp.) and remove the requirement for the adversary to be polynomially bounded, then we say that the stateful machine M obliviously simulates the stateful functionality f with statistical (or perfect resp.) security. Besides the leakage of the individual instruction, the simulator might have some additional information in the form of the public parameters of the functionality. We also remark that Definition 3.1 captures correctness and obliviousness simultaneously, and capture both *deterministic* and *randomized* functionalities. We refer the reader to the relevant discussions in the literature of secure computation for the importance of capturing correctness and obliviousness simultaneously for the case of randomized functionalities [12,27].

Our definition of oblivious simulation is general and captures any stateless or stateful functionality, and thus later in the paper, whenever we define any oblivious algorithm, it suffices to state 1) what functionality it computes; 2) what is the leakage; and 3) what security (i.e., computational, statistical, or perfect) we achieve. We next formally define ordinary ORAM using Definition 3.1.

Ordinary ORAM The ORAM functionality is an oblivious simulation of a "logical memory functionality," parameterized by (N, b), where N is the size of the logical memory and b is the block size:

- **Functionality** The internal state of the functionality consists of an array $\mathsf{mem} \in (\{0,1\}^b)^N$. Upon each instruction of the form $(\mathsf{op}, \mathsf{addr}, \mathsf{data})$, with $\mathsf{op} \in \{\mathtt{read}, \mathtt{write}\}$, $\mathsf{addr} \in [N]$, and $\mathsf{data} \in \{0,1\}^b \cup \{\perp\}$, the functionality proceeds as follows. If $\mathsf{op} = \mathtt{write}$, then $\mathsf{mem}[\mathsf{addr}] = \mathsf{data}$. In both cases, the functionality returns $\mathsf{mem}[\mathsf{addr}]$.
- **Leakage** The simulator has the public parameters of the functionality—N and b. With each instruction $(\mathsf{op}, \mathsf{addr}, \mathsf{data})$, the leakage is just that an access has been performed.

Bandwidth and private storage of oblivious machines Throughout the paper, we use the terminology *bandwidth* to denote the total number of memory read/write operations of size $\Omega(\log N)$ a machine needs to use. We assume the machine/algorithm has only $O(1)$ blocks of private storage.

Remark In this paper, we focus on hiding the access patterns to the memory, but not the data contents. Therefore, we do not explicitly mention that data is (re-)encrypted when it is accessed, but encryption should be added since the adversary can observe memory contents. That is, while we assume that the adversary completely sees the instructions (move, d, addr) and (op, d, data) that are sent to the memory, data should be encrypted. In of all our constructions, the only information that the client stores locally is the secret key used for encrypting these data. Note that the adversary sees in particular the (encrypted) contents and accesses of all disks.

4. Locality-Friendly Building Blocks

In this section, we describe several locality-friendly building blocks that are necessary for our constructions.

4.1. *Oblivious Sorting Algorithms with Locality*

An important building block for our construction is an oblivious sorting algorithm that is locality-friendly. In "Appendix A," we describe an algorithm for Bitonic sort to achieve good locality, and provide a detailed analysis.

Oblivious sort—functionality This is a deterministic functionality in which the input is an array $A[1, \ldots, n]$ of memory blocks (i.e., each $A[i] \in \{0, 1\}^b$, representing a key). The goal is to output an array $A'[1, \ldots, n]$ which is some permutation $\pi : [n] \to [n]$ of the array A, i.e., $A'[i] = A[\pi(i)]$, such that $A'[1] \leq \ldots \leq A'[n]$. The leakage is just n, b, and obliviousness is defined using Definition 3.1.

Theorem 4.1. ((Theorem 2.2, restated) Perfectly secure oblivious sort with locality) *Bitonic sort (when implemented as in "Appendix A") is a perfectly oblivious sorting algorithm that sorts n elements using* $O(n \log^2 n)$ *bandwidth and* $(2, O(\log^2 n))$ *locality.*

In "Appendix B," we also discuss the locality guarantees of Bucket oblivious sort [1].

4.2. *Oblivious Deduplication with Locality*

We define a handy subroutine that removes duplicates obliviously. $Y \leftarrow \mathsf{Dedup}(X, n_Y)$, where X contains some real elements and dummy elements, and n_Y is some target output length. It is assumed that each real element is of the form $((k, k'), v)$ where k is a primary key and k' is a secondary key. The subroutine outputs an array Y of length n_Y in which for each primary key k in X, only the element with the smallest secondary key k' remains (possibly with some dummies at the end). It is assumed that the number of primary keys k is bounded by n_Y.

Given a locality-friendly oblivious sort, we can easily realize oblivious Dedup with locality. We obliviously sort X by the (k, k') tuple, scan X to replace duplicates with dummies, and sort X again to move dummies toward the end. Finally, pad or truncate X to have length n_Y and output. The procedure is just few scans of the array and 2 invocations of oblivious sort, and therefore, the bandwidth and locality are the same as the oblivious sort. Concretely, using Theorem 4.1 this can be implemented using $O(|X| \log^2 |X|)$-bandwidth and $(2, O(\log^2 |X|))$-locality.

4.3. *Locally Initializable ORAM*

In this section, we show that the oblivious sort can be utilized to define an (ordinary) ORAM scheme that is also locally initializable.

A locally initializable ORAM is an ORAM with the additional property that it can be initialized efficiently and in a locality-friendly manner given a batch of initial blocks. The syntax and definitions of a locally initializable ORAM is the same as a normal ORAM, except that the first operation in the sequence is a locality-friendly initialization procedure. More formally, a locally initializable ORAM is an oblivious implementation of the following functionality, parametrized by N and b:

- **Secret state** an array mem of size N and block size b. Initially all are 0.
- $\mathsf{T.Build}(X)$ takes an input array X of $|X| < N$ blocks of the form $(\mathsf{addr}_i, \mathsf{data}_i)$ where each $\mathsf{addr}_i \in [N]$ and $\mathsf{data}_i \in \{0,1\}^b$. Blocks in X have distinct integer addresses that are not necessarily contiguous. The functionality has no output, but it updates its internal state: For every $i = 1, \ldots, |X|$ it writes $\mathsf{mem}[\mathsf{addr}_i] = \mathsf{data}_i$.
- $\mathsf{B} \leftarrow \mathsf{T.Access}(\mathsf{op}, \mathsf{addr}, \mathsf{data})$ with $\mathsf{op} \in \{\texttt{read}, \texttt{write}\}$, $\mathsf{addr} \in [N]$, and $\mathsf{data} \in \{0,1\}^b$. If $\mathsf{op} = \texttt{write}$, then $\mathsf{mem}[\mathsf{addr}] = \mathsf{data}$. In both cases of $\mathsf{op} = \texttt{read}$ and $\mathsf{op} = \texttt{write}$, return $\mathsf{mem}[\mathsf{addr}]$.

The leakage function of locally initializable ORAM reveals $|X|$ and the number of Access operations (as well as the public parameters N and b). Obliviousness is defined as in Definition 3.1 with the above leakage and functionality.

Locality-friendly initialization We now show that the hierarchical ORAM by Goldreich and Ostrovsky [26] can be initialized in a locality-friendly manner, i.e., how to implement Build with $(2, O(\mathsf{poly}\log n))$ locality, where $n = |X|$. To initialize a hierarchical ORAM, it suffices to place all the n blocks in the largest level of capacity n. In the Goldreich and Ostrovsky ORAM, each block is placed into one of the n bins by applying a pseudorandom function $\mathsf{PRF}_K(\mathsf{addr})$ where K is a secret key known only to the CPU and addr is the block's address. By a simple application of the Chernoff bound, except with $\mathsf{negl}(\lambda)$ probability, each bin's utilization is upper bounded by $\alpha \log \lambda$ for any super-constant function α. Goldreich and Ostrovsky [26] show how to leverage oblivious sorting to obliviously initialize such a hash table. For us to achieve locality, it suffices to use a locality-friendly oblivious sort algorithm such as Bitonic sort. This gives rise to the following theorem:

Theorem 4.2. (Computationally secure, locally initializable ORAM) *Assuming one-way functions exist, there exists a computationally secure locally initializable ORAM scheme that has* $\mathsf{negl}(\lambda)$ *failure probability, and can be initialized with n blocks using*

$(n+\lambda) \cdot \mathsf{poly}\log(n+\lambda)$ bandwidth and $(2, \mathsf{poly}\log(n+\lambda))$ locality, and can serve an access using $\mathsf{poly}\log(n+\lambda)$ bandwidth and $(2, \mathsf{poly}\log(n+\lambda))$ locality.

Notice that for ordinary ORAMs, since the total work for accessing a singe block is only polylogarithmic, obtaining polylogarithmic locality per access is trivial. Our goal later is to achieve ORAMs where even if you access a large file or large region, the locality is still polylogarithmic, i.e., one does not need to split up the file into little blocks and access them one by one. Our constructions later will leverage a locally initializable, ordinary ORAM as a building block.

5. Range ORAM

In this section, we define range ORAM and present a construction with polylogarithmic bandwidth and polylogarithmic locality. The construction uses a building block which we call an oblivious range tree (Sect. 5.2). It supports read-only range lookup queries with low bandwidth and good locality. From an oblivious range tree, we show how to construct a range ORAM, which supports reads and updates (Sect. 5.3). Then, we discuss statistical and perfect security in Sect. 5.4. Finally, we extend Range ORAM to online Range ORAM (Sect. 5.5).

Our ORAM construction uses multiple disks only when it invokes an oblivious sort operation (and Dedup operation which invokes an oblivious sort). Thus, for the following algorithms, it can be assumed that the entire data are stored on a single disk. Multiple disks are used only transiently using during an oblivious sort or a Dedup operation.

5.1. *Range ORAM Definition*

A Range ORAM is an oblivious machine that supports read/write range instructions, and interacts with the memory while leaking only the size of the range. Formally, using Definition 3.1, Range ORAM is defined as follows, parameterized by N and b:

Functionality The internal state is an array mem of size N and blocksize b. Range ORAM takes as input range requests in the form $\mathsf{Access}(\mathsf{op}, [s, t], \mathsf{data})$, where $\mathsf{op} \in \{\mathtt{read}, \mathtt{write}\}$, $s, t \in [N]$, $s < t$, and $\mathsf{data} \in (\{0,1\}^b)^{(t-s+1)}$. If $\mathsf{op} = \mathtt{read}$, then it returns $\mathsf{mem}[s, \ldots, t]$. If $\mathsf{op} = \mathtt{write}$, then $\mathsf{mem}[s, \ldots, t] = \mathsf{data}$.
Leakage With each instruction $\mathsf{Access}(\mathsf{op}_i, [s_i, t_i], \mathsf{data}_i)$, range ORAM leaks $t_i - s_i + 1$.

5.2. *Oblivious Range Tree*

A necessary building block for our Range ORAM construction is a Range Tree. An oblivious Range Tree is a *read-only* Range ORAM with an initialization procedure from a list of blocks with possibly non-contiguous addresses. Formally, it is an oblivious simulation of the following reactive functionality with the following leakage (where obliviousness is defined using Definition 3.1):

Functionality Formally, an oblivious Range Tree T supports the following operations:

- $\mathsf{T.Build}(X)$ takes in a list X of blocks of the form $(\mathsf{addr}, \mathsf{data})$. Blocks in X have distinct integer addresses that are not necessarily contiguous. Store X as the secret state. Build has no output.
- $\mathsf{B} \leftarrow \mathsf{T.Access}(\mathtt{read}, [s, t], \bot)$ takes in a range $[s, t]$ and returns all (and only) blocks in X that has addr in the range $[s, t]$. We assume $\mathsf{len} = t - s + 1 = 2^i$ is a power of 2 for simplicity.

Leakage $\mathsf{T.Build}(X)$ leaks $|X|$. Each $\mathsf{T.Access}(\mathtt{read}, [s, t], \bot)$ leaks the value $(t - s + 1)$.

A logical Range Tree For simplicity, assume $n := |X|$ is a power of 2; if not, we simply pad with dummy blocks that have $\mathsf{addr} = \infty$. A logical Range Tree is a full binary tree with n leaves. Each leaf contains a block in X, sorted by addr from left to right. Each internal node is a *superblock*, i.e., blocks from all leaves in its subtree concatenated and ordered by addresses. A height-i superblock thus has size 2^i. The leaves are at height 0, and the root is at height $\log_2 n$.

Metadata tree Each superblock in the logical Range Tree defines a range: $[a_s, a_m, a_t]$ where a_s is the lowest address, a_t is the highest address, and a_m is the middle address (the address of the 2^{i-1}-th block for a height-i superblock). We use another full binary tree to store the range metadata of each superblock, henceforth referred to as the *metadata tree*. The metadata tree is a binary search tree that supports the following search operations:

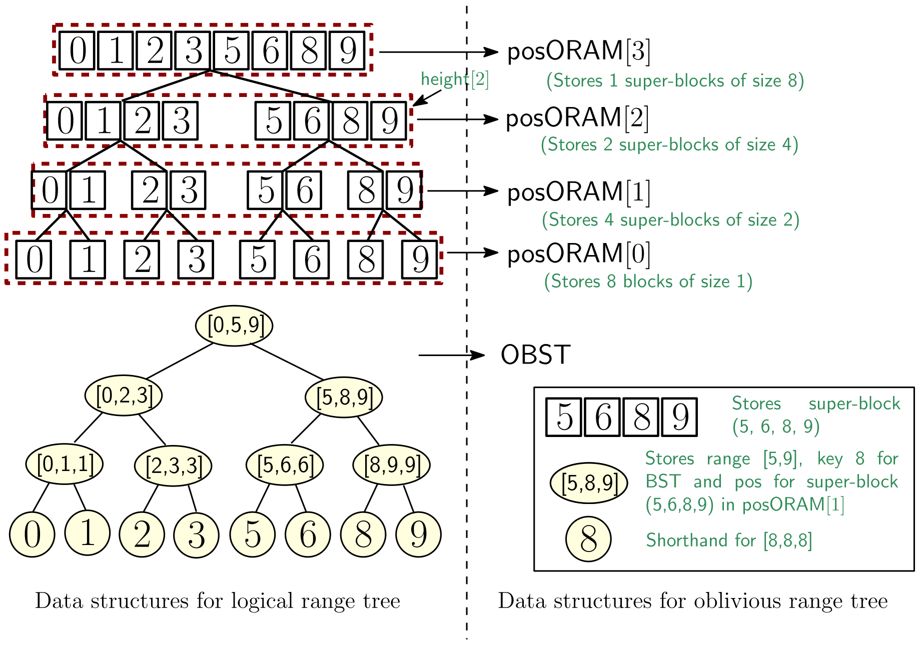
- Given a request range $[s, t]$ with $\mathsf{len} := t - s + 1 = 2^i$, find the leftmost and rightmost height-i (super)blocks whose ranges intersect $[s, t]$, or return $\bot$ if none is found.

Since $t - s + 1 = 2^i$, the leftmost and rightmost height-i (super)blocks that intersect $[s, t]$ (if they exist) are either contiguous or the same node.

Next, to achieve obliviousness, we will put the metadata tree and *each height* of the logical range tree into a separate ORAM, as shown in Fig. 2.

Algorithm 5.1. $\mathsf{T.Build}(X)$. The Build algorithm takes a list of blocks X, constructs the logical Range Tree and metadata tree, and then puts them into ORAMs through local initialization (Sect. 4.3).

1. **Create leaves** Obliviously sort X by the addresses. Pad X to the nearest power of 2 with dummy blocks that have $\mathsf{addr} = \infty$. Let $\mathsf{height}[0]$ denote the sorted X, which will be the leaves of the logical Range Tree.
2. **Create superblocks** For each height $i = 1, 2, \ldots, L := \log_2 n$, create height-$i$ superblocks by concatenating their two child nodes. Let $\mathsf{height}[i]$ denote the set of height-i superblocks. Tag each superblock with its offset in the height.
3. **Create metadata tree** Let $\mathsf{metadata}$ be the resulting metadata tree represented as an array, i.e., $\mathsf{metadata}[i]$ is the parent of $\mathsf{metadata}[2i+1]$ and $\mathsf{metadata}[2i+2]$. Tag each node in the metadata tree with its offset in $\mathsf{metadata}$.
4. **Put each height and metadata tree in ORAMs** For each height $i = 0, 1, \ldots, L$, let H_i be a locally initializable ORAM from Sect. 4.3, and call $\mathsf{H}_i.\mathsf{Build}(\mathsf{height}[i])$ in which each height-i superblock behaves as an atomic block. Let H_{meta} be a locally initializable ORAM, and call $\mathsf{H}_{\text{meta}}.\mathsf{Build}(\mathsf{metadata})$.



0 1 2 3 5 6 8 9
posORAM[3]
height[2]
(Stores 1 super-blocks of size 8)
posORAM[2]
(Stores 2 super-blocks of size 4)
posORAM[1]
(Stores 4 super-blocks of size 2)
posORAM[0]
(Stores 8 blocks of size 1)
[0,5,9]
[0,2,3]
[5,8,9]
[0,1,1]
[2,3,3]
[5,6,6]
[8,9,9]
OBST
Stores super-block (5, 6, 8, 9)
Stores range [5,9], key 8 for BST and pos for super-block (5,6,8,9) in posORAM[1]
Shorthand for [8,8,8]
Data structures for logical range tree
Data structures for oblivious range tree

Fig. 2. An oblivious Range Tree with Locality .

Algorithm 5.2. $\mathsf{T.Access}(\mathtt{read}, [s,t], \perp)$ (with $\mathsf{len} = t - s + 1 = 2^i$)

1. **Look up address** Call $\mathsf{H}_{\mathrm{meta}}.\mathsf{Access}(\cdot)$ $2L$ times to obliviously perform binary searches for the leftmost and rightmost height-i (super)blocks in the logical Range Tree that intersects $[s,t]$. Suppose they have addresses addr_1 and addr_2 (which may be the same and may both be $\perp$).
2. **Retrieve superblocks** Call $\mathsf{B}_1 \leftarrow \mathsf{H}_i.\mathsf{Access}(\mathtt{read}, \mathsf{addr}_1, \perp)$ and $\mathsf{B}_2 \leftarrow \mathsf{H}_i.\mathsf{Access}(\mathtt{read}, \mathsf{addr}_2, \perp)$ to retrieve the two (super)blocks.
3. **Output** Remove blocks from B_1 and B_2 that are not in $[s,t]$. Output $\mathsf{B} = \mathsf{Dedup}(\mathsf{B}_1 \mid\mid \mathsf{B}_2, \mathsf{len})$.

We prove the following theorem:

Theorem 5.3. (Oblivious Range Tree) *Assuming one-way functions exist, there exists a computationally secure oblivious Range Tree scheme that has correctness except with* $\mathsf{negl}(\lambda)$ *probability, and*

- $\mathsf{Build}(X)$ *requires* $n \cdot \mathsf{poly}\log(n+\lambda)$ *bandwidth and* $(2, \mathsf{poly}\log(n+\lambda))$ *locality for an input* X *of length* n,
- Access *requires* $\mathsf{poly}\log(n+\lambda)$ *bandwidth and* $(2, \mathsf{poly}\log(n+\lambda))$ *locality.*

The construction requires $O(n \log n)$ *space. Recall that* $\mathsf{Build}(X)$ *leaks the length* $|X|$ *and each* $\mathsf{Access}(\mathtt{read}, [s,t], \perp)$ *leaks the value* $(t - s + 1)$.

Proof. We start with efficiency analysis and proceed to obliviousness.

Efficiency The $\mathsf{T.Build}$ algorithm invokes the initialization procedure of $O(\log n)$ locally initializable ORAMs (Sect. 4.3); the $\mathsf{T.Access}$ algorithm invokes a polylogarithmic number of ORAM accesses, each having polylogarithmic bandwidth and $(2, \mathsf{poly}\log n)$ locality. It is also not hard to see that the other steps in the above algorithms have $O(n)$ bandwidth and $(2, O(1))$-locality.

Obliviousness We first claim the existence of adaptive simulators $\mathsf{Sim}_0, \ldots, \mathsf{Sim}_L$, where Sim_j corresponds to ORAM H_j. In addition, there exists a simulator $\mathsf{Sim}_{\mathsf{meta}}$, corresponding to $\mathsf{H}_{\mathsf{meta}}$, $\mathsf{Sim}_{\mathsf{Dedup}}$ for the algorithm Dedup, and $\mathsf{Sim}_{\mathsf{sort}}$ for the oblivious sorting algorithm. We construct a simulator for satisfying Definition 3.1 where the function f, $\mathsf{leakage}$ are as defined above.

The simulator Sim. The simulator is online, receiving leakage of instructions from the adversary and outputs memory accesses. With each instruction I:

- Build: Upon receiving leakage $|X|$, invoke $\mathsf{Sim}_{\mathsf{sort}}$ and output its output. Then, restart all simulators $\mathsf{Sim}_0, \ldots, \mathsf{Sim}_L$ where Sim_ℓ is parameterized with block size $2^\ell \cdot b$ and leakage $|X|/2^\ell$, and output their output. Activate the additional simulator $\mathsf{Sim}_{\mathsf{meta}}$ with the input $|X|$. Output the outputs of all these simulators.
- Access: Upon receiving leakage corresponding to $(t - s + 1) = 2^i$, simulate an access to a range $[s, t]$:
  1. Invoke $\mathsf{Sim}_{\mathsf{meta}}$ for L accesses, simulating the accesses to the metadata ORAM, and output them.
  2. Since $(t - s + 1) = 2^i$, we access the i-th level only. Invoke the simulator Sim_i twice, simulating two accesses to it, and appending the simulated instructions to the output.
  3. Invoke $\mathsf{Sim}_{\mathsf{Dedup}}$ on size 2^i.

The updated state of the simulator is simply the states of all activated simulators.

We show that $\mathsf{Expt}^{\mathrm{real},M}(1^\lambda)$ is indistinguishable from $\mathsf{Expt}^{\mathrm{ideal},f}(1^\lambda)$ through a sequence of hybrid experiments:

- $\mathbf{Hyb_0}(\boldsymbol{\lambda})$: This is exactly the real execution. With each instruction I received from the adversary, we hand it to the real construction to receive the memory addresses. In addition, the construction interacts with the real memory and generates the output out_i in each stage, which is also given to the adversary.
- $\mathbf{Hyb_1}(\boldsymbol{\lambda})$: Same as $\mathsf{Hyb}_1(\lambda)$, where now we use the Range Tree functionality in order to produce the output out_i in each step.
- $\mathbf{Hyb_{2,k}}(\boldsymbol{\lambda})$ with $k \in [L]$: In this execution, upon receiving some instruction I from the adversary, we proceed as follows:
  1. $\mathsf{Build}(X)$: Perform Steps 1–3 in Algorithm 5.1. Then,
     - For all $i \leq k$, call to $\mathsf{Sim}_i(|X|/2^i)$ as in the simulation.
     - For all $i > k$, perform $\mathsf{H}_i.\mathsf{Build}(\mathsf{height}[i])$.
  2. $\mathsf{Access}(\mathtt{read}, [s, t], \perp)$: (with $t - s = 2^i$), perform the following steps:
     (a) Call $\mathsf{H}_{\mathsf{meta}}$ to obliviously search for the metadata $\mathsf{addr}_1, \mathsf{addr}_2$ as in the real execution.

(b) If $i \leq k$, call to the simulator Sim_i for simulating two accesses.
(c) If $i > k$, then call to the real oblivious RAM H_i to access both addr_1 and addr_2.

In each step, output the concatenation of all memory address defined as above and proceed to the next instruction.

- $\mathbf{Hyb_3}(\lambda)$: Same as $\mathsf{Hyb}_{2,L}(\lambda)$, except that the metadata ORAM is replaced with $\mathsf{Sim}_{\mathsf{meta}}$.
- $\mathbf{Hyb_4}(\lambda)$: Same as $\mathsf{Hyb}_3(\lambda)$ except that we replace Dedup with $\mathsf{Sim}_{\mathsf{Dedup}}$.
- $\mathbf{Hyb_5}(\lambda)$: Same as $\mathsf{Hyb}_4(\lambda)$ except that we replace the oblivious sort with $\mathsf{Sim}_{\mathsf{sort}}$.

As a result, the experiment uses only the leakage of the instruction, and this is exactly the simulator Sim.

We show that for every adversary $\mathcal{A}$, its view in each one of the hybrid experiment is indistinguishable. Specifically, $\mathsf{Hyb}_5(\lambda)$ is indistinguishable from $\mathsf{Hyb}_4(\lambda)$ due to the security of the oblivious sorting algorithm. The view of the adversary in $\mathsf{Hyb}_4(\lambda)$ is indistinguishable from its view in $\mathsf{Hyb}_3(\lambda)$, due to the security of the Dedup function. The view of the adversary $\mathsf{Hyb}_3(\lambda)$ is indistinguishable from $\mathsf{Hyb}_{2,L}(\lambda)$ due to the security of the metadata ORAM. In a more detail, assume by contradiction that there exists an adversary $\mathcal{A}$ that succeeds to distinguish between $\mathsf{Hyb}_3(\lambda)$ and $\mathsf{Hyb}_{2,L}(\lambda)$. We show the existence of an adversary $\mathcal{A}'$ that succeeds to distinguish between $\mathsf{Expt}^{\mathrm{real},M_{\mathsf{ORAM}}}_{\mathcal{A}'}(1^\lambda)$ and $\mathsf{Expt}^{\mathrm{ideal},f_{\mathsf{ORAM}}}_{\mathcal{A}',\mathsf{Sim}_{\mathsf{meta}}}(1^\lambda)$ as follows:

1. $\mathcal{A}'$ is activated with input $(1^\lambda, \perp, \perp)$ and activates $\mathcal{A}$ on the same input.
2. Upon receiving an instruction $I = \mathsf{Build}(X)$ or $I = \mathsf{Access}(\texttt{read}, [s, t], \perp)$ from $\mathcal{A}$, the adversary $\mathcal{A}'$ simulates the hybrid experiment, in which all levels $\mathsf{T}_0, \ldots, \mathsf{T}_L$ are simulated using $\mathsf{Sim}_0, \ldots, \mathsf{Sim}_L$, and invocations of Dedup and sort are the real constructions. In order to simulate instructions to $\mathsf{H}_{\mathsf{meta}}$, $\mathcal{A}'$ outputs that instruction to its own challenger, receives the output and the memory addresses and uses them to answer $\mathcal{A}$ instruction I.
3. When $\mathcal{A}$ outputs a bit b distinguishing between $\mathsf{Hyb}_3(\lambda)$ and $\mathsf{Hyb}_{2,L}(\lambda)$, the adversary $\mathcal{A}'$ uses this bit to distinguish between interacting with $\mathsf{Sim}_{\mathsf{meta}}$ and the corresponding real ORAM construction.

Likewise, for every $k \in \{0, \ldots, L-1\}$ it holds that the view of the adversary in $\mathsf{Hyb}_{2,k}(\lambda)$ is indistinguishable from $\mathsf{Hyb}_{2,k+1}(\lambda)$ due to the security of the $k+1$th ORAM. Finally, $\mathsf{Hyb}_{2,0}(\lambda, \mathbf{I})$ and $\mathsf{Hyb}_1(\lambda)$ are indistinguishable due to the security of the ORAM T_0. Finally, $\mathsf{Hyb}_1(\lambda)$ and $\mathsf{Hyb}_0(\lambda)$ are indistinguishable due to the correctness of the Range Trees. $\square$

5.3. *Range ORAM from Oblivious Range Tree*

In this section, we show how to construct a Range ORAM from oblivious Range Tree scheme. Since the underlying oblivious Range Tree has good efficiency/locality, so will the resulting Range ORAM. The idea behind our construction is similar to that of the standard hierarchical ORAM [25,26]. Intuitively, where a standard hierarchical ORAM employs an oblivious hash table, we instead employ an oblivious Range Tree.

Data structure We use N to denote both the total size of logical data blocks as well as the security parameter.[4] There are $\log N + 1$ *levels* numbered $0, 1, \ldots, L$, respectively, where $L := \lceil \log_2 N \rceil$ is the maximum level. Each level is an oblivious Range Tree denoted $\mathsf{T}_0, \mathsf{T}_1, \ldots, \mathsf{T}_L$ where T_i has capacity 2^i. Data will be replicated across these levels. We maintain the invariant that data in lower levels are fresher. At any time, each T_i can be in two possible states, *non-empty* or *empty*. Initially, the largest level is marked non-empty, whereas all other levels are marked empty.

Algorithm 5.4. Range ORAM $\mathsf{Access}(\mathsf{op}, [s, t], \mathsf{data})$ (with $t - s + 1 = 2^i$ for some i).

1. Retrieve all blocks in range trees of capacity no more than 2^i, i.e., $\mathsf{fetched} := \cup_{j=0}^{i-1} \mathsf{T}_j$. This can be easily done by fetching its root. Mark blocks in $\mathsf{fetched}$ that are not in the range $[s, t]$ as dummy.

   Each real block in $\mathsf{fetched}$ is tagged with its level number j as a secondary key so that later after calling $\mathsf{Dedup}(\mathsf{fetched}, t - s + 1)$, where Dedup is defined in Sect. 4.2, only the most fresh version of each block remains. We assume each block also carries a copy of its address.
2. For each $j = i, i + 1, \ldots, L$, if T_j is non-empty, let $\mathsf{fetched} = \mathsf{fetched} \cup \mathsf{T}_j$. $\mathsf{Access}(\mathtt{read}, [s, t], \perp)$.
3. Let $\mathsf{data}^* := \mathsf{Dedup}(\mathsf{fetched}, 2^i)$. If $\mathsf{op} = \mathtt{read}$, then data^* will be returned at the end of the procedure. Else, $\mathsf{data}^* := \mathsf{data}$.
4. If all levels $\leq i$ are marked empty then perform $\mathsf{T}_i.\mathsf{Build}(\mathsf{data}^*)$ and mark it as ready. Otherwise:
   (a) Let ℓ denote the smallest level greater than i that is empty. If no such level exists, let $\ell := L$.
   (b) Let $S := \cup_{j=0}^{\ell-1} \mathsf{T}_j$. If $\ell = L$, additionally include $S := S \cup T_L$. Call $\mathsf{T}_\ell.\mathsf{Build}(\mathsf{Dedup}(S, 2^\ell))$ and $\mathsf{T}_i.\mathsf{Build}(\mathsf{data}^*)$. Mark levels ℓ and i as non-empty, and all other levels below ℓ as empty.

Example 5.5. We show a simple example for how levels are updated after some accesses. We assume initially that all blocks are stored in the largest Range Tree. Consider the following sequence of ranges [1, 1], [2, 3], [4, 5], [6, 6] accessed.

- Access [1, 1]: A block of size 1. Added to T_0.
- Access [2, 3]: A block of size 2, and so $i = 1$. Levels $\leq i$ are not empty. The smallest empty level larger than $i = 1$ is 2. Thus, move [1, 1] to T_2 (which has capacity 4), and then put [2, 3] to T_1. At this point, T_0 is empty and T_1 and T_2 are occupied.
- Access [4, 5]: A block of size 2, and so $i = 1$. Levels $\leq i$ are not empty. The smallest empty level larger than $i = 1$ is 3. Thus, move $\{1, 2, 3\}$ to T_3 (which has capacity 8), and then put [4, 5] to T_1. At this point, T_0 and T_2 are empty, and T_1 and T_3 are occupied.

[4] Recall that we use n to denote the size of an instance of the underlying building block, in our case, an Oblivious Range Tree, and N to denote the total size of the memory.

- Access [6, 6]: A block of size 1, and so $i = 0$. Levels $\leq i$ are empty. [6, 6] is added to T_0. At this point, T_2 is empty, and T_0, T_1 and T_3 are occupied.

Theorem 5.6. (Range ORAM) *Assuming one-way functions exist, there exists a computationally secure Range ORAM consuming* $O(N \log N)$ *space with* $\mathsf{negl}(N)$ *failure probability, and* $\mathsf{len} \cdot \mathsf{poly} \log N$ *bandwidth and* $(2, \mathsf{poly} \log N)$ *locality for accessing a range of size* len*. Each access to a range of size* len *leaks the value* len*.*

We remark bandwidth is in an amortized sense: for accessing contiguous regions of lengths $\mathsf{len}_1, \ldots, \mathsf{len}_m$, the total bandwidth is $\left(\sum_{i=1}^{m} \mathsf{len}_i\right) \cdot \mathsf{poly} \log N$. Locality is worst case, for each access of these regions $\mathsf{len}_1, \ldots, \mathsf{len}_m$ the locality is $(2, \mathsf{poly} \log N)$, i.e., the total locality of the m accesses is $(2, m \cdot \mathsf{poly} \log N)$.

Proof. We start with efficiency analysis and proceed to obliviousness.

Efficiency We now analyze the efficiency and locality of our Range ORAM.

- Space: Level i is a Range Tree that requires $O(2^i \cdot \log 2^i)$ space. Therefore, the total space can be bounded by $\sum_{i=0}^{\log N+1} O(2^i \log 2^i) \leq \sum_{i=0}^{\log N+1} O(2^i \log N) = O(N \log N)$.
- *Read phase.* The read phase (Step 1 to 3) invokes one access to each of the $O(\log n)$ oblivious Range Trees and hence has $\mathsf{len} \cdot \mathsf{poly} \log N$ bandwidth and $(2, \mathsf{poly} \log N)$ locality.
- *Rebuild phase.* An alternative way to view our algorithm is to think all each levels' empty bit (where empty denotes 0 and non-empty denotes 1), when concatenated, form a binary counter. Level i is rebuilt every 2^i counter increments. Rebuilding a level i involves initializing (building) the underlying oblivious Range Tree, which costs $n \cdot \mathsf{poly} \log n$ bandwidth and locality where $n = 2^i$. Thus, the per-increment bandwidth for rebuilding is $\mathsf{poly} \log N$—recall that N is both the total logical memory size and the security parameter. It is not hard to see that every time a memory range of size 2^i is requested, the counter's value increases by at most $3 \cdot 2^i$. So the amortized bandwidth for rebuilding is $O(\mathsf{len} \cdot \mathsf{poly} \log N)$ for an access requesting len blocks. The locality of the rebuild phase is straightforward: every access request involves rebuilding at most 2 levels.

Correctness The key argument for correctness of Algorithm 5.4 is to ensure that for any address, if it is stored at a smaller level, the smaller level contains fresher data. Let us show this by contradiction. Suppose this is violated, i.e., for some address addr, T_j contains a fresher version than T_i $(j > i)$. When addr was written to T_j, a stale version of addr is in T_i. However, addr was written to T_j by either steps 4 or 4a. In both cases, all levels $\leq i$ were empty.

Obliviousness Let $\mathsf{Sim}_0, \ldots, \mathsf{Sim}_L$ denote the simulators of the Range Trees. Let $\mathsf{Sim}_{\mathsf{Dedup}}$ denote the simulator for the Dedup algorithm. Consider the functionality of Range ORAM as defined in Sect. 5.1. We show the existence of an online simulator Sim for Range ORAM, participating in the experiment $\mathsf{Expt}^{\mathrm{ideal}, f_{\mathsf{RangeORAM}}}_{\mathcal{A},\mathsf{Sim}}(1^{\lambda})$, defined as follows:

The simulator Sim. Upon initialization, initialize $L+1$ bits corresponding to whether a level is ready or empty. Mark all levels as empty, except for the last level. Invoke Sim_L with leakage 2^L.

Access: Upon receiving $\mathsf{leakage}(I)$ with $\mathsf{leakage}(I) = 2^i$ for some integer i

1. For $j = 0, \ldots, i-1$, access the memory locations devoted to Sim_j.
2. For $j = i, \ldots, L$, if the level j is marked ready, invoke the simulator Sim_j on simulating an access with leakage 2^i.
3. Invoke the simulator $\mathsf{Sim}_{\mathsf{Dedup}}(2^i)$.
4. If all levels $\leq i$ are marked empty, then invoke Sim_i with Build and leakage 2^i. Otherwise,
   (a) Let ℓ denote the smallest level greater than i that is empty. If no such level exists, let $\ell = L$.
   (b) Call $\mathsf{Sim}_{\mathsf{Dedup}}(2^\ell)$. Terminate all running simulators $\mathsf{Sim}_0, \ldots, \mathsf{Sim}_\ell$ and mark all corresponding bits as empty. Restart Sim_ℓ with Build on leakage 2^ℓ, and Sim_i with leakage 2^i, and mark corresponding bits as ready.

The internal state of the simulator is the bits indicating whether a level is ready/empty, and the internal states of the underlying simulators.

We show that the adversary cannot distinguish between a real execution and the ideal one. We show that through a sequence of hybrids:

- $\mathbf{Hyb_0}(\lambda)$: This is exactly the real execution. Upon receiving instruction $I = (\mathsf{op}, [s,t], \mathsf{data})$ from the adversary, we invoke Algorithm 5.4 and output the memory addresses it produces, and out_i.
- $\mathbf{Hyb_1}(\lambda)$: Same as $\mathsf{Hyb}_0(\lambda)$ but the adversary receives in each step the output of the Range ORAM functionality and not the output of the construction. The memory addresses are still according to the construction.
- $\mathbf{Hyb_{2,k}}(\lambda)$ with $k \in \{0, \ldots, L\}$: In this hybrid, we replace all range trees $0, \ldots, k-1$ with simulators $\mathsf{Sim}_0, \ldots, \mathsf{Sim}_{k-1}$.
  Upon receiving some instruction $I = (\mathsf{op}, [s,t], \mathsf{data})$ with $t-s+1 = 2^i$ for some integer i, we follow Algorithm 5.4. Whenever the algorithm performs $\mathsf{T}_j.\mathsf{Build}(X)$ for some $j < k$ and some X, we replace it with an invocation of Sim_j for Build instruction with leakage $|X|$. Whenever the Algorithm performs $\mathsf{T}_j.\mathsf{Access}$, we invoke Sim_j for Access with leakage 2^i.
- $\mathbf{Hyb_3}(\lambda)$: Same as $\mathsf{Hyb}_{2,L}(\lambda)$, where here also the Dedup algorithm is replaced with $\mathsf{Sim}_{\mathsf{Dedup}}$. As a result, we do not use any information in the instruction I beyond $\mathsf{leakage}(I)$, and this is exactly the simulator Sim.

We now claim that the view of the adversary in the experiment $\mathsf{Hyb}_3(\lambda)$ is indistinguishable from its view in $\mathsf{Hyb}_{2,\mathsf{L}}(\lambda)$ due to the security of the Dedup Algorithm. Likewise, for every $k \in \{0, \ldots, L-1\}$ it holds that the view of the adversary in $\mathsf{Hyb}_{2,\mathsf{k}}(\lambda)$ is indistinguishable from its view in $\mathsf{Hyb}_{2,\mathsf{k+1}}(\lambda)$ due to the security of the $k+1$th Tree ORAM. The views in $\mathsf{Hyb}_{2,0}(\lambda)$ and $\mathsf{Hyb}_1(\lambda)$ are identical. Finally, the views in $\mathsf{Hyb}_1(\lambda)$ and $\mathsf{Hyb}_0(\lambda)$ are indistinguishable from the correctness of the Range ORAM construction. $\square$

5.4. *Perfectly Secure Range ORAM*

The computational security in our construction is due to the use of a computationally secure locally initializable hierarchical ORAM (Theorem 4.2).

We can achieve perfect security by making the perfectly secure ORAM construction with polylogarithmic bandwidth in Chan et al. [18] locally initializable.

For a hierarchical ORAM, within each level, the position of a data block is determined by applying a PRF to the block's logical address. To achieve perfect security, Chan et al. [18] replace the PRF with a truly random permutation. To access a block within a level, the client must first figure out the block's correct location within the level. If the client had linear storage, it could simply store the locations (or position labels). To achieve small client storage, Chan et al. recursively store the position labels in a smaller ORAMs, similar to the idea of recursion in tree-based ORAMs [37]. Thus, there are logarithmically many ORAMs (each is a perfectly secure hierarchical ORAM), where the ORAM at depth d stores position labels for the ORAM at depth $d+1$; and finally, the ORAM at the maximum depth $D = O(\log N)$ stores the real data blocks.

The Build procedure for one ORAM depth relies only on oblivious sorts and linear scans, and thus consumes $(2, \mathsf{poly}\log N)$ locality using locality-preserving Bitonic sort. The Build procedure for one ORAM depth outputs its position map, which is subsequently used to initialize the next ORAM depth. Thus, all ORAM depths combined can be initialized with $(2, \mathsf{poly}\log N)$ locality. Thus, we have the following theorem.

Theorem 5.7. (Perfectly secure Range ORAM) *There exists a perfectly secure Range ORAM consuming $O(N \log N)$ space, $\mathsf{len} \cdot \mathsf{poly}\log N$ bandwidth and $(2, \mathsf{poly}\log N)$ locality for accessing a range of size len.*

5.5. *Online Range ORAM*

So far, our range ORAM assumes an abstraction where we have foresight on how many contiguous locations of logical memory we wish to access. We now consider an *online* variant, where the memory requests arrive one by one just as in normal ORAM. Formally:

Functionality: A logical memory functionality that supports the following types of instructions:

- $(\mathsf{op}, \mathsf{addr}, \mathsf{data})$: where $\mathsf{op} \in \{\mathtt{read}, \mathtt{write}\}$, $\mathsf{addr} \in [N]$ and $\mathsf{data} \in \{0,1\}^b \cup \{\bot\}$. If $\mathsf{op} = \mathtt{write}$, then write $\mathsf{mem}[\mathsf{addr}] = \mathsf{data}$. In both cases, return $\mathsf{mem}[\mathsf{addr}]$.

Leakage: Consider a sequence of requests $\mathbf{I} = ((\mathsf{op}_1, \mathsf{addr}_1, \mathsf{data}_1), \ldots, (\mathsf{op}_i, \mathsf{addr}_i, \mathsf{data}_i), \ldots)$. Each instruction leaks one bit indicating whether the last instruction is contiguous, i.e., for every i, the leakage is 1 iff $\mathsf{addr}_{i+1} = \mathsf{addr}_i + 1$.

Blackbox construction of online range ORAM from range ORAM. Given a range ORAM construction, we can convert it to an online range ORAM scheme as follows, incurring only logarithmic further blowup. Intuitively, the idea is to prefetch a contiguous region of size 2^k every time a 2^k contiguous region has been accessed. That is, if a contiguous region of overall size 2^k is being read, then it is fetched as k distinct blocks of size $1, 2, 4, 8, \ldots, 2^k$. The detailed construction is given below:

Let $\mathsf{prefetch}$ be a dedicated location in memory storing prefetched contiguous memory regions. In the array $\mathsf{prefetch}$, each element $\mathsf{prefetch}[i]$ stores a pair of addr, data, where addr is the logical address of the element and data is its current data. Moreover, we initialize $\mathsf{rsize} := 1$, $p = 1$, and let $\mathsf{prefetch} := \perp$, and denote the underlying Range ORAM as rORAM. Upon receiving a memory request $(\mathsf{op}, \mathsf{addr}, \mathsf{data}^*)$ with $\mathsf{op} \in \{\mathtt{read}, \mathtt{write}\}$, $\mathsf{addr} \in [N]$ and $\mathsf{data}^* \in \{0, 1\}^b \cup \{\perp\}$:

1. If $\mathsf{prefetch}[p].\mathsf{addr} \neq \mathsf{addr}$ then do the following.
   (a) Perform $\mathsf{rORAM.Access}(\mathtt{write}, \mathsf{prefetch}[1].\mathsf{addr}, \mathsf{prefetch}[\mathsf{rsize}].\mathsf{addr}, \mathsf{prefetch})$, i.e., we write back the entire $\mathsf{prefetch}$ into the range ORAM.
   (b) $\mathsf{prefetch} \leftarrow \mathsf{rORAM.Access}(\mathtt{read}, \mathsf{addr}, \mathsf{addr} + 1, \perp)$. That is, request a region of length 1 consisting of only the requested logical address and store the result in $\mathsf{prefetch}$.
   (c) Reset $p := 1$ and $\mathsf{rsize} := 1$;
2. If $\mathsf{op} = \mathtt{write}$, then update $\mathsf{prefetch}[p].\mathsf{data} = \mathsf{data}^*$.
3. Let $v := \mathsf{prefetch}[p]$. Increment $p := p + 1$.
4. If $p > \mathsf{rsize}$, then do the following.
   (a) Perform $\mathsf{rORAM.Access}(\mathtt{write}, \mathsf{prefetch}[1].\mathsf{addr}, \mathsf{prefetch}[\mathsf{rsize}].\mathsf{addr}, \mathsf{prefetch})$., i.e., write $\mathsf{prefetch}$ back into the range ORAM.
   (b) $\mathsf{prefetch} = \mathsf{rORAM.Access}(\mathtt{read}, \mathsf{prefetch}[\mathsf{rsize}].\mathsf{addr} + 1, 2\mathsf{rsize})$.
   (c) Set $p := 1$ and $\mathsf{rsize} := 2 \cdot \mathsf{rsize}$.
5. Return v.

It is not hard to see that given the above algorithm, accessing each range of size R will be broken up into at most $O(\log R)$ accesses, to regions of sizes $1, 2, 4, \ldots, R$, respectively, and each size has one read request and one write request. Security is straightforward as range ORAM is oblivious, and the transformation between the leakage profiles of online range ORAM and range ORAM is straightforward. We have the following theorem:

Theorem 5.8. (Online Range ORAM) *There exists a perfectly secure online Range ORAM, which on receiving* len *consecutive memory locations online performs* $\mathsf{len} \cdot \mathsf{poly}\log N$ *bandwidth and achieves* $(2, \mathsf{poly}\log N)$ *locality.*

Proof. Accessing each range of size R is broken up into at most $\lfloor \log R \rfloor + 1$ access, to regions of sizes $1, 2, 4, \ldots, 2R$, respectively, and each size has one read request and one write request. Thus, given a perfect Range ORAM with $\mathsf{len} \cdot \mathsf{poly}\log N$ amortized bandwidth and $(2, \mathsf{poly}\log N)$ locality for accessing a range of size len, an access of a range R to the Online Range ORAM results in amortized bandwidth of $\sum_{i=0}^{\lfloor \log R \rfloor + 1} 2^i \cdot \mathsf{poly}\log N \leq 4R\mathsf{poly}\log N$. Thus, there is no blowup in amortized bandwidth. However, we do have blowup in locality. Assume that the range ORAM has $(2, \log^c N)$ worst-case locality regardless of the size of the range being accessed. Our Online ORAM performs

$O(\log R)$ accesses when accessing a consecutive range of size R, and therefore, it results in $(2, O(\log R) \cdot log^c N)$ locality, i.e., $(2, O(\log^{c+1} N))$-locality.

For security, recall that with each request, the leakage profile tells us whether the requested address is consecutive. That is, it tells us whether the condition in Step 1 is satisfied or not. Let $\mathsf{Sim}_{\mathsf{rORAM}}$ be the simulator of the Range ORAM. The (online) simulator performs as follows:

1. Receive as leakage whether the accessed element is a continuation of the previous access. If it is not:
   (a) Call the simulator $\mathsf{Sim}_{\mathsf{rORAM}}$ on access with range of size rsize.
   (b) Call the simulator $\mathsf{Sim}_{\mathsf{rORAM}}$ on access with range of size 1.
   (c) Reset $p := 1$ and $\mathsf{rsize} := 1$.
2. Increment p. If $p > \mathsf{rsize}$:
   (a) Call $\mathsf{Sim}_{\mathsf{rORAM}}$ on access with range of size rsize.
   (b) Call $\mathsf{Sim}_{\mathsf{rORAM}}$ on access with range of size $2 \cdot \mathsf{rsize}$.
   (c) Let $\mathsf{rsize} = 2\mathsf{rsize}$

Clearly, since the simulator receives the leakage from the functionality, the internal variables p, rsize of the Online ORAM are identical to that of the simulator. The accesses performed by the Online ORAM are determined by those variables, and whether or not the condition in Step 1 holds. As a result, the access pattern of the Online Range ORAM is simulatable. □

6. Lower Bound for More Restricted Leakage

In Sect. 5.5, the online range ORAM leaks which instructions form a contiguous group of addresses. In this section, we show that if we restrict the leakage and do not allow the adversary to learn whether adjacent instructions access contiguous addresses, the lower bound for bandwidth to achieve locality will be significantly worse.

Model assumptions We first clarify the model in which we prove the lower bound.

1. We restrict the leakage such that the adversary knows only the number N of logical blocks stored in memory, and the total number T of online operations, each of which has the form $(\mathsf{op}, \mathsf{addr}, \mathsf{data})$, where $\mathsf{op} \in \{\texttt{read}, \texttt{write}\}$, $\mathsf{addr} \in [N]$ and $\mathsf{data} \in \{0, 1\}^b \cup \{\perp\}$.
2. Just like earlier ORAM lower bounds [9,25,26]), we assume the so-called *balls-and-bins model*, i.e., the blocks are opaque objects and the algorithm, for instance, cannot use encoding techniques to combine blocks in the storage. Note that all known ORAM algorithms indeed fall within this model.
3. We assume that the algorithm has an offline phase in which it can preprocess memory before seeing any instructions. However, recall that the instructions are online, i.e., the algorithm must finish serving an instruction before seeing the next one.

Notation. Recall that we use D to denote the number of disks (each of which has a single head), ℓ to denote the locality (where we consider the very general case $\ell \leq \frac{N}{10}$), m to denote the memory size blowup,[5] and β to denote the bandwidth. Moreover, suppose the CPU has only c block of local cache, where we just need a loose bound $c \leq \frac{N}{10}$. We shall prove the following theorem.

Theorem 6.1. *For any $\ell, c \leq \frac{N}{10}$, any Online Range ORAM satisfying the restricted leakage that has (D, ℓ)-locality with c blocks of cache storage will incur $\Omega(\frac{N}{\mathsf{D}})$ bandwidth.*

Proof intuition. By our leakage restriction assumption, the adversary cannot distinguish between the following two scenarios.

1. There are N operations that access contiguous addresses in the order from 0 to $N-1$.
2. There are N operations, each of which access an address chosen independently uniformly at random from $[N]$.

Observe that to achieve (D, ℓ)-locality, in scenario 1, there can be at most ℓ jumping moves for the disk heads. Therefore, the same must hold for scenario 2. To serve an online request in scenario 2, we consider the following cases.

1. The block of the requested address is already in the cache. (However, the ORAM might still pretend to do some accesses.) Observe this happens with probability at most $\frac{c}{N} \leq \frac{1}{10}$, since the next requested address is chosen independently uniformly at random.
2. The online request is served by some disk head jump, which takes $O(1)$ physical accesses. Again, the ORAM might make other accesses to hide the access pattern. Observe at most $\ell \leq \frac{N}{10}$ requests can be served this way.
3. The online request is served by linear scan of the disk heads. By the Chernoff Bound, except with $e^{-\Theta(N)}$ probability, at least $\frac{N}{2}$ of the requests are served by linear scan. The following lemma gives a stochastic lower bound on the number of physical accesses in this case.

For ease of notation, we assume that $K := \frac{N-c}{\mathsf{D}}$ is an integer.

Lemma 6.2. (Stochastic Lower Bound on the Number of Physical Accesses) *Suppose in Scenario 2, the block of the next random address requested is not in the ORAM's cache. Moreover, suppose this request is served by only linear scan of disk heads, i.e., no jump move is made. Then, the random variable of the number of physical accesses for serving this request stochastically dominates the random variable with uniform distribution on $\{1, 2, \ldots, \frac{N-c}{\mathsf{D}}\}$.*

Proof. Consider some configuration of the disk heads. Without loss of generality, assume that the cache currently stores the blocks for exactly c distinct addresses. For each of the remaining $N-c$ addresses, we can assign it to the disk head that takes a minimum number of accesses to reach a corresponding block by linear scan, where a tie can be resolved arbitrarily. For each $j \in [\mathsf{D}]$, let a_j be the number of addresses assigned to disk head j; observe that we have $\sum_{j \in [\mathsf{D}]} a_j = N - c$.

[5] However, as we shall see, m does not play a role in the lower bound.

For each integer $1 \le i \le K = \frac{N-c}{\mathsf{D}}$, observe that the number of addresses that take at least i physical accesses to reach is at least $\sum_{j\in[\mathsf{D}]} \max\{0, a_j - i + 1\} \ge \mathsf{D} \cdot (K - i + 1)$, where the last equality holds when all a_j's equal K.

Hence, the probability that at least i physical accesses is needed is at least $\frac{\mathsf{D}\cdot(K-i+1)}{N-c} = \frac{K-i+1}{K}$, which implies the required result. □

Lemma 6.3. (Lower Bound on Bandwidth) *Except with probability at most $e^{-\Theta(N)}$, the average number of physical accesses to serve each request in Scenario 2 is at least $\Omega(\frac{N}{D})$.*

Proof. As observed above, except with at most $e^{-\Theta(N)}$ probability, at least $\frac{N}{2}$ of the online requests must be served by linear scan of disk heads. By Lemma 6.2, the number of physical accesses for each such request stochastically dominates the uniform distribution on $\{1, 2, \ldots, \frac{N-c}{\mathsf{D}}\}$, which has expectation $\Theta(\frac{N}{\mathsf{D}})$, since we assume the cache size $c \le \frac{N}{10}$.

Since the addresses of the online requests are picked independently after the previous requests are served, by Chernoff bound, except with probability $e^{-\Theta(N)}$, the average number of physical accesses to serve each such online request is at least $\Omega(\frac{N}{\mathsf{D}})$, as required. □

7. File ORAM

In this section, we show how to construct a FileORAM scheme. We first define this new primitive in Sect. 7.1. In Sect. 7.2, we construct a non-recurrent read-only file hashing primitive, which is a FileORAM that supports only read accesses and achieves obliviousness if no file is accessed more than once. Finally, we show how to rely on core ideas behind the hierarchical ORAM framework to obtain a full-fledged FileORAM from non-recurrent read-only file hashing schemes (see Sect. 7.3).

Bucket oblivious sort In this section, we aim for concrete efficiency. Toward that end, we use a sorting algorithm with locality that is statistically secure, proposed in [1]. In "Appendix B," we give an overview of the algorithm, and prove its locality properties. We have:

Theorem 7.1. *Let $\alpha \in \omega(1)$ and $n > \log^2 \lambda$. The Bucket oblivious sort implements the sorting functionality except for $\mathsf{negl}(\lambda)$ probability, it uses $O(1)$ client storage, takes $O(n \log n \log^2(\alpha \log \lambda))$ work and $O(3, \log n \log^2(\alpha \log \lambda))$-locality.*

7.1. *Definition*

Compared to a Range ORAM which supports accesses to any contiguous memory region, a FileORAM provides a more constrained functionality that supports accesses to a set of files of predefined ranges. More specifically, in Range ORAM every memory range $[s, t]$ can be accessed (i.e., for any arbitrary values of s, t as long as $t > s$). In contrast, in FileORAM, the number of allowed ranges is known and fixed in advance and do not overlap. A FileORAM should leak only the length of the requested file but should hide any additional information. In particular, this includes the number of files of each length, or to distinguish between whether two different files of the same length are requested, or the same file is requested twice. We assume that each file is padded to the next power of 2 blocks, which at most doubles the memory consumption. The $\mathsf{FileORAM}$ functionality is defined as follows, and its oblivious simulation is defined using Definition 3.1 where the allowed leakage is as follows:

Functionality The requests are in the form $\mathsf{Access}(\mathsf{op}, \mathsf{fid}, \mathsf{data})$ where $\mathsf{op} \in \{\texttt{read}, \texttt{write}\}$, $\mathsf{fid} \in [k]$ and $\mathsf{data} \in (\{0,1\}^b)^{\mathsf{len}(\mathsf{fid})}$. Initially, $\mathsf{state} = \emptyset$, where state is an internal state that stores pairs of file identifiers and the associated data. If $\mathsf{op} = \texttt{write}$ then: If $\mathsf{fid} \notin \mathsf{state}$, then add $(\mathsf{fid}, \mathsf{data})$ to state; Otherwise, update its associated value to data. If $\mathsf{op} = \texttt{read}$ then look for fid in state and assign the associated value to data. If $\mathsf{fid} \notin \mathsf{state}$, then assign $\perp$. In all cases, return data.
Leakage Let $\mathsf{len}(\mathsf{fid})$ be the length of file fid. For simplicity, we assume the length of each file $\mathsf{len}(\mathsf{fid})$ is a power of 2 (otherwise, pad each file to the next power of 2). Let N be a bound on the sum of lengths of all files. The leakage is simply the length of each accessed file. Formally, given a sequence of instructions

$$\mathbf{I} = ((\mathsf{op}_1, \mathsf{fid}_1, \mathsf{data}_1), \ldots, (\mathsf{op}_m, \mathsf{fid}_m, \mathsf{data}_m)) \text{ , wedefine}$$
$$\mathsf{leakage}(\mathbf{I}) := (N, \mathsf{len}(\mathsf{fid}_1), \ldots, \mathsf{len}(\mathsf{fid}_m)) \text{ .}$$

7.2. *Non-Recurrent File Hashing Scheme with Locality*

A file hashing scheme H receives a set of files as input and builds a hash table that allows quick read requests. We consider a relaxed notion of security, in which obliviousness is guaranteed when the adversary is restricted to only access each (non-dummy) file at most once.
Functionality The functionality supports the following instructions:

- $\mathsf{H.Build}(X)$ takes as input an array X, where each element is of the form $(\mathsf{fid}_i, j, \mathsf{data}_j)$. fid_i is a file identifier and $\mathsf{data}_j \in \{0,1\}^b$ is the j-th block of the file fid_i. The length of a file fid_i is the maximal j for which $(\mathsf{fid}_i, j, \mathsf{data}_j) \in X$. It is assumed that for each given file fid_i with length len, exactly one block of the form $(\mathsf{fid}_i, j, \mathsf{data}_j)$ for each $j < \mathsf{len}$ exists in X. The Build instruction simply set the secret state state to be X.
- $F \leftarrow \mathsf{H.Read}(\mathsf{fid}, \mathsf{len})$ takes in a possibly dummy file identifier fid to be fetched of purported length len and returns F, the blocks of the file fid in the secret state state. If the file fid does not exist in state, or if $\mathsf{fid} = \perp$, then $F = \perp$.

Leakage The allowed leakage is as follows. For a sequence

$$\mathbf{I} = (\mathsf{Build}(X), \mathsf{Read}(\mathsf{fid}_1, \mathsf{len}_1), \ldots, \mathsf{Read}(\mathsf{fid}_m, \mathsf{len}_m)) \text{ , wedefine}$$
$$\mathsf{leakage}(\mathbf{I}) := (|X|, \mathsf{len}_1, \ldots, \mathsf{len}_m) \text{ .}$$

Obliviousness under non-recurrent requests We say that a non-recurrent file hashing scheme satisfies obliviousness, iff Definition 3.1 is respected when the adversary $\mathcal{A}$ is constrained to submit request sequences $\mathbf{I}$ where all non dummy files $\mathsf{fid} \neq \perp$ appear at most once (i.e., no file is accessed twice).

Construction Below, we present our non-recurrent file hashing scheme. $\mathsf{Build}(X)$ has no output, but it stores a secret state[6] in the memory which consists of an array of length $2|X|$. Build arranges the memory in B bins of size Z each and places each file fid of length len in len consecutive bins starting from bin $\mathsf{PRF}_k(\mathsf{fid})$. In Read, the len bins are accessed and only the elements that are associated with fid are retrieved.

Algorithm 7.2. $\mathsf{H.Build}(X)$

1. Let $n := |X|$. The algorithm uses B bins of size Z each, where we let $Z := \alpha \log \lambda$ with $\alpha \in \omega(1)$, and let $B = \lceil 2n/Z \rceil$. Append to X exactly $ZB/2 - n$ elements.
2. Generate a random key k. In a single linear scan of X, mark every non-dummy element $(\mathsf{fid}, j, \mathsf{data}_j)$ with a targeted bin number $(\mathsf{PRF}_k(\mathsf{fid}) + j) \mod B$.
3. Copy X to an array Y of length ZB, and append ZB dummy elements to Y such that exactly Z dummy elements to each of the B bins.
4. Oblivious sort Y according to their bin assignment. Upon ties, prefer real elements over dummy elements, and break all other ties arbitrarily. At the end of this step, real elements that are assigned to the i-th bin appear before real elements that are assigned to the $(i+1)$-th bin. Between these real elements, there are exactly Z dummy elements.
5. Scan the array Y. For every bin $i \in [B]$, let l_i denote the number of real elements in that bin. If $l_i > Z$, then output $\mathsf{overflow}$ and abort. Otherwise, mark the next $Z - l_i$ dummy elements with bin i, and the remaining l_i dummy elements with exceed.
6. Oblivious sort the array Y again, where all the exceeded elements are moved to the very end. Truncate the array to be of length $2n$, and store it together with the PRF key K as the secret state.

Algorithm 7.3. $\mathsf{H.Read}(\mathsf{fid}, \mathsf{len})$

1. If $\mathsf{fid} \neq \perp$, compute the starting bin number $g := \mathsf{PRF}_k(\mathsf{fid})$. Else, choose g uniformly at random from $[0, \ldots, B-1]$.
2. Retrieve all blocks from bins $g, g+1, \ldots, g + \mathsf{len} - 1$ (each mod B).
3. In a single linear scan, extract from each bin the unique[7] block with file identifier fid. Write down exactly len blocks. Output those len elements.

We then obtain the following theorem:

[6] We assume that all information that the client stores at the external memory is encrypted and authenticated using some global secret key. The secret key used in the construction can be derived from the global secret key that the client holds by a proper labeling of the instance of the file hashing scheme, and therefore, no additional key should be stored.

[7] This assumes that there is no file with more than B blocks. In case this assumption does not hold, some bins will contain more than one blocks of the file. We will visit those bins more than once, extracting exactly one block in each scan of the bin.

Theorem 7.4. (Oblivious non-recurrent File hashing scheme) *Assuming one-way functions exist, for any super-constant function $\alpha := \omega(1)$, there exists a computationally secure non-recurrent file hashing scheme that requires $O(n)$ space for files of total length n, and except with $\mathsf{negl}(\lambda)$ probability*

- *Build: takes $O(n \log n \log^2(\alpha \log \lambda))$ work and has $(3, O(\log n \log^2(\alpha \log \lambda)))$ locality, and*
- *Read: each access of a file with length len costs $O(\mathsf{len} \cdot \alpha \log \lambda)$ work and $(1, O(1))$ locality.*

Recall that $\mathsf{Build}(X)$ leaks $|X| = n$, and each $\mathsf{Read}(\mathsf{fid}_i, \mathsf{len}_i)$ leaks len_i.

Proof. First, assume that $n > \log^2 \lambda$. We use Algorithm 7.3. We start with correctness of the scheme, then we show obliviousness and conclude the proof with analyzing efficiency and locality.

Correctness is immediate, where we just have to show that the event $\mathsf{overflow}$ in Build occurs with only negligible probability. If $\mathsf{overflow}$ does not occur, then clearly the scheme returns the same output as the functionality upon every Read operation. Note also that the functionality is deterministic, and thus, the outputs are, in fact, identical. □

Claim 7.5. (Overflow analysis) *According to the assignment used in Build, the probability of $\mathsf{overflow}$ within each bin is negligible in λ.*

Proof. Let $F_1, \ldots, F_k$ be the files that exist in the input array X, and let n_i denote the length of the file F_i. It holds that $\sum_{i=1}^{k} n_i \leq |X| = n \leq BZ/2$. For simplicity, assume that there is no file with length greater than B. For $\beta \in \{0, \ldots, B-1\}$ let X_β be a random variable denotes the load of the bin B_β, and for every $i \in [k]$ let $Y_\beta[i]$ be an indicator that gets 1 if and only if some element of the file F_i fells into bin K_β. Note that $X_\beta = \sum_{i=1}^{k} Y_\beta[i]$. Moreover, for a fixed $\beta \in \{0, \ldots, B-1\}$, $i \in [k]$ we have that $E[Y_\beta[i]] = n_i/B$. This holds since there is no file with length greater than B, and therefore, an element of some file is in the bin B_β if and only if its head was chosen to be in one of the previous n_i consecutive bins. This implies that

$$E[X_\beta] = E\left[\sum_{i=1}^{k} Y_\beta[i]\right] = \sum_{i=1}^{k} E\left[Y_\beta[i]\right] = \frac{\sum_{i=1}^{k} n_i}{B} \leq Z/2\ .$$

Moreover, the random variables $Y_\beta[1], \ldots, Y_\beta[k]$ are *independent* and are taking values in $\{0, 1\}$. By Chernoff's bound we have that the probability to exceed Z (and output $\mathsf{overflow}$) is negligible in λ, and the claim is obtained by a simple union bound on the number of bins B. The analysis can also be easily adapted for the case where we have a file with length $n_i > B$. In that case, each bin receives at least $\lfloor n_i/B \rfloor$ real elements of that file, and random $[n_i \mod B]$ consecutive bins receive one more element in addition, and therefore, this case is reduced to the case where all files are of length $< B$. □

Obliviousness We show that the memory addresses accessed can be simulated by a simulator Sim receiving only $\mathsf{leakage}$.

- Upon receiving a leakage $|X|$ for $\mathsf{Build}(X)$, the simulator performs few linear scans of the memory and invokes the simulator of the underlying oblivious sort twice, according to the Build algorithm.
- Upon receiving a leakage len for simulating some $\mathsf{H.Read}(\mathsf{fid}, \mathsf{len})$ instruction, the simulator chooses a random bin $g \in [B]$, and accesses $g, g+1, \ldots, g+\mathsf{len}-1$ (modulo B).

We claim that the memory locations produced by the simulator are indistinguishable from the memory locations in the real execution:

- Instruction $\mathsf{H.Build}(X)$ is clearly the same in both executions.
- Upon receiving instruction $\mathsf{H.Read}(\mathsf{fid}, \mathsf{len})$ with $\mathsf{fid}_i = \perp$, then Algorithm 7.3 chooses a random bin $g \in [B]$ and access the bins $g, g+1 \ldots, g+\mathsf{len} - 1$ (modulo B). This is exactly as the simulation.
- Instruction $\mathsf{H.Read}(\mathsf{fid}, \mathsf{len})$ with $\mathsf{fid} \neq \perp$, the non-recurrence property guarantees that there was no previous access to this fid. Algorithm 7.3 computes $g = \mathsf{PRF}_k(\mathsf{fid})$ and accesses bins $g, g+1 \ldots, g+\mathsf{len}-1$ (module B), whereas the simulator chooses g at random. Assuming the pseudorandom property of the PRF, and relying on the fact that fid was not queried before, these two distributions are the same and are also indistinguishable from the access pattern of $\mathsf{H.Read}(\perp, \mathsf{len})$.

Efficiency and locality The algorithm Build is just a constant number of invocations of oblivious sorts and a constant number of linear scans. Therefore, it can be implemented in time $O(n \log n \log^2(\alpha \cdot \log \lambda))$ and locality $(3, O(\log n \log^2(\alpha \log \lambda)))$ using Bucket oblivious sort (Theorem 7.1). As for Read, each invocation of the algorithm accesses a single region in the memory (maybe with some wraparound). Therefore, it can be implemented using a single head and $O(1)$ move operations.

For the case where $\log \lambda < n \leq \log^2 \lambda$, we will just use Bitonic sort as our sort. In that case, it is easy to see that Build takes $O(n \log^2 n)$ work and has $(2, O(\log^2 n))$ locality. Since $n < \log^2 \lambda$, this can be bounded by $O(n \log n \log^2(\alpha \log \lambda))$-work and $(3, \log n \log^2(\alpha \log \lambda))$-locality.

For the case of $n \leq \log \lambda$, our Build algorithm does nothing, and we just use a linear scan with each access.

This complete the proof of Theorem 7.4. □

7.3. *Constructing* $\mathsf{FileORAM}$ *from Non-Recurrent File Hashing Scheme*

In this section, we show how to construct a File ORAM with locality in linear space, in a blackbox manner from non-recurrent file hashing scheme. Below we use N to bound the total number of blocks in all files combined (i.e., the size of the logical memory).[8]

Data structure There are $\log N + 1$ *levels* numbered $0, \ldots, L$ where $L := \lceil \log_2 N \rceil$ is the largest level. Each level is a non-recurrent file hashing scheme and its data structure is denoted $\mathsf{H} := (\mathsf{H}_0, \ldots, \mathsf{H}_L)$ where H_i has capacity 2^i. At any time, each table H_i can be in two possible states, *ready* or *empty*.

[8]Recall that we use n to denote the size of an instance of the underlying building block, in our case, non-recurrent file hashing scheme, and N to denote the total size of the memory.

$\mathsf{FileORAM.Access}(\mathsf{op}, \mathsf{fid}, \mathsf{data})$. We first present the access algorithm assuming we know the length len of the requested file. We again assume $\mathsf{len} = 2^i$ for some non-negative integer i. After that we describe how to retrieve the length of the requested file using a metadata ORAM.

Algorithm 7.6. $\mathsf{H.Access}(\mathsf{op}, \mathsf{fid}, \mathsf{data}, \mathsf{len})$, where $\mathsf{len} = 2^i$ for some non-negative integer i:

```
1. $\mathsf{found} := \mathtt{false}$.
2. For each $\ell = i, \dots L$ in increasing order, if $\mathsf{H}_\ell$ is marked ready:
   (a) If not $\mathsf{found}$, then perform $\mathsf{fetched} := \mathsf{H}_\ell.\mathsf{Read}(\mathsf{fid}, \mathsf{len})$. If $\mathsf{fetched} \neq \perp$,
       let $\mathsf{found} := \mathtt{true}$, $\mathsf{data}^* := \mathsf{fetched}$.
   (b) Else $\mathsf{H}_\ell.\mathsf{Read}(\perp, \mathsf{len})$.
3. Let $D := \{(\mathsf{fid}, \mathsf{data}^*)\}$ if $\mathsf{op} = \mathtt{read}$; else let $D := \{(\mathsf{fid}, \mathsf{data})\}$.
4. If $\mathsf{H}_i$ is marked empty, let $\mathsf{H}_i := \mathsf{Build}(D)$ and mark it as ready. Else, perform the
   following rebuilding:
   (a) Let $\ell > i$ be the smallest level index greater than $i$ such that $\mathsf{H}_\ell$ is marked
       empty. If all levels $\ell > i$ are marked ready, then let $\ell := L$. In other words, $\ell$
       is the target level to be rebuilt.
   (b) Let $S := \mathsf{H}_0 \cup \dots \cup \mathsf{H}_{\ell-1}$ (if $\ell = L$, then additionally let $S := S \cup \mathsf{H}_L$). Further,
       tag each non-dummy element in $S$ with its level number, i.e., if a non-dummy
       element in $S$ comes from $\mathsf{H}_j$, tag it with the level number $j$. Thus, each element
       $j$ in $S$ is of the form $((\mathsf{fid}_j, l_j), \mathsf{data}_j)$.
   (c) Run $\mathsf{H}_\ell.\mathsf{Build}(\mathsf{Dedup}(S, 2^\ell))$, and mark $\mathsf{H}_\ell$ as ready. Further, set $\mathsf{H}_0 = \dots =$
       $\mathsf{H}_{\ell-1} := \emptyset$ and their status bits to empty.
   (d) Write back $\mathsf{H}_i.\mathsf{Build}(D)$, and mark it as ready.
5. Return $\mathsf{data}^*$.
```

To retrieve the length, we introduce metadata structure $\mathsf{H}_{\mathsf{meta}}$ that stores the length of each block. It is similar to H except that it always takes in $\mathsf{len} = 1$. Now each $\mathsf{FileORAM}$ access first retrieves the length of the requested file from $\mathsf{H}_{\mathsf{meta}}$, and then calls $\mathsf{H.Access}(\cdot)$.

Theorem 7.7. ($\mathit{FileORAM}$ in linear space with locality) *Let N be the total number of blocks in all files combined and assume that $N \in \mathsf{poly}(\lambda)$. Assuming one-way function exist, for any super-constant function $\alpha := \omega(1)$, there exists a computationally secure $\mathit{FileORAM}$ scheme that requires $O(N)$ space, and except with $\mathsf{negl}(\lambda)$ probability, requires $O(\mathsf{len} \cdot \log^2 N \cdot \log^2(\alpha \log \lambda))$ work and $(3, O(\log N \cdot \log^2(\alpha \log \lambda))$ locality.*

Proof. We start with efficiency analysis, and proceed to locality, space and obliviousness.

Amortized work for access Each access of a file of length len involves looking up in all levels $\log \mathsf{len}, \dots, \log N$, each costing $O(\mathsf{len} \cdot \alpha \log \lambda)$. Thus, the cost to retrieve a file of length len is $O(\log N \cdot \mathsf{len} \cdot (\alpha \log \lambda))$.

For rebuild, let isFull_i be the bit indicating whether the ith level is empty or ready. We can view the concatenations of bits $(\mathsf{isFull}_L, \ldots, \mathsf{isFull}_0)$ as a binary counter, denoted $\mathsf{counter}$. When accessing a file with length $\mathsf{len} = 2^i$, we perform the following rebuild:

- If level i is empty (i.e., $\mathsf{isFull}_i = 0$), we just rebuild level i (and set $\mathsf{isFull}_i = 1$). This is equivalent to incrementing the counter by 2^i.
- Otherwise, we look for the first $\ell > i$ for which $\mathsf{isFull}_\ell = 0$ (or set $\ell = L$ if all levels after i are full). We then rebuild level H_ℓ by taking all elements in all levels $\mathsf{H}_{\ell-1}, \ldots, \mathsf{H}_0$. After this operation, $\mathsf{isFull}_\ell = 1$ and $\mathsf{isFull}_{\ell-1}, \ldots, \mathsf{isFull}_0$. This is equivalent to increasing the counter by at most 2^i, as we ignore the values of $\mathsf{isFull}_{i-1}, \ldots, \mathsf{isFull}_0$. That is, when $\mathsf{isFull}_{i-1} = \ldots = \mathsf{isFull}_0 = 0$, we increase the counter by 2^i, whereas if $\mathsf{isFull}_{i-1} = \ldots = \mathsf{isFull}_0 = 1$ we just increment the counter by 1.

At this point, level i is empty, and we rebuild it with the data D. This is equivalent to incrementing the counter again by 2^i.

With each access of file of length $\mathsf{len} = 2^i$, the value of $\mathsf{counter}$ is increased by at most $2 \cdot 2^i$. Rebuilding a level of size 2^j costs $O(2^j \log 2^j (\log^2(\alpha \cdot \log \lambda)))$ total work. As a result, we can bound the amortized cost for rebuild when accessing a file of length len with $O(\mathsf{len} \cdot \log^2 N \cdot \log^2(\alpha \log \lambda))$.

Locality For accessing both the metadata and the main data, each access requires rebuild with a locality of $(3, O(\log N \log^2(\alpha \cdot \log \lambda))$. Each access requires looking up $\log(N/\mathsf{len})$ levels, each with a locality of $(1, O(1))$ resulting in a locality of $O(1, O(\log N))$. Thus, we achieve a locality of $(3, O(\log N \log^2(\alpha \cdot \log \lambda)))$ for our $\mathsf{FileORAM}$ construction.

Server storage The non-recurrent file hashing scheme construction uses a space of $4n$ for obliviously hashing a total of n blocks. We use $\log N$ instances of that construction, of exponentially increasing sizes in our $\mathsf{FileORAM}$ construction, requiring $\sum_{i=0}^{\log N}(4 \cdot 2^i) < 8 \cdot N = O(N)$ space.

Obliviousness We show the existence of an online simulator Sim that in each step receives some leakage $\mathsf{leakage}(I)$ for some instruction I and produces the memory address for that instruction.

Let $\mathsf{Sim}_0, \ldots, \mathsf{Sim}_L$ be the simulators for the non-recurrent file hashing schemes, $\mathsf{H}_0, \ldots, \mathsf{H}_L$, respectively, let $\mathsf{Sim}_{\mathsf{Dedup}}$ be the simulator for the deduplication algorithm, and let $\mathsf{Sim}_{\mathsf{meta}}$ be the simulator for the metadata ORAM.

The simulator Sim. The simulator for the $\mathsf{FileORAM}$ initializes $L+1$ bits representing whether the levels are ready or empty. Upon initialization, all the bits are empty, except for the level L which is marked as ready, and it also invokes the simulator Sim_L on Build instruction with input 2^L. The internal state of the simulator is all its internal bits, and the internal states of the underlying simulators.
Simulating $\mathsf{Access}(\mathsf{op}, \mathsf{fid}, \mathsf{data}, \mathsf{len})$, using only leakage $\mathsf{len} = 2^i$ where $\mathsf{len} = \mathsf{len}(\mathsf{fid})$:

1. Invoke the simulator $\mathsf{Sim}_{\mathsf{meta}}$ to simulate access to the metadata ORAM.
2. For $\ell = i, \ldots, L$, if H_ℓ is marked *ready*, invoke the simulator Sim_ℓ on leakage len.

3. If H_i is marked *empty*, then invoke Sim_i for simulating Build operation with input leakage len.
4. Otherwise, if H_i is marked *ready*, then let $\ell > i$ be the smallest level index greater than i such that H_ℓ is marked *empty*. If all levels are full, set $\ell := L$. Then, invoke $\mathsf{Sim}_{\mathsf{Dedup}}(2^\ell)$ and append the simulated instructions it produces to the output. Halt the simulators $\mathsf{Sim}_i, \ldots, \mathsf{Sim}_\ell$ and mark the bits corresponding to these levels as *empty*. Initialize Sim_ℓ on Build with leakage 2^ℓ and initialize Sim_i on Build with leakage 2^i. Mark these two levels as *ready*.

We now show that the view of the adversary, upon interactive with the simulator Sim and receiving outputs from the $\mathsf{FileORAM}$ functionality, is indistinguishable from the real construction (that interacts with the memory). We show that through a sequence of hybrid experiments, defined as follows:

- $\mathbf{Hyb_0}(\boldsymbol{\lambda})$: This is exactly the real execution. That is, upon receiving instruction $I_i = \mathsf{Access}(\mathsf{op}, \mathsf{fid}, \mathsf{data}, \mathsf{len})$ from the adversary, we invoke Algorithm 7.6 and output the memory addresses it produces, and the outputs out_i after completing the instruction I_i.
- $\mathbf{Hyb_1}(\boldsymbol{\lambda})$: This is as $\mathsf{Hyb}_0(\lambda)$ where the out_i is now being computed by the $\mathsf{FileORAM}$ functionality.
- $\mathbf{Hyb_{2,k}}(\boldsymbol{\lambda})$ with $k \in [L]$: Same as $\mathsf{Hyb}_1(\lambda)$, where we replace all non-recurrent file hashing scheme, $\mathsf{H}_0, \ldots, \mathsf{H}_{k-1}$ with simulators $\mathsf{Sim}_0, \ldots, \mathsf{Sim}_{k-1}$.
Upon receiving some instruction $\mathsf{Access}(\mathsf{op}, \mathsf{fid}, \mathsf{data}, \mathsf{len})$, we follow Algorithm 7.6. Whenever the Algorithm performs $\mathsf{H}_j.\mathsf{Build}(X)$ for some $j < k$ and some X, we replace it with an invocation of Sim_j for Build instruction with leakage $|X|$. Whenever the Algorithm performs $\mathsf{H}_j.\mathsf{Access}$ for $j < k$, we invoke Sim_j for Access with leakage len.
- $\mathbf{Hyb_3}(\boldsymbol{\lambda})$: Same as $\mathsf{Hyb}_{2,L}(\lambda)$ where the only difference is as follows: whenever calling to $\mathsf{Dedup}(S, 2^t)$ for some integer t and set S, we replace it with the $\mathsf{Sim}_{\mathsf{Dedup}}(2^t)$.
- $\mathbf{Hyb_4}(\boldsymbol{\lambda})$: Same as $\mathsf{Hyb}_3(\lambda)$ where every invocation of $\mathsf{H}_{\mathsf{meta}}$ is replaced with an access to $\mathsf{Sim}_{\mathsf{meta}}$. As such, the only information that we use in the $I = \mathsf{Access}(\mathsf{op}, \mathsf{fid}, \mathsf{data}, \mathsf{len})$ is the value len. This is exactly the simulation execution.

The view of the adversary in $\mathsf{Hyb}_4(\lambda)$ is indistinguishable from its view in $\mathsf{Hyb}_3(\lambda)$ due to the security of the metadata ORAM. Likewise, the view of the adversary in $\mathsf{Hyb}_3(\lambda)$ is indistinguishable from its view in $\mathsf{Hyb}_{2,\mathsf{L}}(\lambda)$ due to the security of the Dedup Algorithm.

For every $k \in \{0, \ldots, L-1\}$ it holds that $\mathsf{Hyb}_{2,k+1}(\lambda)$ is indistinguishable from $\mathsf{Hyb}_{2,\mathsf{k}}(\lambda)$ due to the security of the $(k+1)$th non-recurrent file hashing scheme. For that, we have to show that all accesses to H_{k+1} in $\mathsf{Hyb}_{2,\mathsf{k}}(\lambda)$ are non-recurrent, since this is a necessary condition to replace the construction with the simulator. When a file fid is first found in some level H_{k+1}, the corresponding file is entered into D. According to the definition of the ORAM algorithm, it is not hard to see until the next time T_k is rebuilt, fid exists in some H_ℓ where $\ell < k+1$. As a result, any instruction for looking for fid in level $k+1$, the bit found is guaranteed to be `true`, and in level H_{k+1} we will look for for the file id $\perp$. We conclude that until H_{k+1} is rebuilt, no lookup query will ever be issued again for fid to H_k. This holds for every file, and therefore, the instructions the accesses to H_{k+1} are non-recurrent, and can be replaced by the simulator.

It is easy to see that $\mathsf{Hyb}_{2,0}(\lambda)$ is identical to $\mathsf{Hyb}_1(\lambda)$. Finally, $\mathsf{Hyb}_1(\lambda)$ and $\mathsf{Hyb}_0(\lambda)$ are indistinguishable due to the correctness of the construction (the functionality is deterministic, and therefore, we can separately consider the access pattern and correctness). $\square$

8. Conclusions and Open Problems

We initiate a study of locality in oblivious RAM. For conclusion, we obtain the following results:

- There is an ORAM scheme that makes use of only 2 disks, that preserves the locality of the input program. Namely, if the input program accesses in total ℓ discontiguous regions, the ORAM scheme accesses at most $\ell \cdot \mathsf{poly}\log N$ discontiguous regions. Moreover, if the program accesses in total T logical addresses, then the ORAM accesses in total $T \cdot \mathsf{poly}\log N$ addresses. The ORAM leaks the sizes of the contiguous regions being accessed.
- We present asymptotic improvements (albeit, with statistical security) and a novel oblivious sort.
- Without leaking the sizes, we show a lower bound that the bandwidth of an oblivious program must be $\Omega(N)$, assuming $O(1)$-disks.

Open problems. We hope that our result will inspire future work on this topic. In the following, we provide several open questions on further understanding the trade-off between locality and bandwidth in oblivious compilation.

Preserving the number of disks. Our ORAM construction compiles $(1, \ell)$-local program into $(2, \mathsf{poly}\log N)$-local program that is oblivious. Is it possible to achieve a compiler that preserve the number of disks? We emphasize that our construction uses the second disk only in the oblivious sorting, and it unclear whether sorting with $(1, \ell \cdot \mathsf{poly}\log N)$-locality is possible to achieve.

Supporting more expressive input programs. Our motivated applications (e.g., outsourced file server, outsourced range query database), involve fetching some region from the memory and then accessing it in a streaming fashion. That is, we focused so far on supporting ORAM for $(1, \ell)$-local programs. A natural generalization is to construct an ORAM scheme that supports more expressive input programs, such as (D, ℓ)-local programs for $\mathsf{D} \geq 2$. This allows, for instance, computing inner products of D-arrays, or merging D-arrays. The input program sends to the memory instructions that also specify which disks to access, i.e., instructions of the form $(\mathsf{move}, \mathsf{d}, \mathsf{addr})$ and $(\mathsf{op}, \mathsf{d}, \mathsf{data})$, as defined in Sect. 3.

Such an ORAM scheme can be constructed quite easily using online range ORAM, and moreover, the ORAM even *preserves the number of disks*, i.e., it converts (D, ℓ)-local program to $(\mathsf{D}, \ell \cdot \mathsf{poly}\log N)$-local program. In a nutshell, the ORAM just holds D instances of online range ORAM on each disk. Observe that each range ORAM mainly uses one disk and the second disks only serves as a "workspace" for performing the oblivious sorts. We can reuse disks as the "workspaces" for other disks, e.g., if the online range ORAM of disk 1 wants to sort an array, it uses disk 2 as its workspace: It stores the current address in disk 2 and moves the head to a designated empty place. When completing the sort, it restores the address of disk 2. It is easy to see that this at most doubles the number of "jumps" in a simulation of an online range ORAM.

However, such a scheme not only reveals the lengths accessed in each disk, but also reveals the interleaving pattern of the accesses, i.e., each access also reveals which disk is being accessed. While this seems reasonable leakage for supporting, e.g., a program that computes inner product of two arrays, this leakage seems much more harmful in other computations, e.g., merging two sorted arrays. Understanding what computations are reasonable while leaking the interleaving pattern, or how to hide this pattern while preserving the number of disks, are intriguing open problems.

Locality-preserving OPRAM. We have considered a single CPU in this work. A natural question is whether we can extend the construction to support multiple CPUs, namely, to construct an oblivious parallel RAM (OPRAM) that preserves locality.

Asymptotic efficiency. We have showed the theoretic feasibility of constructing a Range ORAM with polylogarithmic work and locality. In this feasibility result, we favored conceptual simplicity over optimizing polylogarithmic factors. Nevertheless, it is interesting to see to what extent the constructions can be optimized. Perhaps locality-preserving ORAM can be constructed with the same bandwidth efficiency as a regular ORAM? Moreover, what is the space overhead of locality-preserving ORAM?

Acknowledgements

This work was partially supported by a Junior Fellow award from the Simons Foundation to Gilad Asharov. Currently supported by the Israel Science Foundation (grant No. 2439/20), by the BIU Center for Research in Applied Cryptography and Cyber Security in conjunction with the Israel National Cyber Bureau in the Prime Minister's Office, and by the European Union's Horizon 2020 research and innovation program under the Marie Skłodowska-Curie Grant Agreement No. 891234. Supported in part by NSF Award CNS-1561209, NSF Award CNS-1217821, NSF Award CNS-1704788, AFOSR Award FA9550-15-1-0262, a Microsoft Faculty Fellowship, a Google Faculty Research Award and a JP Morgan fellowship to Rafael Pass. This work was supported in part by NSF grants CNS-1314857, CNS-1514261, CNS-1544613, CNS-1601879, CNS-1617676, an Office of Naval Research Young Investigator Program Award, a Packard Fellowship, a Sloan Fellowship, Google Faculty Research Awards, a VMWare Research Award, and a Baidu Faculty Research Award to Elaine Shi. Kartik Nayak was partially supported by a Google Ph.D. Fellowship Award. T-H. Hubert Chan was partially supported by the Hong Kong RGC under the GGrants 17200418 and 17201220.

A. Appendix: Locality of Bitonic sort

In this section, we first analyze the locality of Bitonic sort, which runs in $O(n \log^2 n)$ time.

We call an array of numbers bitonic if it consists of two monotonic sequences, the first one ascending and the other descending, or vice versa. For an array S, we write it as $\widehat{S}$ if it is bitonic, as $\overrightarrow{S}$ (resp. $\overleftarrow{S}$) if it is sorted in an ascending (resp. descending) order.

The algorithm is based on a "bitonic split" procedure $\overrightarrow{\mathsf{Split}}$, which receives as input a bitonic sequence $\widehat{S}$ of length n and outputs a sorted sequence $\overrightarrow{S}$. $\overrightarrow{\mathsf{Split}}$ first separates $\widehat{S}$ into two bitonic sequences $\widehat{S}_1, \widehat{S}_2$, such that all the elements in S_1 are smaller than all the elements in S_2. It then calls $\overrightarrow{\mathsf{Split}}$ recursively on each sequence to get a sorted sequence. For simplicity, we assume that the length of the array is a power of 2.

Procedure A.1. $\overrightarrow{S} = \overrightarrow{\mathsf{Split}}(\widehat{S})$

- If $|\overrightarrow{S}| = 1$ then return the single element. Otherwise:
- Set $k = |\overrightarrow{S}|/2$.
- Let $\widehat{S}_1 = \langle \min(a_0, a_{k+0}), \min(a_1, a_{k+1}), \ldots, \min(a_{k-1}, a_{2k-1})\rangle$.
- Let $\widehat{S}_2 = \left\langle \max(a_0, a_{k+0}), \max(a_1, a_{k+1}), \ldots, \max(a_{n/2-1}, a_{2k-1})\right\rangle$.
- $\overrightarrow{S}_1 = \overrightarrow{\mathsf{Split}}(\widehat{S}_1)$, $\overrightarrow{S}_2 = \overrightarrow{\mathsf{Split}}(\widehat{S}_2)$ and $\overrightarrow{S} = (\overrightarrow{S}_1, \overrightarrow{S}_2)$.

Similarly, $\overleftarrow{S} = \overleftarrow{\mathsf{Split}}(\widehat{S})$ sorts the array in a descending order. We refer to [8] for details.

To sort an array S of n elements, the algorithm first converts S into a bitonic sequence using the Split procedures in a bottom up fashion, similar to the structure of merge–sort. Specifically, any size-2 sequence is a bitonic sequence. In each iteration $i = 1, \ldots, \log n - 1$, the algorithm merges each pair of size-2^i bitonic sequences into a size-2^{i+1} bitonic sequence. Toward this end, it uses the $\overrightarrow{\mathsf{Split}}$ and $\overleftarrow{\mathsf{Split}}$ alternately, as two sorted sequences $(\overrightarrow{S}_1, \overleftarrow{S}_2)$ form a bitonic sequence. The full Bitonic sort algorithm is presented below:

Algorithm A.2. BitonicSort(S)

```
1. Convert $S$ to a bitonic sequence: For $i = 1, \ldots, \log n - 1$:
    (a) Let $S = (\widehat{S}_0, \ldots, \widehat{S}_{n/2^i-1})$ be the size-$2^i$ bitonic sequences from the previous iteration.
    (b) For $j = 0, \ldots, n/2^{i+1} - 1$, $\widehat{B}_j = (\overrightarrow{\mathsf{Split}}(\widehat{S}_{2j}), \overleftarrow{\mathsf{Split}}(\widehat{S}_{2j+1}))$.
    (c) Set $S = (\widehat{B}_0, \ldots, \widehat{B}_{n/2^{i+1}-1})$.
2. The array $\widehat{S}$ is now a bitonic sequence. Apply $\overrightarrow{S} = \overrightarrow{\mathsf{Split}}(\widehat{S})$ to obtain a sorted sequence.
```

Locality and obliviousness It is easy to see that the sorting algorithm is oblivious, as all accesses to the memory are independent of the input data. For locality, first note that procedure $\overrightarrow{\mathsf{Split}}$ and $\overleftarrow{\mathsf{Split}}$ are $(2, O(\log n))$-local. No move operations are needed between instances of recursions, as these can be executed one after another as iterations (and using some vacuous reads). Thus, Algorithm A.2 is $(2, O(\log^2 n))$-local as it runs in $\log n$ iterations, each invoking $\overrightarrow{\mathsf{Split}}$ and $\overleftarrow{\mathsf{Split}}$. Figure 3 gives a graphic representation of the algorithm for input size 8 and Fig. 4 illustrates its locality. The $(2, O(\log^2 n))$ locality of Bitonic sort is also obvious from the figure.

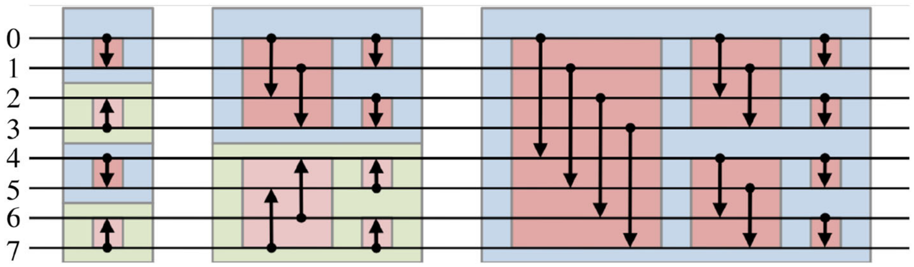


Fig. 3. Bitonic sorting network for 8 inputs. Input come in from the left end, and outputs are on the right end. When two numbers are joined by an arrow, they are compared, and if necessary are swapped such that the arrow points from the smaller number toward the larger number. This figure is modified from [44] .

Time $\longrightarrow$

	Pass 1							Pass 2						Pass 3						
Disk 1	0	1	2	3	4	5	6	0	1	2	3	4	5	0	1	2	3	4	5	6
Disk 2	1	2	3	4	5	6	7	2	3	4	5	6	7	1	2	3	4	5	6	7
Operation	$\downarrow$	$\perp$	$\uparrow$	$\perp$	$\downarrow$	$\perp$	$\uparrow$	$\downarrow$	$\downarrow$	$\perp$	$\perp$	$\uparrow$	$\uparrow$	$\downarrow$	$\perp$	$\downarrow$	$\perp$	$\uparrow$	$\perp$	$\uparrow$

Fig. 4. Locality of Bitonic sort for 8 elements. The figure shows the allocation of the data in the two disks for an 8 element array. For each input, either a compare–and–swap operation is performed in the specified direction or the input is ignored as denoted by $\perp$. The figure shows the first 3 passes out of the required 6 passes for 8 elements (see Fig. 3) .

Remark. Observe that in each pass of $\overrightarrow{\mathsf{Split}}$ (or $\overleftarrow{\mathsf{Split}}$), a min/max operation is a *read–compare–write* operation. Thus, strictly speaking, each memory location is accessed twice for this operation—once for reading and once for writing. When the write is performed, the read/write head has already moved forward and is thus not writing back to the same two locations that it read from. Going back to the same two locations would incur an undesirable move head operation. However, we can easily convert this into a solution that still preserves $(2, O(1))$-locality for each pass of $\overrightarrow{\mathsf{Split}}$ by introducing a *slack* after every memory location (and thus using twice the amount of storage). In this solution, every memory location a_i is followed by a'_i; the entire array is stored as $((a_0, a'_0), \ldots, (a_{n-1}, a'_{n-1}))$ where a_i stores real blocks and a'_i is a slack location. When a_i and a_j are compared, the results can be written to a'_i and a'_j, respectively, without incurring a move operation. Before starting the next iteration, we can move the data from slack locations to the actual locations in a single pass, thus preserving $(2, O(1))$-locality for each pass of $\overrightarrow{\mathsf{Split}}$ (and $\overleftarrow{\mathsf{Split}}$).

A.1. *Concurrent Executions of Bitonic Sorts*

In our constructions, we sometimes need to invoke Bitonic sorts on disjoint segments of equal size in an array. Let n be the array size and k be the segment size. If we naively sort each segment sequential, we would incur $(2, O((n/k) \cdot \log^2 k))$ locality. We can save the factor n/k by running each step of the Bitonic sort over all instances before starting the next step. Each step requires a scan on the segments, so after finishing one segment, the memory heads are right at the start of the next segment. It is not hard to see that this approach of "striped concurrent execution" achieves $(2, O(\log^2 k))$ locality.

Theorem A.3. (Perfectly secure concurrent oblivious sorts with locality) *Concurrent Bitonic sort can obliviously sort all disjoint size-k segments of a length-n array in* $O(n \cdot \log^2 k)$ *work and* $(2, O(\log^2 k))$ *locality.*

Specifically, the $k = n$ case is Theorem 2.2.

B. The Locality of Bucket Oblivious Sort [1]

In Sect. 2.3 we mentioned that Bitonic sort is locality-friendly. However, compared to AKS and Zig-zag sorts, Bitonic sort is asymptotically worse, although it performs much better in practice. To investigate the asymptotic overhead of Range ORAM, using Bitonic sort causes an additional logarithmic factor in bandwidth, and we are looking for an alternative oblivious sort that is locality-friendly and does not introduce the additional overhead.

In the following, we give an overview of Bucket Oblivious Sort of [1]. The sort is statistically secure, locality-friendly, and is $\mathsf{poly}\log\log$ factor away from the optimal oblivious sorts [4,28]. Let Z be a statistical security parameter that controls the error probability.

The basic idea. The core idea of the algorithm is to assign a random bin and then route the elements to the bins obliviously through a butterfly network. In a more detail, the n elements are divided into $B = 2n/Z$ buckets of size $Z/2$ each and $Z/2$ dummy elements are added to each bucket. Now, imagine that these B buckets form the inputs of a butterfly network—for simplicity, assume B is a power of two. Each element is uniformly randomly assigned to one of the B output buckets, represented by a key of $\log B$ bits. The elements are then routed through the butterfly network to their respective destinations. When the client can store two buckets locally at a time, at level i the client simply reads elements from two buckets that are of distance 2^i away in level i, and writes them to two adjacent buckets in level $i + 1$. The decision how to split the union of the two buckets from level i into two buckets in level $i + 1$ is done using the ith bit of each element's key. See Fig. 5 for a graphical illustration.

The above algorithm is clearly oblivious, as the order in which the client reads and writes the buckets is fixed and independent of the input array. If no bucket overflows, all elements reach their assigned destinations. By setting Z appropriately, the overflow probability can be bounded.

After assigning the elements into the bins, the dummy elements are removed (using a linear scan), each output bucket is permuted and all buckets are concatenated. Asharov et al. [1] showed that this implements an oblivious random permutation. Then, to get oblivious sort, they also show that any non-oblivious comparison based sorting algorithm can be applied on the output of the oblivious random permutation, and the resulting sorting algorithm is oblivious. They show:

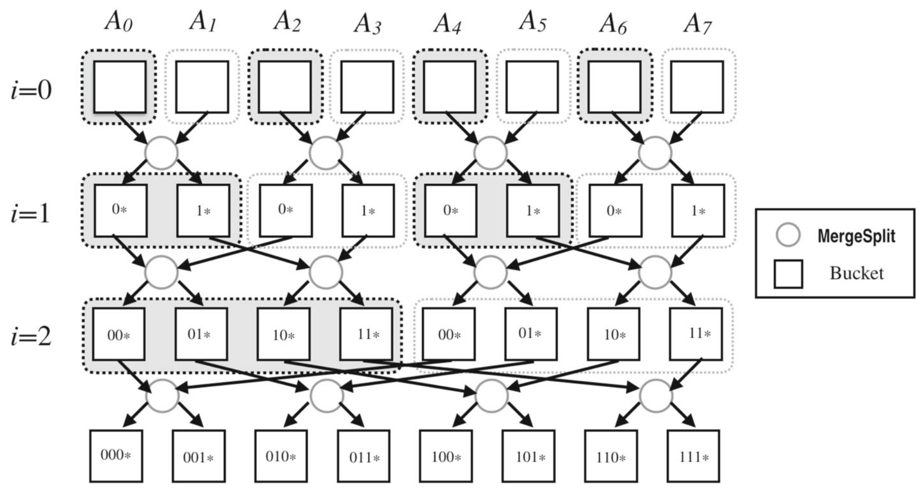


Fig. 5. Oblivious random bin assignment with 8 buckets. The MergeSplit procedure takes elements from two buckets at level i and put them into two buckets at level $i+1$, according to the $(i+1)$-th most significant bit of the keys. At level i, every 2^i consecutive buckets are semi-sorted by the most significant i bits of the keys .

Theorem B.1. [1] *There exists a statistically secure oblivious sort algorithm which, except with* $\approx e^{-Z/6}$ *probability, assuming that the client has local memory of size* $O(1)$, *then, the sort completes in* $O(n(\log n + Z)\log^2 Z)$ *bandwidth.*

The locality of Bucket oblivious sort. While it is mentioned in [1] that the algorithm can be implemented with good locality, there is no analysis. We now analyze the locality of this sort.

Each MergeSplit can be realized with a single invocation of Bitonic sort. Concretely, we first scan the two input buckets to count how many real elements should go to buckets A'_0 vs. A'_1, then tag the correct number of dummy elements going to either buckets, and finally perform a Bitonic sort for moving the elements obliviously. Next, we need to permute each output bucket obliviously with $O(1)$ local storage. This can be done as follows. First, assign each element in a bucket a uniformly random label of $c \cdot \log n$ bits for some constant $c > 3$. Then, obliviously sort the elements by their random labels using Bitonic sort. Since the labels are "short," i.e., the probability that two labels collide is n^{-c}. In case of a collision, we simply retry. The probability to have any collision is bounded by $\binom{n}{2}n^{-c} < 1/n$ for $c > 3$, and thus, in worst case, for $n > 3$ we will have by repeating the process Z times we obtain a failure probability that is smaller than $e^{-Z/6}$. Finally, run merge–sort. We have:

Theorem B.2. *The Bucket oblivious sort implements the sorting functionality except for* $\approx e^{-Z/6}$ *probability; It uses* $O(1)$ *client storage, consumes* $2n(\log n + Z)\log^2 Z$ *work, and* $(3, O((\log n + Z)\log^2 Z)$*-locality.*

Theorem 7.1 is obtained when $Z = \omega(1)\log\lambda$ for some super-constant function $\alpha := \omega(1)$ and for $n \geq \log^2\lambda$. In that case, the probability of failure is negligible, and the algorithm uses $O(n(\log n)\log^2(\alpha\log\lambda))$ work and $(3, O(\log n \log^2(\alpha \log\lambda)))$-locality.

Proof of Theorem B.2:. We show efficiency and locality.

Efficiency The algorithm has a linear scan of adding the dummy elements and tagging each element with a random bin. Then, we have $\log B$ iterations, where in each iteration we have $B/2$ instances of Bitonic sort on $2Z$ elements. This step can be bounded by $\log B \cdot B/2 \cdot 2Z \cdot \log^2(2Z)$ which is bounded by $2n\log n \log^2 Z$. Finally, we have also the permutation of each bucket. We have B buckets, and to permute them with error probability $e^{-Z/6}$ we have to run Z invocation of Bitonic sort, that is, this step is $BZ \cdot Z\log^2 Z$ which is $2nZ\log^2 Z$. The final step is an invocation of the merge–sort, which results in $2n\log n$ work. Overall, the work of the bucket sort algorithm is $2n(\log n + Z)\log^2 Z$ work.

Locality The algorithm consists of few linear scans, and concurrent executions of Bitonic sort (i.e., we run Bitonic sort on pair of buckets concurrently). In "Appendix A.1," we show that concurrent execution of Bitonic sort can sort disjoint size Z segments of a length n array in $O(n\log^2 Z)$ work and $(2, O(\log^2 Z))$ locality. We have $\log B$ levels in the algorithm, which results in $O(n\log n\log^2 Z)$ work and $(2, O(\log n\log^2 Z))$ locality. Permuting the bins is takes $O(\log^2 Z)$ locality for one trial, and we have to repeat it Z times which takes in total $O(Z\log^2 Z)$ locality. Finally, we invoke the merge–sort. It is easy to see that merging two sorted arrays can be implemented in $O(3, O(1))$ locality: We linearly scan that the input arrays and write into the output array. In merge–sort, we have several instances of pairs of arrays that have to be merged at the same level. This is implemented in a similar manner to concurrent executions of Bitonic sort. We conclude that the merge–sort requires $O(n\log n)$ time and $(3, O(\log n))$-locality. Overall, the Bucket Oblivious Sort algorithm requires $2n(\log n + Z)\log^2 Z$ work and $(3, O((\log n + Z)\log^2 Z)$-locality. □

References

[1] G. Asharov, T.-H.H. Chan, K. Nayak, R. Pass, L. Ren, E. Shi, Bucket oblivious sort: An extremely simple oblivious sort, in *3rd Symposium on Simplicity in Algorithms, SOSA@SODA 2020* (SIAM, 2020), pp. 8–14

[2] L. Arge, P. Ferragina, R. Grossi, J.S. Vitter, On sorting strings in external memory (extended abstract), in *ACM Symposium on the Theory of Computing (STOC '97)* (1997), pp. 540–548

[3] G. Asharov, I. Komargodski, W.-K. Lin, K. Nayak, E. Peserico, E. Shi, Optorama: Optimal oblivious RAM, in *Advances in Cryptology—EUROCRYPT 2020, Proceedings, Part II, volume 12106 of Lecture Notes in Computer Science* (Springer, 2020), pp. 403–432

[4] M. Ajtai, J. Komlós, E. Szemerédi, An $O(N\log N)$ sorting network, in *ACM Symposium on Theory of Computing (STOC '83)* (1983), pp. 1–9

[5] D. Apon, J. Katz, E. Shi, A. Thiruvengadam, Verifiable oblivious storage, in *Public Key Cryptography (PKC'14)* (2014), pp. 131–148

[6] G. Asharov, M. Naor, G. Segev, I. Shahaf, Searchable symmetric encryption: optimal locality in linear space via two-dimensional balanced allocations, in *ACM Symposium on Theory of Computing (STOC '16)* (2016), pp. 1101–1114

[7] G. Asharov, G. Segev, I. Shahaf, Tight tradeoffs in searchable symmetric encryption, in *CRYPTO (1)*, vol. 10991 (2018), pp. 407–436

[8] K.E. Batcher, Sorting Networks and Their Applications. AFIPS ’68 (1968)

[9] E. Boyle, M. Naor, Is there an oblivious RAM lower bound?, in *ACM Conference on Innovations in Theoretical Computer Science (ITCS ’16)* (2016), pp. 357–368

[10] E. Brewer, L. Ying, L. Greenfield, R. Cypher, T. T’so, Disks for data centers—white paper for FAST 2016. Technical report, Google (2016)

[11] A. Chakraborti, A.J. Aviv, S.G. Choi, T. Mayberry, D.S. Roche, R. Sion, rORAM: Efficient Range ORAM with $O(\log^2 N)$ Locality, in *Network and Distributed System Security (NDSS)* (2019)

[12] R. Canetti, Security and composition of multiparty cryptographic protocols. *J. Cryptology*, **13**(1), 143–202 (2000)

[13] T.H.H. Chan, K.-M. Chung, B. Maggs, E. Shi, Foundations of differentially oblivious algorithms, in *Symposium on Discrete Algorithms (SODA)* (2019)

[14] R. Curtmola, J.A. Garay, S. Kamara, R. Ostrovsky, Searchable symmetric encryption: improved definitions and efficient constructions, in *ACM Conference on Computer and Communications Security (CCS ’06)* (2006), pp. 79–88

[15] D. Cash, S. Jarecki, C.S. Jutla, H. Krawczyk, M.-C. Rosu, M. Steiner, Highly-scalable searchable symmetric encryption with support for boolean queries, in *Advances in Cryptology—CRYPTO 2013. Proceedings, Part I* (2013), pp. 353–373

[16] M. Chase, S. Kamara, Structured encryption and controlled disclosure, in *Asiacrypt* (Springer, 2010), pp. 577–594

[17] K.-M. Chung, Z. Liu, R. Pass, Statistically-secure ORAM with $\tilde{O}(\log^2 n)$ overhead, in *Asiacrypt* (2014)

[18] T.-H.H. Chan, K. Nayak, E. Shi, Perfectly secure oblivious parallel RAM, in *Theory of Cryptography Conference (TCC)* (2018)

[19] T.-H.H. Chan, E. Shi, Circuit OPRAM: unifying statistically and computationally secure orams and oprams, in *Theory of Cryptography—15th International Conference, TCC 2017, volume 10678 of Lecture Notes in Computer Science* (Springer, 2017), pp. 72–107

[20] D. Cash, S. Tessaro, The locality of searchable symmetric encryption, in *Advances in Cryptology—EUROCRYPT 2014*, vol. 8441 (2014), pp. 351–368

[21] I. Demertzis, C. Papamanthou, Fast searchable encryption with tunable locality, in *SIGMOD Conference* (ACM, 2017), pp. 1053–1067

[22] I. Demertzis, D. Papadopoulos, C. Papamanthou, Searchable encryption with optimal locality: Achieving sublogarithmic read efficiency, in *CRYPTO* (2018)

[23] S. Devadas, M. van Dijk, C.W. Fletcher, L. Ren, E. Shi, D. Wichs, Onion ORAM: a constant bandwidth blowup oblivious RAM, in *TCC* (2016)

[24] M.T. Goodrich, M. Mitzenmacher, Privacy-preserving access of outsourced data via oblivious RAM simulation, in *ICALP* (2011)

[25] O. Goldreich, R. Ostrovsky, Software protection and simulation on oblivious RAMs. *J. ACM* (1996)

[26] O. Goldreich, Towards a theory of software protection and simulation by oblivious RAMs, in *STOC* (1987)

[27] O. Goldreich, *The Foundations of Cryptography—Volume 2, Basic Applications* (Cambridge University Press, 2004)

[28] M.T. Goodrich, Zig-zag sort: a simple deterministic data-oblivious sorting algorithm running in o(n log n) time, in *STOC* (2014)

[29] G. Kellaris, G. Kollios, K. Nissim, A. O’Neill, Generic attacks on secure outsourced databases, in *ACM CCS* (2016), pp. 1329–1340

[30] G. Kellaris, G. Kollios, K. Nissim, A. O’Neill, Accessing data while preserving privacy. *CoRR*, arXiv:abs/1706.01552 (2017)

[31] E. Kushilevitz, S. Lu, R. Ostrovsky, On the (in)security of hash-based oblivious RAM and a new balancing scheme, in *SODA* (2012)

[32] K. Kurosawa, Y. Ohtaki, How to update documents verifiably in searchable symmetric encryption, in *International Conference on Cryptology and Network Security* (Springer, 2013), pp. 309–328

[33] S. Kamara, C. Papamanthou, Parallel and dynamic searchable symmetric encryption, in *Financial Cryptography and Data Security* (2013), pp. 258–274

[34] K.G. Larsen, J.B. Nielsen, Yes, there is an oblivious RAM lower bound!, in *CRYPTO* (2018), pp. 523–542

[35] S. Patel, G. Persiano, M. Raykova, K. Yeo, Panorama: Oblivious RAM with logarithmic overhead, in *FOCS* (2018)
[36] C. Ruemmler, J. Wilkes, An introduction to disk drive modeling, *IEEE Computer*, **27**(3), 17–28 (1994)
[37] E. Shi, T.-H.H. Chan, E. Stefanov, M. Li, Oblivious RAM with $O((\log N)^3)$ worst-case cost, in *ASIACRYPT* (2011)
[38] E. Stefanov, M. van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, S. Devadas, Path ORAM—an extremely simple oblivious ram protocol, in *CCS* (2013)
[39] J.S. Vitter, External memory algorithms and data structures. *ACM Comput. Surv.* **33**(2), 209–271 (2001)
[40] J.S. Vitter, Algorithms and data structures for external memory. *Foundations and Trends in Theoretical Computer Science* **2**(4), 305–474 (2006)
[41] P. Van Liesdonk, S. Sedghi, J. Doumen, P. Hartel, W. Jonker, Computationally efficient searchable symmetric encryption, in *Workshop on Secure Data Management* (Springer, 2010), pp. 87–100
[42] X. Wang, T.-H.H. Chan, E. Shi, Circuit ORAM: On Tightness of the Goldreich-Ostrovsky Lower Bound, in *ACM Conference on Computer and Communications Security* (ACM, 2015), pp. 850–861
[43] X.S. Wang, Y. Huang, T.-H.H. Chan, A. Shelat, E. Shi, SCORAM: Oblivious RAM for Secure Computation, in *CCS* (2014)
[44] Wikipedia. Bitonic sorter. https://en.wikipedia.org/wiki/Bitonic_sorter#/media/File:BitonicSort1.svg. Online; accessed (August 2021).
[45] P. Williams, R. Sion, Usable PIR, in *Network and Distributed System Security Symposium (NDSS)* (2008)
[46] P. Williams, R. Sion, Round-optimal access privacy on outsourced storage, in *ACM Conference on Computer and Communication Security (CCS)* (2012)
[47] P. Williams, R. Sion, B. Carbunar, Building castles out of mud: practical access pattern privacy and correctness on untrusted storage, in *CCS* (2008), pp. 139–148