Locality-Preserving Oblivious RAM

Gilad Asharov[1⋆], T-H. Hubert Chan[2], Kartik Nayak[3⋆⋆], Rafael Pass[1], Ling Ren[4⋆⋆], and Elaine Shi[1]

[1] Cornell/Cornell Tech
[2] The University of Hong Kong
[3] University of Maryland
[4] MIT

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 poly-logarithmic overhead both in terms of bandwidth and locality. We also study the tradeoff 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 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 [23, 25, 36], originally proposed in the seminal work by Goldreich and Ostrovsky [23, 25], 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, 34, 36, 38, 42]) it is now understood that ORAM schemes can be constructed incurring only *logarithmic* bandwidth [6]; and moreover, this is asymptotically optimal [25, 33].

⋆ Currently a researcher at JP Morgan AI Research.
⋆⋆ Currently at VMware Research

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., [6,18,23,25,36,38,42]) 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 [16,30], 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.*

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 [12] 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., [7, 8, 13, 20, 21]). 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)[5]; 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.

2 Technical Roadmap

In the following we provide a summary of results and techniques. In Section 2.1 we discuss our modeling of locality. In Section 2.2 we discuss our lower bounds,

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

providing tradeoffs between the locality of a program, leakage and bandwidth. Towards introducing our construction, we start in Section 2.3 with a warmup– oblivious sort with "good" locality. In Section 2.4 we introduce range ORAM, our core building block for achieving locality, in which in Section 2.5 we overview its construction. In Section 2.6 we overview a variant of Range ORAM, called "Online Range ORAM", which can also be viewed as a locality preserving ORAM.

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 [7,8,13,20]. 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 [7,8,13,20] 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 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 must fetch from the disk upon each request. This model was adopted by the SSE line of work, however, 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 provide servers with multiple disks. Constraining to such a single-disk model might rule out interesting cryptographic algorithms of practical value. Therefore, 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 eack 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 movements operations in these disks. We refer the readers to Section 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 vary 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 Section 7 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 tradeoffs between leakage profiles and performance. We consider schemes that leak only the total number of accesses (just as in standard ORAM[6]) 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 ORAM 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 Section 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

[6] 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.

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 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 [2] and Zig-zag sort [26] 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 [9] 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.*

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 $\mathsf{op} = \mathtt{read}$, 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 $\mathsf{op} = \mathtt{write}$, 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.*

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 super-blocks 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 super-block 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 super-blocks of that

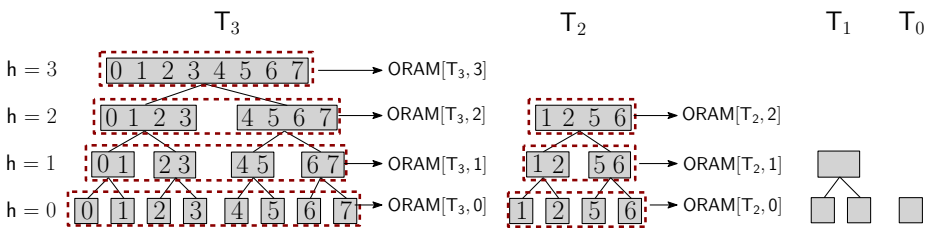

Fig. 1: Hierarchy of range trees. Logically, data is divided into trees of exponentially increasing sizes. In each tree block, a parent super-block stores the contents of both its children. If a block appears in more than one tree, the smallest tree contains the freshest copy. The above figure shows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

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 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 Figure 1, each of $\mathsf{T}_0, \mathsf{T}_1, \mathsf{T}_2$ 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 super-block of size 2^j (leaves have height 0 and store primitive blocks).

Range ORAM's data structure. As shown in Figure 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 [23, 25], 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 Figure 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 super-blocks

(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 super-blocks at the same height will be stored in a separate ORAM — thus an ORAM at height j of the tree stores super-blocks 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 super-blocks 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 super-blocks that intersect the range requested — it is not difficult to see that there are at most *two* such super-blocks.
- For every Range Tree smaller than 2^i in size, we simply fetch the root.
- Write down all these super-blocks 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* super-blocks to request in the desired height that stores super-blocks of size 2^i;
2. Fetch these two desired super-blocks from the corresponding ORAM and write down the fetched super-block in a contiguous region (starting at a designated position) on the server's memory.

All these fetched super-blocks 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 super-blocks of size 2^i. Using an ordinary ORAM scheme, this step requires reading polylogarithmically many regions of size 2^i. Here, since every super-block 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 Section 2.3.

Maintain phase of the Range ORAM. Inspired by the hierarchial ORAM [23, 25], here a super-block fetched will be written to the smallest Range Tree that is large enough to fit this super-block. 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 Section 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 Section 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 Section 5 for the algorithmic details.

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 $\mathsf{op} \in \{\texttt{read}, \texttt{write}\}$, 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; and 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 Section 5.5 for further details.

2.7 Related Work

Related work on locality. Algorithmic performance with data stored on the disk has been studied in the external memory models (e.g., [4,35,40,41] 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., [17, 19, 28, 31, 39], and references within) is to store an inverted index. Our work is inspired by recent works that study locality in SSE schemes [7, 8, 13, 20, 21]. Our new locality ORAM formulation can be viewed as a generalization of the one-time ORAM (with free rebuild) construct adopted in recent SSE constructions.

In a concurrent work, Demertzis, Papadopoulos and Papamanthou [20] also consider such a one-time ORAM (with free rebuild) abstraction for an SSE application. In their construction, they leverage as a building block a perfectly secure (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 $O(\mathsf{len} \cdot \sqrt{N})$-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 [6, 27, 32, 34, 36, 38, 42–46] construct ORAMs in different settings. Most of ORAM constructions follow one of two frameworks: the hierarchical framework, originally proposed by Goldreich and Ostrovsky [23, 25], or the tree-based framework proposed by Shi et al. [36].

Up until recently, the asymptotically most efficient scheme was given by [32], providing $O(\log^2 N/\log\log N)$ bandwidth. A recent improvement was given by Patel et al. [34], reducing the bandwidth to $O(\log N \cdot \mathsf{poly}\log\log N)$. The scheme of Asharov et al. [6] achieves $O(\log N)$ bandwidth, and matches the lower bounds given by Goldreich and Ostrovsky [23,25] and Larsen and Nielsen [33]. Further, the Goldreich-Ostrovsky lower bound is also known not to hold when the memory (i.e., ORAM server) is capable of performing computation [3,22], which is beyond the scope of this paper.

In a subsequent work, Chakraborti et al. [14] 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.

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 [23,25,27,32,36,38,42].

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 $\mathsf{op} \in \{\mathtt{read}, \mathtt{write}\}$, $\mathsf{d} \in [\mathsf{D}]$ and $\mathsf{data} \in \{0,1\}^b \cup \{\perp\}$. If $\mathsf{op} = \mathtt{read}$, then $\mathsf{data} = \perp$ and mem should return the content of the block pointed to by the d-th disk; If $\mathsf{op} = \mathtt{write}$, 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, deteministic 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, $\mathsf{Expt}^{\mathrm{real},M}_{\mathcal{A}}$ and $\mathsf{Expt}^{\mathrm{ideal},f}_{\mathcal{A},\mathsf{Sim}}$ are computationally indistinguishable.*

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

$\underline{\mathsf{Expt}^{\mathrm{ideal},f}_{\mathcal{A},\mathsf{Sim}}(1^\lambda)}$:
$\mathsf{out}_0 = \mathsf{addr}_0 = \perp$
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 [11, 24].

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 use ordinary ORAM as an example to show how to use our definitions.

Ordinary ORAM. As an example, a conventional ORAM, first proposed by Goldreich and Ostrovsky [23], 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 $\mathsf{op} \in \{\texttt{read}, \texttt{write}\}$, $\mathsf{addr} \in [N]$, and $\mathsf{data} \in \{0,1\}^b \cup \{\perp\}$, the functionality proceeds as follows. If $\mathsf{op} = \texttt{write}$, 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.

We remark that previous constructions of ORAM [32, 38, 42] in fact satisfy Definition 3.1.

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 mem-

ory, 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 $(\mathsf{move}, \mathsf{d}, \mathsf{addr})$ and $(\mathsf{op}, \mathsf{d}, \mathsf{data})$ that are sent to the memory, data should be encrypted. Note, however, that the adversary sees in particular the 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.

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.*

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 towards 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 is 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 [23] 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 [23] 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 poly-logarithmic bandwidth and poly-logarithmic locality. The construction uses a building block which we call an oblivious range tree (Section 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 (Section 5.3). Then, we discuss statistical and perfect security in Section 5.4. Finally, we extend Range ORAM to online Range ORAM (Section 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 is 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 $\mathsf{op} \in \{\texttt{read}, \texttt{write}\}$, $s, t \in [N]$, $s < t$, and $\mathsf{data} \in (\{0,1\}^b)^{(t-s+1)}$. If $\mathsf{op} = \texttt{read}$, then it returns $\mathsf{mem}[s, \ldots, t]$. If $\mathsf{op} = \texttt{write}$, 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 construction Range ORAM 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:

- $\mathsf{T}.\mathsf{Build}(X)$ 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.
- $\mathsf{B} \leftarrow \mathsf{T}.\mathsf{Access}(\texttt{read}, [s,t], \perp)$ 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.

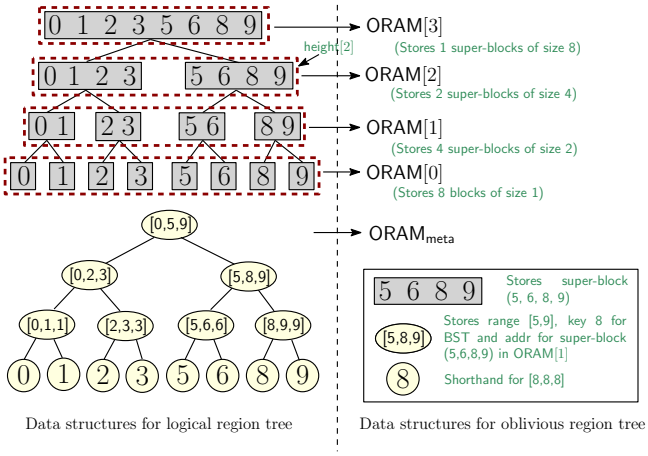

Fig. 2: An oblivious Range Tree with Locality.

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

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 *super-block*, i.e., blocks from all leaves in its subtree concatenated and ordered by addresses. A height-i super-block thus has size 2^i. The leaves are at height 0, and the root is at height $\log_2 n$.

Metadata tree. Each super-block 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 super-block). We use another full binary tree to store the range metadata of each super-block, henceforth referred to as the *metadata tree*. The metadata tree is a natural 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 $\perp$ 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 Figure 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 (Section 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 super-blocks.** For each height $i = 1, 2, \ldots, L := \log_2 n$, create height-$i$ super-blocks by concatenating their two child nodes. Let $\mathsf{height}[i]$ denote the set of height-i super-blocks. Tag each super-block 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 Section 4.3, and call $\mathsf{H}_i.\mathsf{Build}(\mathsf{height}[i])$ in which each height-i super-block behaves as an atomic block. Let H_{meta} be a locally initializable ORAM, and call $\mathsf{H}_{\text{meta}}.\mathsf{Build}(\mathsf{metadata})$.

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

1. **Look up address.** Call $\mathsf{H}_{\text{meta}}.\mathsf{Access}(\cdot)$ $2L$ times to obliviously search 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 super-blocks.** 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 $\mathsf{B} = \mathsf{Dedup}(\mathsf{B}_1 \,||\, \mathsf{B}_2, \mathsf{len})$.

We prove the following theorem in the full version.

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*

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

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 [23, 25]. 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. 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 Section 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. We show a simple example for how levels are updated after some accessed. We assume initially that all blocks are stored in the largest Range Tree. Consider the following sequence of ranges $[1,1], [2,3], [4,5], [6,6]$.

- 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.
- 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.

The following theorem is proven in the full version of the paper.

Theorem 5.5 (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.*

We remark that the both bandwidth and locality are in an amortized sense: for sufficiently large amount of accesses of contiguous addresses $\mathsf{len}_1, \ldots, \mathsf{len}_m$, the total bandwidth is $(\sum_{i=1}^{m} \mathsf{len}_i) \cdot \mathsf{poly} \log N$ and locality is $(2, m \cdot \mathsf{poly} \log N)$.

5.4 Perfectly Security 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. [15] 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. [15] 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 5.6 (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 $\mathsf{op} \in \{\texttt{read}, \texttt{write}\}$, $\mathsf{addr} \in [N]$ and $\mathsf{data} \in \{0,1\}^b \cup \{\perp\}$. If $\mathsf{op} = \texttt{write}$, 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. Initially, let $\mathsf{rsize} := 1$, $p = 1$, and let $\mathsf{prefetch} := \perp$. Upon receiving a memory request:

- If $\mathsf{prefetch}[p]$ does not match the logical address requested, then do the following.
  1. First, write back the entire $\mathsf{prefetch}$ back into the range ORAM.
  2. Next, request a region of length 1 consisting of only the requested logical address, store the result in $\mathsf{prefetch}$;
  3. Reset $p := 1$ and $\mathsf{rsize} := 1$;
- Read and write $\mathsf{prefetch}[p]$, and let $p := p + 1$.
- If $p > \mathsf{rsize}$, then do the following.
  1. First, let $\mathsf{rsize} := 2 \cdot \mathsf{rsize}$.
  2. Next, write $\mathsf{prefetch}$ back into the range ORAM.
  3. Now, prefetch the next contiguous region containing rsize logical addresses, and store them in $\mathsf{prefetch}$, and let $p := 1$.

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. Thus we have the following theorem.

Theorem 5.7 (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.*

6 Lower Bound for More Restricted Leakage

In Section 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 $\mathsf{op} \in \{\mathtt{read}, \mathtt{write}\}$, $\mathsf{addr} \in [N]$ and $\mathsf{data} \in \{0,1\}^b \cup \{\perp\}$.
2. Just like earlier ORAM lower bounds [10, 23, 25]), 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[7], 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.

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

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 $\sum_{j\in[\mathsf{D}]} a_j = N - c$.

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* $\Omega(\frac{N}{\mathsf{D}})$.

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 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.

– 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 Section 3. As we discuss further in the appendices of the online full version [5], depending on how we formulate the allowable leakage, the problem can be easy or an open challenge.

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 poly-logarithmic work and locality. In this feasibility result, we favored conceptual simplicity over optimizing poly-logarithmic 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?

Acknowledgments

This work was partially supported by a Junior Fellow award from the Simons Foundation to Gilad Asharov. This work was supported in part by NSF grants CNS-1314857, CNS-1514261, CNS-1544613, CNS-1561209, 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 grant 17200418.

References

1. Bitonic sorter. https://en.wikipedia.org/wiki/Bitonic_sorter, online; accessed October 2018
2. Ajtai, M., Komlós, J., Szemerédi, E.: An $O(N \log N)$ sorting network. In: ACM Symposium on Theory of Computing (STOC '83). pp. 1–9 (1983)
3. Apon, D., Katz, J., Shi, E., Thiruvengadam, A.: Verifiable oblivious storage. In: Public Key Cryptography (PKC'14). pp. 131–148 (2014)
4. Arge, L., Ferragina, P., Grossi, R., Vitter, J.S.: On sorting strings in external memory (extended abstract). In: ACM Symposium on the Theory of Computing (STOC '97). pp. 540–548 (1997)
5. Asharov, G., Chan, T.H.H., Nayak, K., Pass, R., Ren, L., Shi, E.: Locality-preserving oblivious ram. Online full version of this paper, https://eprint.iacr.org/2017/772
6. Asharov, G., Komargodski, I., Lin, W.K., Nayak, K., Peserico, E., Shi, E.: OptORAMa: Optimal oblivious RAM. Cryptology ePrint Archive, Report 2018/892
7. Asharov, G., Naor, M., Segev, G., Shahaf, I.: Searchable symmetric encryption: optimal locality in linear space via two-dimensional balanced allocations. In: ACM Symposium on Theory of Computing (STOC '16). pp. 1101–1114 (2016)
8. Asharov, G., Segev, G., Shahaf, I.: Tight tradeoffs in searchable symmetric encryption. In: CRYPTO (1). vol. 10991, pp. 407–436 (2018)
9. Batcher, K.E.: Sorting Networks and Their Applications. AFIPS '68 (1968)
10. Boyle, E., Naor, M.: Is there an oblivious RAM lower bound? In: ACM Conference on Innovations in Theoretical Computer Science (ITCS '16). pp. 357–368 (2016)
11. Canetti, R.: Security and composition of multiparty cryptographic protocols. J. Cryptology **13**(1), 143–202 (2000)
12. Cash, D., Jarecki, S., Jutla, C.S., Krawczyk, H., Rosu, M., Steiner, M.: Highly-scalable searchable symmetric encryption with support for boolean queries. In: Advances in Cryptology - CRYPTO 2013. Proceedings, Part I. pp. 353–373 (2013)
13. Cash, D., Tessaro, S.: The locality of searchable symmetric encryption. In: Advances in Cryptology - EUROCRYPT 2014. vol. 8441, pp. 351–368 (2014)
14. Chakraborti, A., Aviv, A.J., Choi, S.G., Mayberry, T., Roche, D.S., Sion, R.: rORAM: Efficient Range ORAM with $O(\log^2 N)$ Locality. In: Network and Distributed System Security (NDSS) (2019)
15. Chan, T.H., Nayak, K., Shi, E.: Perfectly secure oblivious parallel RAM. In: Theory of Cryptography Conference (TCC) (2018)
16. Chan, T.H., Chung, K.M., Maggs, B., Shi, E.: Foundations of differentially oblivious algorithms. In: Symposium on Discrete Algorithms (SODA) (2019)
17. Chase, M., Kamara, S.: Structured encryption and controlled disclosure. In: Asiacrypt. pp. 577–594. Springer (2010)
18. Chung, K.M., Liu, Z., Pass, R.: Statistically-secure ORAM with $\tilde{O}(\log^2 n)$ overhead. In: Asiacrypt (2014)
19. Curtmola, R., Garay, J.A., Kamara, S., Ostrovsky, R.: Searchable symmetric encryption: improved definitions and efficient constructions. In: ACM Conference on Computer and Communications Security (CCS '06). pp. 79–88 (2006)
20. Demertzis, I., Papadopoulos, D., Papamanthou, C.: Searchable encryption with optimal locality: Achieving sublogarithmic read efficiency. In: CRYPTO (2018)
21. Demertzis, I., Papamanthou, C.: Fast searchable encryption with tunable locality. In: SIGMOD Conference. pp. 1053–1067. ACM (2017)
22. Devadas, S., van Dijk, M., Fletcher, C.W., Ren, L., Shi, E., Wichs, D.: Onion ORAM: a constant bandwidth blowup oblivious RAM. In: TCC (2016)

23. Goldreich, O.: Towards a theory of software protection and simulation by oblivious RAMs. In: STOC (1987)
24. Goldreich, O.: The Foundations of Cryptography - Volume 2, Basic Applications. Cambridge University Press (2004)
25. Goldreich, O., Ostrovsky, R.: Software protection and simulation on oblivious RAMs. J. ACM (1996)
26. Goodrich, M.T.: Zig-zag sort: a simple deterministic data-oblivious sorting algorithm running in o(n log n) time. In: STOC (2014)
27. Goodrich, M.T., Mitzenmacher, M.: Privacy-preserving access of outsourced data via oblivious RAM simulation. In: ICALP (2011)
28. Kamara, S., Papamanthou, C.: Parallel and dynamic searchable symmetric encryption. In: Financial Cryptography and Data Security. pp. 258–274 (2013)
29. Kellaris, G., Kollios, G., Nissim, K., O'Neill, A.: Generic attacks on secure outsourced databases. In: ACM CCS. pp. 1329–1340 (2016)
30. Kellaris, G., Kollios, G., Nissim, K., O'Neill, A.: Accessing data while preserving privacy. CoRR **abs/1706.01552** (2017), http://arxiv.org/abs/1706.01552
31. Kurosawa, K., Ohtaki, Y.: How to update documents verifiably in searchable symmetric encryption. In: International Conference on Cryptology and Network Security. pp. 309–328. Springer (2013)
32. Kushilevitz, E., Lu, S., Ostrovsky, R.: On the (in)security of hash-based oblivious RAM and a new balancing scheme. In: SODA (2012)
33. Larsen, K.G., Nielsen, J.B.: Yes, there is an oblivious RAM lower bound! In: CRYPTO. pp. 523–542 (2018)
34. Patel, S., Persiano, G., Raykova, M., Yeo, K.: Panorama: Oblivious RAM with logarithmic overhead. FOCS (2018)
35. Ruemmler, C., Wilkes, J.: An introduction to disk drive modeling. IEEE Computer **27**(3), 17–28 (1994)
36. Shi, E., Chan, T.H.H., Stefanov, E., Li, M.: Oblivious RAM with $O((\log N)^3)$ worst-case cost. In: ASIACRYPT (2011)
37. Shi, E., Chan, T.H.H., Stefanov, E., Li, M.: Oblivious ram with o((logn)3) worst-case cost. In: ASIACRYPT. pp. 197–214 (2011)
38. Stefanov, E., van Dijk, M., Shi, E., Fletcher, C., Ren, L., Yu, X., Devadas, S.: Path ORAM – an extremely simple oblivious ram protocol. In: CCS (2013)
39. Van Liesdonk, P., Sedghi, S., Doumen, J., Hartel, P., Jonker, W.: Computationally efficient searchable symmetric encryption. In: Workshop on Secure Data Management. pp. 87–100. Springer (2010)
40. Vitter, J.S.: External memory algorithms and data structures. ACM Comput. Surv. **33**(2), 209–271 (2001)
41. Vitter, J.S.: Algorithms and data structures for external memory. Foundations and Trends in Theoretical Computer Science **2**(4), 305–474 (2006)
42. Wang, X., Chan, T.H., Shi, E.: Circuit ORAM: on tightness of the goldreich-ostrovsky lower bound. In: ACM Conference on Computer and Communications Security. pp. 850–861. ACM (2015)
43. Wang, X.S., Huang, Y., Chan, T.H.H., Shelat, A., Shi, E.: SCORAM: Oblivious RAM for Secure Computation. In: CCS (2014)
44. Williams, P., Sion, R.: Usable PIR. In: Network and Distributed System Security Symposium (NDSS) (2008)
45. Williams, P., Sion, R.: Round-optimal access privacy on outsourced storage. In: ACM Conference on Computer and Communication Security (CCS) (2012)
46. Williams, P., Sion, R., Carbunar, B.: Building castles out of mud: practical access pattern privacy and correctness on untrusted storage. In: CCS. pp. 139–148 (2008)

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.

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

- Let $\widehat{S}_1 = \langle \min(a_0, a_{n/2}), \min(a_1, a_{n/2+1}), \ldots, \min(a_{n/2-1}, a_{n-1}) \rangle$.
- Let $\widehat{S}_2 = \langle \max(a_0, a_{n/2}), \max(a_1, a_{n/2+1}), \ldots, \max(a_{n/2-1}, a_{n-1}) \rangle$.
- $\overrightarrow{S}_1 = \overrightarrow{\mathsf{Split}}(\widehat{S}_1)$, $\overrightarrow{S}_2 = \overrightarrow{\mathsf{Split}}(\widehat{S}_2)$ 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 [9] 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. Towards 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: $\mathsf{BitonicSort}(S)$

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 Figure 4 illustrates its locality. The $(2, O(\log^2 n))$ locality of Bitonic sort is also obvious from the figure.

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 $\overleftarrow{\mathsf{Split}}$).

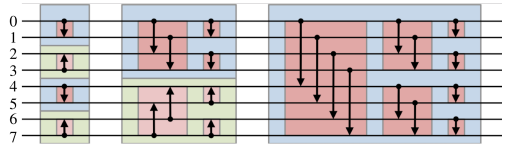

Fig. 3: Bitonic sorting network for 8 inputs. Input come in from the left end, and outputs are on the right end. When two numbers are joined by an arrow, they are compared, and if necessary are swapped such that the arrow points from the smaller number toward the larger number. This figure is modified from [1].

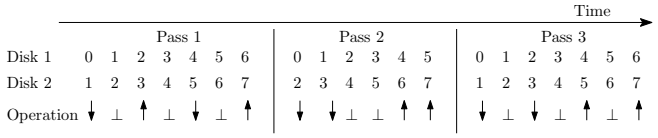

Fig. 4: Locality of Bitonic Sort for 8 elements. The figure shows the allocation of the data in the two disks for an 8 element array. For each input, either a compare-and-swap operation is performed in the specified direction or the input is ignored as denoted by $\perp$. The figure shows the first 3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