Oblivious Computation with Data Locality

Gilad Asharov
Cornell Tech
asharov@cornell.edu

T-H. Hubert Chan
The University of Hong Kong
hubert@cs.hku.hk

Kartik Nayak
UMD
kartik@cs.umd.edu

Rafael Pass
Cornell Tech
rafael@cs.cornell.edu

Ling Ren
MIT
renling@mit.edu

Elaine Shi
Cornell University
runting@gmail.com

Abstract

Oblivious RAM compilers, 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 (by shuffling the data to pseudorandom positions in memory).

In this work, we initiate the study of *locality-friendly oblivious RAMs*—Oblivious RAM compilers that preserve the locality of the accessed memory regions, while leaking only the lengths of contiguous memory regions accessed; we refer to such schemes as *Range ORAMs*. Our main results demonstrate the existence of a statistically-secure Range ORAM with only poly-logarithmic overhead (both in terms of the number of memory accesses, and in terms of locality). In our most optimized construction, the overhead is only a logarithmic factor greater than the best ORAM scheme (without locality).

To further improve the parameters, we also consider the weaker notion of a *File ORAM*: whereas a Range ORAM needs to support read/write access to arbitrary regions of the memory, a File ORAM only needs to support access to pre-defined non-overlapping regions (e.g., files being stored in memory). Assuming one-way functions, we present a computationally-secure File ORAM that, up to $\log \log n$ factors matches the best ORAM schemes (i.e., we essentially get "locality for free".)

As an intermediate result, we also develop a novel sorting algorithm which is also asymptotically optimal (up to $\log \log n$ factors) and enjoys good locality (can be implemented using $O(\log n)$ sequential accesses). This sorting algorithm can serve as a practical alternative to the previous sorting algorithms used in other oblivious RAM compilers and other applications, and might be of an independent interest.

To the best of our knowledge, before our work, the only works combining locality and obliviousness were for the special case of symmetric searchable encryption [Cash and Tessaro (EUROCRYPT'14), Asharov et al. (STOC'16)]. Searchable encryption can be viewed as a special case of a "read-only" File ORAM which leaks whether the same files are accessed again, and whether files contain the same keyword; this leakage, however, has been shown to be harmful in many applications, and is prevented by the definition of a File ORAM.

Keywords: Oblivious RAM, locality, randomized algorithms.

Contents

1 Introduction

Oblivious RAM [25,27,45], originally proposed in the seminal work by Goldreich and Ostrovsky [25, 27], is a cryptographic primitive that provably hides a program's access patterns to sensitive data. ORAM is a powerful building block with numerous applications in both theoretical settings (e.g., multiparty computation [31, 38], Garbled RAMs [24, 39]) as well as in secure systems (e.g., secure processors [21, 22, 40, 43], and cloud outsourcing [30, 46, 47, 58]). Thus far, *bandwidth overhead* has been considered as the the primary metric in the study of ORAM schemes. Numerous works [25,27, 30, 37, 45] have shown how to construct ORAM schemes with poly-logarithmic bandwidth blowup (namely, the ORAM must on average read and write poly-logarithmic blocks to in order to serve a memory request for a single block), and for sufficiently large blocks, one can construct ORAM schemes that almost tightly matches [52] the Goldreich and Ostrovsky logarithmic lower bound [25, 27].

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 modern computing platforms — from on-chip cache to memory and disk, almost every layer in the storage hierarchy has been highly optimized for locality. Specifically, caching and prefetching are prevalently adopted in between layers of the memory hierarchy: when data is fetched from a slower storage medium (e.g., DRAM) to a faster one (e.g., on-chip cache), typically, an entire cache-line worth of data is fetched at once. In addition, due to physical limitations, some storage mediums such as rotational hard-drives are significantly more efficient when accessing sequential regions rather than random seeks.

Unfortunately, existing ORAM schemes (e.g., [15, 25, 27, 45, 48, 52]) are not locality-friendly. Randomization in ORAM scheme is inherent due to the demand to hide the access pattern of the program, and ORAM schemes (pseudo-)randomly permute blocks and shuffle them around the memory. As a result, if a program wants to access a single region of $\Theta(N)$ *contiguous* blocks, all known ORAM schemes would have to access more than $N \log N$ random (i.e., discontiguous) memory 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 scheme cannot preserve locality — and indeed we formalize this intuition and formally show that any ORAM scheme that hides between these two access sequences with "good" locality must necessarily suffer from impractical bandwidth overhead.

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 memory regions. For example, a file server allows a client to retrieve a file at a time; a database may support range queries over time-series data, allowing a client to fetch all data, say, between a week ago and yesterday. We thus ask the question, can we design ORAM schemes that preserve the (possibly already publicly known) locality in the input request sequence? More specifically, we ask the following question.

Can we construct an ORAM scheme that preserves data locality while leaking only the lengths of contiguous regions accessed?

At first sight, even this relaxed goal also seems impossible to achieve with non-trivial efficiency.

On one hand, ORAM schemes need to break correlations between contiguous addresses, and do so by putting blocks into random positions. On the other hand, locality is usually obtained by highly structured memory layouts. Indeed, prior work considering obliviousness and locality [6, 12] focused on constructing locality-friendly searchable symmetric encryption schemes, a strictly weaker notion of security where much more leakage is allowed. In this setting, besides leaking the length of the contiguous segment accessed, the schemes also reveal the access pattern. Hiding this pattern while guaranteeing data locality is a much more challenging task.

1.1 Our Contributions

In this paper, we study oblivious RAMs with data locality and show that, somewhat surprisingly, such schemes can be constructed. We show efficient ORAM schemes that preserve the locality of the input request sequence: specifically, if the request is for a single contiguous region of size up to N, then the ORAM scheme accesses no more than polylogarithmically many contiguous regions. In comparison, existing ORAM schemes access super-linearly many contiguous regions in this scenario.

More specifically, we make the following contributions:

Modeling locality. First, we propose a relatively general notion of locality: we say that an algorithm has (h, l)-locality, if the algorithm needs to *sequentially* access l *contiguous* memory regions with h concurrent read/write *heads*. Roughly speaking, one may think of the number of concurrent heads as the number of concurrent cache-lines that a memory architecture supports. If multiple cache-lines have been prefetched, then sequentially reading the prefetched cache-lines is very fast. Alternatively, one may think of h as the number of concurrent read/write heads when the data is stored on a disk, where every movement of the heads introduces delays. We use the term "locality-friendly" to denote a scheme or an algorithm in which l is "small", and h is a small constant.

Locality with no leakage. We study whether locality can be obtained without leaking the length of the accessed region. We show that a scheme with good locality must incur expensive bandwidth overhead, even when allowing large space blowup, and $h = O(\mathsf{poly} \log N)$ heads. We show:

Theorem 1.1 (Our lower bound — informal)**.** *Any ORAM scheme with* $(O(\mathsf{poly} \log N), O(\mathsf{poly} \log N))$*-locality when accessing a region of size* len *has bandwidth blowup of* $\mathsf{len} \cdot \Omega(N^{1-\epsilon})$ *for any constant* $\epsilon > 0$*. This holds even when allowing total memory size* $O(N^2)$ *and CPU storage* $O(\mathsf{poly} \log N)$*, where* N *is the size of the logical memory.*

The lower bound implies that in order to obtain good locality and bandwidth, we must introduce some leakage on the size of accessed region. We discuss this leakage later in this section.

Range ORAM. We investigate ORAMs with data locality. We show that if an incoming request wants to access a possibly large contiguous memory region, then our oblivious simulation does not need to access too many contiguous regions as long as the memory or storage medium has $h = O(1)$ number of read/write heads. More specifically, we show a locality-friendly ORAM construction achieving poly-logarithmic bandwidth overhead and additionally satisfy $(O(1), \mathsf{poly} \log N)$-locality where N denotes the ORAM's logical memory capacity. We call this primitive as "Range ORAM", as this ORAM supports instructions of accessing ranges (regions) in the memory (as opposed to just single blocks as in ordinary ORAM). We prove the following informally stated theorem:

Theorem 1.2. *There exists a statistically secure Range ORAM construction consuming $O(N \log N)$ space with (amortized) $\mathsf{len} \cdot \tilde{O}(\log^3 N)$ bandwidth*[1] *and $(3, \tilde{O}(\log^3 N))$-locality, for accessing a range of length* len.

In comparison (see Table 1), for all existing ORAM schemes, accessing a single region of len contiguous blocks, the ORAM scheme will end up accessing $\mathsf{len} \cdot \omega(\log N)$ blocks residing at discontiguous physical locations. That is, by leaking only the length of the accessed range, we improve locality by a linear factor over current ORAM schemes.

We additionally extend our results to perfect security by leveraging the same modular construction but instead using perfectly secure building blocks. This gives rise to the following corollary.

Corollary 1.3. *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 $(O(1), \mathsf{poly} \log N)$-locality, for accessing a range of length* len.

File ORAM. In order to simulate instructions with address space N, our Range ORAM consumes $O(N \log N)$ space and incurs an additional logarithmic factor bandwidth in comparison with the best known, non-local ORAM. This gives rise to the question of whether locality-friendly ORAMs can be constructed with linear space, and matching the bandwidth cost of the best known non-local ORAM? We study this question for a relaxed notion of Range ORAM, and answer this question in the affirmative for that relaxation.

Specifically, as opposed to Range ORAM that supports any arbitrary range-accesses to the memory, in the relaxed primitive the set of allowed ranges is fixed in advance, and the ranges do not overlap. This setting naturally fits the application where a client stores several files of various sizes on the cloud and wants to access these files obliviously, while leaking only the sizes of the files it accesses but nothing else. We call such an ORAM scheme "File ORAM". This primitive has a direct application in the context of searchable symmetric encryption schemes, as we elaborate in the related work (Section 1.2). We prove the following theorem.

Theorem 1.4. *Assuming the existence of one-way functions, there exists a computationally-secure File ORAM consuming $O(N)$ space, and with $\mathsf{len} \cdot \tilde{O}(\log^2 N)$ (amortized) bandwidth and $(3, \tilde{O}(\log N))$-locality for accessing a file of size* len.

File ORAM achieves comparable efficiency to the best known ORAM candidates (barring $\mathsf{poly} \log \log$ terms), while achieving sub-exponential gains in terms of locality in comparison with best known ORAM schemes (see Table 1). Thus, essentially, we get locality for free!

Oblivious sorting. In designing locality-friendly Oblivious RAM, we created a new locality-friendly oblivious sorting algorithm that is optimal in runtime (up to $\mathsf{poly} \log \log$ factors) that may be of independent interest. Previously, although asymptotically optimal oblivious sorting algorithms exist (e.g., AKS [2,3], Zigzag [29], and Randomized Shell Sort [28]), these algorithms do not achieve locality, and moreover several of them [2, 3, 29] suffer from large constants in practice. Thus the non-optimal sorting algorithm Bitonic sort [7] with $O(n \log^2 n)$ runtime is often the scheme of choice in practical implementations. Our new oblivious sorting algorithm is conceptually simple: even disregarding locality, it could serve as a possible practical alternative to Bitonic sort especially for large data sizes.

[1] Throughout the paper, we let $\tilde{O}(f(n))$ to denote $O(f(n) \cdot (\log f(n))^c)$ for some constant c (might be negative).

Theorem 1.5 (Locality-friendly oblivious sort)**.** *There exists a statistically secure oblivious sort algorithm which, except with* $\mathsf{negl}(\lambda)$ *probability, completes in* $O(n \log n \log\log^2 \lambda)$ *work and with* $(3, O(\log n \log\log^2 \lambda))$ *locality.*

We conclude with summarizing our results in Table 1.

	Security	**Space**	**Bandwidth**	**Locality**	**Leakage**
This work I: Range ORAM	Stat.	$O(N \log N)$	$\mathsf{len} \cdot \tilde{O}(\log^3 N)$	$\tilde{O}(\log^3 N)$	len
This work II: File ORAM	Comp.	$O(N)$	$\mathsf{len} \cdot \tilde{O}(\log^2 N)$	$\tilde{O}(\log N)$	len
Ordinary ORAM [37, 52]	Comp./Stat.	$O(N)$	$\mathsf{len} \cdot \tilde{O}(\log^2 N)$	$\mathsf{len} \cdot O(\log^2 N)$	None
This work III: Lower bound	Stat.	$O(N^2)$	$\mathsf{len} \cdot \Omega(N^{1-\epsilon})$	$O(\mathsf{poly}\log N)$	None
This work IV: Oblivious sort	Stat.		$\tilde{O}(n \log n)$	$\tilde{O}(\log n)$	n

Table 1: Summarization of our results an comparison with conventional ORAM when accessing a contiguous range of length len. The lower bound holds for any $\epsilon > 0$.

On leaking the lengths of accessed regions. Our schemes leak the lengths of the accesses regions to achieve locality. Here, we mention several points regarding that leakage: (1) As mentioned, our lower bound implies that this leakage is inherent, as any ORAM scheme accessing sequential region with good locality without this leakage must incur unreasonable bandwidth blowup. (2) Our schemes can be viewed as a strict generalization of ordinary ORAM schemes, and we do not leak more than ordinary ORAM when the locality feature of the scheme is not invoked (i.e., when used without the additional supported functionality); (3) Even when the locality feature is invoked and the lengths of the regions accessed are leaked, in numerous applications we nonetheless offer security just as strong as ordinary ORAM: e.g., consider a range query application [34] where entries that fall within a queried range are returned. In this case, even ordinary ORAM would leak indirectly the number of matching entries (i.e., lengths) via the communication volume — and thus in such applications our Range ORAM provides just as strong security as ordinary ORAM. Similarly, in an outsourced file server setting, where the client stores and retrieves files from a server, absent further privacy mechanisms the length of a retrieved file is leaked even with an ordinary ORAM — and thus again, in such applications our File ORAM provides just as strong security as an ordinary ORAM. (4) 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. [35] suggested to offer strengthened privacy guarantees.

1.2 Related Work

Related work on locality. Algorithmic performance with data stored on the disk has been studied in the external memory models (e.g., [5, 44, 50, 51] and references within). Fundamental problems in this area include scanning, permuting, sorting, range searching, where there are known lower bounds and matching upper bounds. To the best of our knowledge, the only works that involve both locality and *obliviousness* are those in the area of searchable symmetric encryption.

Locality in searchable symmetric encryption. Searchable symmetric encryption (SSE) enables a client to encrypt an index of record/keyword pairs and later issue tokens allowing an untrusted server to retrieve the (identifiers of) all records matching a keyword. The typical approach

(e.g., [14, 17, 33, 36, 49], and references within) is to store an inverted index. As aiming to provide truly practical solutions, SSE schemes offer much weaker security guarantees than ORAMs. The security requirement is that the server does not learn any information about keywords for which the client did not issue any queries. Nevertheless, the access patterns of searched keywords are revealed, and the server can deduce whether two different keywords appear in the same document, whether the same keyword is queried twice, or to identify a popular document. Recent works [10, 32, 59] show that this may lead to statistical inference attacks based on side information on data. Hiding the access pattern (while still leaking the lengths of the accessed lists) would significantly increase the security of SSE schemes.

The issue of locality in SSE surfaced in recent works [11, 17] where implementations of SSE schemes imply that locality is a crucial efficiency measure for scaling the schemes to large databases. Locality in SSE was first studied (in the theoretical perspective) in the work of Cash and Tessaro [12], showing that any searchable symmetric encryption scheme must be sub-optimal in either its space consumption, locality, or bandwidth. Intuitively, if a scheme is very local and efficient in terms of bandwidth, then after several queries the server can look at memory locations accessed and infer statistics about what has not been opened. In particular, if one of the keywords appears in many documents, its list of identifiers is very long. A scheme with good locality and bandwidth means there is a very large region of the encrypted index that will not be touched by other searches, and the server will notice that this happens after several searches with small number of results. The way to overcome this attack is by padding the lists (leading to non-optimal space consumption), by accessing more addresses with each query (non-optimal locality), or by retrieving more elements than needed (non-optimal bandwidth).

Asharov et al. [6] presented schemes with optimal locality, space and almost optimal bandwidth, using subtle balanced allocation techniques. Specifically, one scheme offers $\tilde{O}(\log \log N)$ bandwidth, under the assumption that no keyword appears in more than $N^{1-1/\log \log N}$ documents. Another scheme does not require such an assumption, and offers $\tilde{O}(\log N)$ bandwidth. On an intuitive level, these constructions achieve an obliviousness notion somewhat similar to "one-time ORAM": the access pattern remains hidden as long as each keyword is read only once and a document does not contain two previously queried keywords.

We remark that our File ORAM primitive fits the application of searchable encryption, while guaranteeing much stronger security while simultaneously offering data locality and optimal storage. One can store each list of identifiers corresponding to some keyword in a separate file, and then store all files in a File ORAM. With each query, the client first fetches the file containing the list of identifiers corresponding to its keyword, decrypts this list of identifiers, and then retrieve each one of the documents. The security guarantee here is significantly stronger than ordinary SSE; however, this of course comes with a price in efficiency. We compare File ORAM and current SSE schemes in Table 2.

ORAM Scheme	**Space**	**Bandwidth**	**Locality**	**Leakage**
SSE [14, 17, 33, 36, 49]	$O(N)$	$\mathsf{len} \cdot O(1)$	$O(\mathsf{len})$	Access Pattern, Sizes
SSE I [6]	$O(N)$	$\mathsf{len} \cdot \tilde{O}(\log N)$	$O(1)$	Access Pattern, Sizes
SSE II [6] (*)	$O(N)$	$\mathsf{len} \cdot \tilde{O}(\log \log N)$	$O(1)$	Access Pattern, Sizes
This work: File ORAM	$O(N)$	$\mathsf{len} \cdot \tilde{O}(\log^2 N)$	$\tilde{O}(\log N)$	Sizes

Table 2: Comparison between efficiency of searchable symmetric encryption schemes in linear space and our File ORAM upon accessing a keyword of size len. (*) The second scheme of [6] assumes that no keyword appears in more than $O(N^{1-1/\log \log N})$ documents.

The notion of locality introduced by [12] is more restricted than our locality model. It essentially corresponds to storage devices with a single read/write head, i.e., matches our notion of locality $(1, l)$. Although their model captures a wide range of practical storage devices, it fails to separate, say, two algorithms — one that accesses memory at random (e.g., quicksort), and the other scans through 3 arrays concurrently (e.g., merging two sorted arrays into one), where our locality model successfully captures such algorithms.

Oblivious RAM (ORAM). Numerous works [30, 37, 42, 45, 48, 52, 53, 55–57] construct ORAMs in different settings. Most of ORAM constructions follow one of two frameworks: the hierarchical framework, originally proposed by Goldreich and Ostrovsky [25, 27], or the tree-based framework proposed by Shi et al. [45]. To date, some of the (asymptotically) best schemes include the following: 1) Kushilevitz et al. [37] showed a *computationally secure* ORAM scheme with $O(\log^2 N / \log\log N)$ runtime blowup for general block sizes; and 2) Wang et al. construct Circuit ORAM [52], a *statistically secure* ORAM that achieves $O(\alpha \log^2 N)$ runtime blowup for general block sizes and $O(\alpha \log N)$ runtime blowup for large enough blocks, for any super-constant function α.

On the lower bound side, Goldreich and Ostrovsky [25, 27] demonstrated that any ORAM scheme (with constant CPU cache) must incur at least $\Omega(\log N)$ runtime blowup. Boyle and Naor [8] discussed some limitations of the model this lower bound captures, but showed that it is hard to remove these limitations. Further, the Goldreich-Ostrovsky lower bound is also known not to hold when the memory (i.e., ORAM server) is capable of performing computation [4, 19], which is beyond the scope of this paper.

To the best of our knowledge, all previous works in ORAM ignore the issue of locality.

2 Technical Roadmap

In this section we provide a high-level overview of our results. As it turns out, obtaining the aforementioned results, i.e., $\widetilde{O}(\log^3 N)$ Range ORAM and $\widetilde{O}(\log^2 N)$ File ORAM requires the combination of numerous building blocks. To aid understanding, first, in Sections 2.1 and 2.2, we describe a high-level blueprint that will allow us to construct Range ORAM and File ORAM achieving *polylogarithmic* bandwidth blowup and locality blowup. Then, in Section 2.3, we describe how to improve the efficiency of important building blocks to further save up to polylogarithmic factors — these building blocks, such as locality-friendly oblivious sorting and locality-friendly oblivious data structures, are also important in their own right due to the usefulness of sorting and data structures in many applications.

2.1 Range ORAM

Intuitively speaking, a *Range ORAM* is a machine that interacts with the memory and receives instructions from the CPU. It supports read/write instructions of arbitrary ranges to the memory, rather than just single blocks as in ordinary ORAM. 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 rather than these lengths.

Basic scheme: read-only Range ORAM. Assuming that the CPU sends only read instructions, we can achieve data locality and obliviousness in the following way. The basic idea is to make replications of a set of super-blocks that form contiguous memory regions. Specifically, let N

bound the size of the logical memory (and assume without loss of generality that N is a power of 2). 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 $\log N$ 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 length, reading any contiguous memory of length 2^i region would boil down to making two accesses to the i-th ORAM.

Indeed this approach would work if the ORAM scheme is *read-only*. However, it would break down if we also need to support writes. 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. Interestingly, the way we solve this issue is not by avoiding replicating data, but rather by introducing more replication.

Range trees. While read-only range ORAM does not suffice for our goal, its structure still plays an essential role in our final construction, as it provides freedom to access the same data using different block sizes obliviously. We define an intermediate logical data structure which we call "Range tree". Range trees are essentially read-only range ORAM with the following extended functionality: While we defined read-only Range ORAM on the whole memory, we allow range tree to be defined on partial (not necessarily contiguous) part of the memory, and contain only 2^i total primitive blocks, for some $i \in \{1, \ldots, \log N\}$. Besides primitive blocks, range tree also contains replication of the data with different block sizes. In more details, a range tree of size 2^i is the following (logical) data-structure: the leaves store 2^i primitive blocks sorted by their addresses, whereas each internal node replicates and stores all blocks contained in the leaves of its subtree. Namely, each node at height j stores a super-block of size 2^j (leaves have height 0 and store primitive blocks). A Range tree supports range queries in for the form $[s, t]$, and returns all blocks in the tree with addresses within $[s, t]$. A requested range might not exist in the tree. Lookups for a range should guarantee obliviousness, and should not reveal whether or not the range was present in the tree. Range trees do not support writes, and all replicas of the same primitive block within a Range tree are identical.

We construct Range tree of size 2^i by storing i different ORAMs, one ORAM for each height of the tree. Each ORAM operates on a different block size, corresponding to the block-size of the height it stores. As we will mention soon, this ORAM must also have a locality-friendly initialization procedure. In order to allow lookup, we build a metadata ORAM that stores a binary search tree, such that given a desired memory range $[s, t]$ of length 2^i we may find out which height-i super blocks we would like to request within this range tree. Note that the previous solution of read-only Range ORAM is in fact $2^{\log_2 N}$-range tree storing the entire memory, and since all ranges appear in the tree, the additional metadata is redundant. In order to lookup for an arbitrary range $[s, t]$ of size 2^j in a 2^i-range tree, one has to perform at most two lookups for 2^j-blocks when $j < i$, and read only the root (for receiving partial data) when $j > i$.

A hierarchy of range trees. In order to support writes in Range ORAM, we construct a hierarchy of range trees. This idea is inspired by the hierarchical ORAMs [25,27] which employ an exponentially growing hierarchy of oblivious hash tables. Here, instead of using a hash table for each level of the hierarchy as in ordinary ORAM, we adopt a range tree instead.

We construct a hierarchy of range trees henceforth denoted $\mathsf{T}_0, \mathsf{T}_1, \ldots, \mathsf{T}_L$ where $L := \log_2 N$. T_i contains 2^i primitive blocks. Observe that the last tree T_L contains all data, and replications in

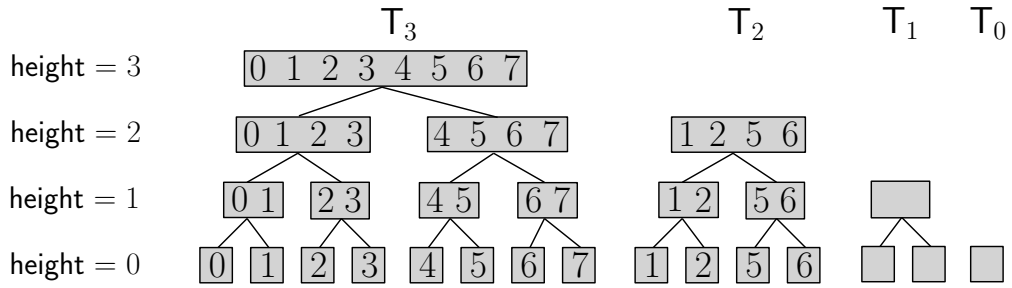

Figure 1: Hierarchy of range trees. Logically, data is divided into trees of exponentially increasing sizes. In each tree block, a parent super-block stores the contents of both its children. If a block appears in more than one tree, the smallest tree contains the freshest copy. The above figure shows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

all possible block-sizes, and therefore contains the same data as the aforementioned read-only Range ORAM solution. The extra levels $\mathsf{T}_0, \ldots, \mathsf{T}_{L-1}$ are additional replications that give us some more freedom in accessing data and maintaining data coherence. We maintain the following important invariant:

- If blocks for the same address appear in multiple range trees, the block in a smaller tree of the hierarchy is always fresher than the one in a larger tree.

Given the hierarchy of $\log N$ range trees and the aforementioned invariant, we can perform the following to read or write requests for any contiguous memory region $[s, t]$ of length $t - s + 1 = 2^i$. Assuming a power of 2 is without loss of generality, since if not, we can pad it to the nearest power of 2. Suppose that each level has a bit whose value is either *empty* or *ready*. Each access of a range $[s, t]$ first collects all copies of the data from all trees, merges all trees $\mathsf{T}_0, \ldots, \mathsf{T}_{\ell-1}$ to the first empty tree T_ℓ with $\ell > i$, and then writes the fetched data back to the tree T_i (instead of T_0 as in ordinary ORAM). In a more detail:

> **Read Phase: Read and collect the range $[s, t]$.** For each range tree $\mathsf{T}_i, \ldots, \mathsf{T}_L$, fetch all size-$2^i$ super-blocks that intersect the range requested. For each range tree $\mathsf{T}_0, \ldots, T_{i-1}$, fetch the super-block contained in its root (whose size is $< 2^i$). Reconstruct the freshest state of all blocks requested. That is, if the same address appear in multiple trees, take the one from the smallest tree. Let D denote the collected elements.
>
> **Write Phase: Write D back to the tree T_i.** Find the first empty level $\ell > i$. If no such level exists, let $\ell := L$. Now, merge all T_j where $j < \ell$ into T_ℓ, keeping only the freshest copy for each block. If $\ell = L$, then merge all levels into T_ℓ. Mark all levels $\mathsf{T}_1, \ldots, \mathsf{T}_{\ell-1}$ as empty, and T_ℓ as ready. Construct the logical range tree T_i to contain D, and mark it as ready.

As for obliviousness, the Read Phase is oblivious due the obliviousness of the range trees. As for the Write Phase, which level is being accessed and rebuilt is determined by the length of the accessed ranges (and is independent of which ranges are being accessed). We will show how to implement the eviction procedure, and the rebuilding of the trees with locality-friendly and oblivious algorithms.

How data locality is preserved. We highlight why this solution preserves data locality.

- *Read phase locality.* In order to access a possibly large memory range of length 2^i, we need to *i)* access at most polylogarithmically many metadata blocks; and *ii)* for each of the logarithmically

many range trees, fetch $O(1)$ size-2^i super-blocks from each range tree. It is not hard to see these amount to reading polylogarithmically many contiguous memory ranges.

- *Write phase locality.* Write-phase locality can be guaranteed, as long as the underlying ORAMs we employ (to store the metadata as well as super-blocks in each height of each range tree) supports locality-friendly initialization (i.e., batched rebuilding). We show that such initialization can be achieved with *locality-friendly oblivious sorting*.

2.2 File ORAM

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

Obviously, we can use Range ORAM to realize File ORAM — but since File ORAM is can be very useful abstraction in practice, we ask the question: can we construct a File ORAM that is asymptotically more efficient, and potentially matching the bandwidth blowup of the best known, non-local ORAM scheme? Can we construction a File ORAM that matches the space overhead of known ORAM schemes (i.e., linear overhead)? This latter question is also interesting as there is an evidence for the tradeoff between space overhead and locality, as the lower bound of [12] implies. We answer the above question in the affirmative: we show that File ORAM can be realized in linear space and $\tilde{O}(\log^2 N)$ work (ignoring $\mathsf{poly}\log\log$ factors) while achieving polylogarithmic locality. In this section, we will first give a blueprint that does not care about optimizing $\mathsf{poly}\log$ factors — to obtain the final $\tilde{O}(\log^2 N)$ result would require additional techniques described in Section 2.3.

Naïve construction. A naïve construction for File ORAM is the following. Given a file structure $\mathcal{F} = (F_{\mathsf{fid}_1}, \ldots, F_{\mathsf{fid}_k})$ of a total size N, we build $\log N$ different ORAMs where the i-th ORAM holds up to $N/2^i$ files of size 2^i. If we use Circuit ORAM [52] with a merged stash across all recursion levels [13], accessing each file of 2^i boils down to accessing $O(\log^2 N)$ super-blocks of length 2^i. It is not hard to see that this naïve scheme achieves a $O(\log^2 N)$ work and $(3, O(\log^2 N \log\log^2 N))$-locality. However, the scheme requires $O(N \log N)$ space, which brings us back to our starting point.

Non-recurrent file hashing scheme. We start with constructing File ORAM for non-recurrent accesses, namely, when the same file is accessed at most once. At a high level, to achieve this, we define a new primitive, which we call "Non-recurrent File Hashing scheme", build upon the two-dimensional balanced allocation scheme [6]. However, we show how to perform this balanced allocation, *obliviously*.

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

When the user accesses file fid, we compute the starting bin g by applying the pseudorandom function on fid, and then read the $\mathsf{len}(\mathsf{fid})$ consecutive bins $g, \ldots, g + \mathsf{len}(\mathsf{fid}) - 1$ to retrieve all the blocks that correspond to the file fid. This guarantees good locality, as all the data that is associated with the file fid is stored in len consecutive bins, i.e., a contiguous region of length $\mathsf{len} \cdot \tilde{O}(\log N)$. As no file is accessed more than once, this range of memory locations is pseudorandom. As such, we can also support accesses of fictitious files in a non-recurrent hashing scheme: when accessing a file with $\mathsf{fid} = \perp$ and some len, we choose a random bin g uniformly at random, and read the bins $g, \ldots, g + \mathsf{len} - 1$.

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

Achieving obliviousness for recurrent memory requests. To make recurrent requests, we make the following observation: the hierarchical ORAM framework originally proposed by Goldreich and Ostrovsky [25,27], is in fact, a method for constructing a recurrent ORAM from a non-recurrent ORAM. Thus, we will apply the hierarchical ORAM framework atop our oblivious non-recurrent File Hashing scheme, somewhat similarly to the way we converted Range trees to Range ORAM. Here, however, the ith table T_i is a non-recurrent file hashing of size 2^i which consumes space $O(2^i)$, whereas range tree of size 2^i consumes space $O(i \cdot 2^i)$. As a result, the total space consumption of our file ORAM is $O(N)$ and not $O(N \log N)$ as in our Range ORAM.

2.3 Locality-Friendly Oblivious Sort

The aforementioned blueprint would suffice for us to obtain both Range ORAM and File ORAM with polylogarithmic bandwidth and locality blowup. To obtain asymptotically tighter parameters, i.e., the aforementioned $\widetilde{O}(\log^3 N)$ bandwidth blowup for Range ORAM and $\widetilde{O}(\log^2 N)$ for File ORAM, we need various additional techniques.

In particular, as mentioned earlier, both the Range ORAM and the File ORAM relies on a locality-friendly oblivious sorting algorithm to allow locality-friendly initialization, and thus the efficiency of our final algorithm will depend on the efficiency of oblivious sorting.

Among known oblivious sorting algorithms, we observe that there is a way to implement bitonic sort such that it preserves data locality — unfortunately, bitonic sort incurs an additional logarithmic factor in cost in comparison with the best known oblivious sorting algorithm (without locality). We thus ask the question, can we realize an oblivious sorting algorithm with locality whose bandwidth overhead matches that of the best known oblivious sorting algorithm (barring $\mathsf{poly}\log\log$ factors), i.e., can the locality come (almost) for free in oblivious sorting?

Indeed, we show how to achieve this by demonstrating a new, statistically secure oblivious sorting algorithm with $\widetilde{O}(n \log n)$ overhead for sorting n elements, and polylogarithmic locality.

Intuitively speaking, we observe that the composition of the following two algorithms yields an oblivious sorting algorithm:

- First, we *obliviously* permute the input array at random;
- Second, we apply a *non-oblivious*, comparison-based sorting algorithm on the permuted array (e.g., merge-sort).

It is not hard to see that combining these two steps yield in an oblivious sort even though the second step is non-oblivious. We show how to implement the first step efficiently and obliviously.

In fact, we do not fully implement an oblivious permutation. Instead, we implement a weaker primitive such that the composition still works and results in an oblivious algorithm.

In particular, we implement an "oblivious random bin assignment" algorithm whose core idea is a butterfly-like shuffle. This random bin assignment algorithm places blocks into random bins of relatively small size, and pads each bin with dummy elements. We later remove these dummy elements using the non-oblivious sort.

Our new locality-friendly oblivious sorting algorithm immediately improves the performance of both Range ORAM and File ORAM by a logarithmic factori. As oblivious sorting is a powerful building block in the design of a wide class of oblivious algorithms, we believe that our locality-friendly oblivious sorting algorithm is of independent interest. For example, several earlier works [30, 38, 41] have shown that oblivious sorting is sufficient for making any MapReduce or GraphLab algorithms oblivious while incurring asymptotically smaller overhead than generic ORAM.

Finally, we mention that besides a more efficient oblivious sorting algorithm, to obtain the aforementioned $\widetilde{O}(\log^3 N)$ Range ORAM result also requires leveraging additional tools such as locality-friendly oblivious data structures and position-based ORAMs — clearly, these building blocks themselves can be of independent interest.

2.4 Organization

The rest of the paper is organized as follows. In Section 3 we define the notion of locality and oblivious simulations. In Section 4 we present several building blocks that are necessary for our construction, such as oblivious sort, oblivious deduplication and locally initializable ORAM, and state that all of them can be constructed with good locality. We delay the construction of our optimized oblivious sort to Section 7. Our main constructions of Range ORAM and File ORAM are given in Section 5 and 6, respectively. Finally, in Section 8 we show our lower bound, formalizing that efficient locality-friendly ORAM must leak the size of the lengths of the memory regions accessed.

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 n's, it holds that $\mathsf{negl}(n) < n^{-c}$. 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 [25, 27, 30, 37, 45, 48, 52].

3.1 Memory with Multiple Read/Write Heads and Data Locality

To understand the notion of data locality, it may be convenient to view the memory as a rotational hard drive or other possible storage medium where sequential accesses are faster than random accesses. Assume that the memory has H read/write heads, and in order to serve a `read` or `write` request with address addr, the memory has to move one of its read/write heads to the physical location addr to perform the operation. Any such movement of the heads introduces cost and delays, and the machine that interacts with the memory would like to minimize the move heads operations. Traditionally, the latter can be improved by storing related items to physically proximate areas on 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 $\mathsf{mem}[N, b, \mathsf{H}]$, and is indexed by the address space $[N] = \{1, 2, \ldots, N\}$, where 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 mem is equipped with H read/write heads and supports the following two types of instructions.

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

Locality. The number of move operations defines locality. A sequence of memory operations has (H, l)-locality if it contains l move operations to a memory that is equipped with H heads.

3.2 Oblivious Machines

In this section, we define oblivious simulation of possibly randomized functionalities, either stateless (non-reactive) or stateful (reactive). Most prior works considered oblivious simulation of deterministic functionalities; however, in our paper, we will need oblivious simulation of randomized functionalities as well. Thus we generalize the definitions to allow oblivious simulation of randomized functionalities as well. Moreover, we capture a stronger notion than what is usually considered, in which the adversary is adaptive and can issue request as a function of previous access pattern.

Oblivious simulation of a stateless functionality. We consider machines that interact with the memory via move and `read`/`write` operations (throughout the paper it is often clear from context how M should move the memory heads, in which case we omit explicitly writing move operations). In case of a stateless (non-reactive) functionality, the machine M receives one instruction I as input, interacts with the memory, compute the output and halts.

Given a stateless, possibly randomized functionality f, and a leakage function $\mathsf{leakage}$, we say that M obliviously simulates f if M correctly computes the same (possibly randomized) function as f except with negligible probability for all inputs, and moreover, the access pattern of the instructions M sends to the memory do not leak anything beyond the allowed leakage. The formal definition follows.

Let $\textsc{Real}^M(1^\lambda, I) := (y, \mathsf{addresses})$ be a pair of random variables where y denotes of the outcome of executing $M(1^\lambda, I)$ on input I, and and $\mathsf{addresses}$ represents the addresses incurred during the execution. Let $\mathcal{F}^{f,\mathsf{leakage}}$ be a wrapper functionality that outputs a pair $f(I;\rho), \mathsf{leakage}(I;\rho)$ where

the same randomness ρ is given to the leakage function and to f (we note that $\mathsf{leakage}$ might consume some additional random coins; nevertheless, it receives the randomness f used to compute the function). The simulator receives this leakage, and has to simulate the access pattern as in the real, without receiving the output of the function. Formally:

Definition 3.1 (Oblivious simulation of a stateless (non-reactive) functionality)**.** *We say that the stateless algorithm M obliviously simulates a stateless, possibly randomized functionality f w.r.t. to the leakage function* $\mathsf{leakage} : \{0,1\}^* \to \{0,1\}^*$*, iff there exists a p.p.t. simulator* Sim *and a negligible function $\epsilon(\cdot)$, such that for any λ and I,*

$$\textsc{Real}^M(1^\lambda, I) \overset{\epsilon(\lambda)}{\equiv} \{y, \mathsf{Sim}(1^\lambda, L) \;\mid\; (y, L) \overset{\$}{\leftarrow} \mathcal{F}^{f,\mathsf{leakage}}(I)\}$$

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.

Intuitively, the above definition requires indistinguishability of the joint distribution of the output of the computation and the access pattern, somewhat similar to the standard definition of secure computation in which the joint distribution of the output of the function and the view of the adversary is considered (see the relevant discussions in [9, 26]). Note that here we handle correctness and obliviousness in a single definition.

As an example, consider an algorithm that receives an instruction to randomly permute some array in the memory, while leaking only the size of the array. Ignoring hiding the input for now, such a task should also hide the chosen permutation. As such, our definition requires that the simulation would output an access pattern that matches a permutation that the ideal functionality chose, without seeing that chosen permutation. This guarantees that the access pattern does not reveal which permutation was chosen. On the other hand, if we do not consider the joint distribution of the output and the access pattern, then we allow algorithms in which there is some mapping between the access pattern and the chosen permutation, which are not oblivious.

Special case for deterministic f. As a special case, if the functionality f is deterministic, then the above definition equates to the classical notion of oblivious simulation of a deterministic functionality, requiring:

- **Correctness:** there exists a negligible function $\epsilon(\cdot)$ such that for every λ and I, $M(1^\lambda, I) = f(I)$ except with $\epsilon(\lambda)$ probability.
- **Obliviousness:** there exists a stateful p.p.t. simulator Sim, such that for any λ and I,

  $$\mathsf{Addresses}(M(1^\lambda, I)) \overset{\epsilon(\lambda)}{\equiv} \mathsf{Sim}(1^\lambda, \mathsf{leakage}(I))$$

  where $\mathsf{Addresses}(M(1^\lambda, I))$ is a random variable denoting the addresses incurred by an execution of M over the input I. Note that in this case, the $\mathsf{leakage}$ function takes only I as input, but not the randomness of the functionality f.

For example, an oblivious sorting algorithm is an oblivious simulation of the functionality that receives an array and sorts it, 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 and the simulator Sim are interactive machines that might receive instructions (or leakage) as long as they are activated.

Intuitively, the machine M receives instructions from the CPU and interacts with the memory in order to answer them. In the experiment, we let the adversary $\mathcal{A}$ issue the instructions, and choose the next instruction adaptively. In the real execution, the machine M receives the instruction I, interacts with the memory and answers the instruction. The adversary receives both the answer and also sees the memory addresses M accessed. In the ideal execution, when $\mathcal{A}$ outputs an instruction, the functionality computes the answer, and the memory addresses are simulated by the simulator upon receiving some leakage. Formally,

Definition 3.2 (Adaptively secure oblivious simulation of stateful functionalities)**.** *Let M be an interactive machine, and let* $\mathsf{leakage} : \{0,1\}^* \to \{0,1\}^*$ *be a* stateful *leakage function. We say that M obliviously simulates a possibly randomized, stateful functionality f w.r.t. to the leakage function* $\mathsf{leakage}$ *iff there exists a p.p.t. simulator* Sim*, such that for any non-uniform p.p.t. adversary $\mathcal{A}$, $\mathcal{A}$'s view in the following two experiments,* $\mathsf{Expt}_{\mathcal{A}}^{\mathrm{real},M}$ *and* $\mathsf{Expt}_{\mathcal{A},\mathsf{Sim}}^{\mathrm{ideal},f}$ *are computationally indistinguishable — below we use the notation $\mathcal{F}^{f,\mathsf{leakage}}$ to denote a stateful wrapper functionality that upon receiving every new input I_i, outputs a pair $f(\mathsf{state}, I_i; \rho)$ and $\mathsf{leakage}(\mathsf{state}, I_i, \rho)$ where ρ is the randomness consumed by f in the execution, and updates the internal* state*.*

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

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

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.

Our definition of oblivious simulation is in the form of a general wrapper for 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 [25], is an oblivious simulation of a "logical memory functionality",

- **Functionality:** 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 $\mathsf{data} \in \{0,1\}^b \cup \{\perp\}$ perform reads and writes to a logical memory, such that the block fetched is always the last value written. In particular, if $\mathsf{op} = \texttt{read}$, the block at the logical address addr is fetched; and if $\mathsf{op} = \texttt{write}$, the content of the block with address addr is updated to data.

- **Leakage:** The leakage of an ORAM is only the total number of blocks, and the total number of accesses, i.e., $\mathsf{leakage}(\mathbf{I}) := (N, m)$.

We remark that previous constructions of ORAM [37, 48, 52] in fact satisfy Definition 3.2.

Locality, work, and private storage of oblivious machines. A machine M has locality (H, ℓ) if for every sequence of instructions $\mathbf{I}$ of length n, the machine M sends at most $\ell(n)$ move operations to a memory that is equipped with H read/write heads. Throughout the paper, we use the terminology *work* (or interchangably, bandwidth) to denote the total number of memory read/write operations of size $\Omega(\log N)$ a machine needs to use; we use the terminology *oblivious simulation overhead* or *overhead* for short, to denote the work consumed by an oblivious machine divided by the work consumed by (the optimal) non-oblivious machine for the same task. The entire paper assumes the machine/algorithm has only $O(1)$ blocks of private storage.

Remark. In this paper, we focus on hiding the access patterns to the memory, but not the data contents. Therefore, we do not explicitly mention that data is (re-)encrypted when it is accessed, but encryption should be added if the adversary can observe memory contents.

4 Locality-Friendly Building Blocks

In this section we describe several locality-friendly building blocks that are necessary for our constructions. In Section 4.1 we overview sorting algorithms that are locality-friendly. In Section 4.2 we describe the oblivious deduplication functionality, which can be implemented directly from oblivious sort. In Section 4.3 we describe oblivious RAM with locality-friendly initialization.

4.1 Oblivious Sorting Algorithms with Locality

An important building block for our constructions is an oblivious sorting algorithm that is locality-friendly. In this section, we show how to construct such an algorithm. We first discuss Merge-sort as an example of a local algorithm, but not asymptotically optimal. We proceed to Bitonic-Sort, which is a perfectly secure oblivious sorting algorithm with locality but runs in time $O(n \log^2 n)$ for sorting an array of size n. While it suffices for our purposes, it introduces an extra overhead. We then proceed to construct a novel sorting algorithm that has optimal overhead (up to $\log \log \lambda$-factors), and is statistically-oblivious.

Merge sort. As a warmup, we note that merge sort is a *non-oblivious* sort algorithm that is locality-friendly. It is easy to see that the basic operation that merges two sorted lists of size $n/2$ into a sorted list of size n can be implemented by a single scan with three heads, one head for each of the input arrays and one head for the destination array. In addition, each iteration $i = 0, \ldots, \log n - 1$ that merges $n/2^i$ pairs of subarrays of size 2^{i-1} into $n/2^i$ sorted arrays of size 2^i can be implemented using one linear scan, and without introducing additional move operations. Specifically, after merging two subarrays, the heads are incremented to the consecutive subarrays using vacuous reads, and move operations are required only in the beginning of each iteration.

As a result, the merge sort algorithm can be implemented by scanning the array $\log_2 n$ times with three heads, resulting in $(3, \log_2 n)$ locality and $O(n \cdot \log_2 n)$ work for an input array of size n.

Bitonic sort. An *oblivious* sorting algorithm that enjoys good locality is Bitonic sort, introduced by Batcher [7]. In Appendix A, we describe a particular method to implement Bitonic sort to achieve locality, and provide a detailed analysis.

Theorem 4.1 (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 $O(n \log^2 n)$ work and $(2, O(\log^2 n))$ locality.*

A new oblivious sorting algorithm with locality. While Bitonic sort suffices for our purposes for constructing ORAMs with locality, it is not asymptotically optimal and introduces an additional $\log n$ factor. In Section 7 we construct a novel sorting algorithm that is locality-friendly and asymptotically faster. We show:

Theorem 4.2 (Statistically secure, efficient oblivious sort)**.** *There exists a locality-friendly oblivious sort algorithm with statistical security, with $O(n \log n \log\log^2 \lambda)$ work and $(3, O(\log n \log\log^2 \lambda)$ locality, where λ is the security parameter.*

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 oblivious sort with locality, 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.

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.

- $\mathsf{T.Build}(X)$ takes an input array X of blocks of the form $(\mathsf{addr}, \mathsf{data})$. Blocks in X have distinct integer addresses that are not necessarily contiguous. It creates the data-structure in the memory and outputs some secret state to the CPU.
- $\mathsf{B} \leftarrow \mathsf{T.Access}(\mathsf{op}, \mathsf{addr}, \mathsf{data})$ with $\mathsf{op} \in \{\texttt{read}, \texttt{write}\}$, $\mathsf{addr} \in [N]$, and $\mathsf{data} \in \{0,1\}^B$. If $\mathsf{op} = \texttt{read}$, then the ORAM returns the block with address addr, which should match the content lastly written to that block. If $\mathsf{op} = \texttt{write}$, the content of the block with address addr is updated to data.

The leakage function of locally initializable ORAM reveals $|X|$ and the number of $\mathsf{Request}$ operations. Obliviousness is defined as in Definition 3.2 with the above leakage and functionality.

Locality-friendly initialization. We now show that the hierarchical ORAM by Goldreich and Ostrovsky [25] can be initialized in a locality-friendly manner. 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 [25] 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 — specifically, Bitonic sort or our new sorting algorithm (Section 7). This gives rise to the following theorem:

Theorem 4.3 (Computationally secure, locally initializable ORAM)**.** *Assuming one-way function exists, there exists a computationally secure locally-initializable ORAM scheme that has* $\mathsf{negl}(\lambda)$ *failure probability, and can be initialized with* n *blocks using* $(n+\lambda) \cdot \mathsf{poly}\log(n+\lambda)$ *work and* $(2, \mathsf{poly}\log(n+\lambda))$ *locality, and can serve an access using* $\mathsf{poly}\log(n+\lambda)$ *work 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 first present a construction with poly-logarithmic work and poly-logarithmic locality to show feasibility. The construction uses a building block which we call an oblivious range tree (Section 5.2). It supports range lookup queries with low overhead and good locality, but it does not support updates. Next, we show how to construct a range ORAM, which supports reads and updates, from oblivious range trees (Section 5.3). Finally, we introduce techniques to improve the efficiency of range ORAM (Section 5.4).

5.1 Range ORAM Definition

A Range ORAM is an oblivious machine that receives read/write range instructions, and interacts with the memory while leaking only the size of the range. Formally, Range ORAM is defined using Definition 3.2 using the following functionality and leakage:

Functionality: Range ORAM takes as input range requests in the form $\mathsf{Access}(\mathsf{op}, [s,t], \mathsf{data})$, where $\mathsf{op} \in \{\mathtt{read}, \mathtt{write}\}$, $s, t \in [N]$, $s < t$, and $\mathsf{data} \in (\{0,1\}^b)^{(t-s+1)}$. If $\mathsf{op} = \mathtt{read}$, then all blocks with addresses in the range $[s,t]$ are returned. If $\mathsf{op} = \mathtt{write}$, data denotes the updated value of blocks in the range $[s,t]$.

Leakage: A Range ORAM reveals to the adversary only its capacity and the length $t-s+1$ of every request and the total number of requests. Formally, given a sequence of instructions

$$\begin{aligned} \mathbf{I} &= ((\mathsf{op}_1, [s_1,t_1], \mathsf{data}_1), \ldots, (\mathsf{op}_m, [s_m,t_m], \mathsf{data}_m)) \text{ , we define} \\ \mathsf{leakage}(\mathbf{I}) &:= (N, t_1 - s_1 + 1, \ldots, t_m - s_m + 1) \end{aligned}$$

In Section 8, we show a lower bound stating that if we restrict the leakage and do not allow the adversary to learn the length of each operation, then simultaneously achieving a poly-logarithmic bandwidth and poly-logarithmic locality is not possible.

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 3.2):

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

Leakage: The leakage is as follows:

$$\begin{aligned} \mathbf{I} &= (\mathsf{Build}(X), \mathsf{Access}(\mathtt{read}, [s_1,t_1], \perp), \ldots, \mathsf{Access}(\mathtt{read}, [s_m,t_m], \perp)) \text{ , we define} \\ \mathsf{leakage}(\mathbf{I}) &:= (|X|, t_1 - s_1 + 1, \ldots, t_m - s_m + 1) \end{aligned}$$

A logical Range Tree. For simplicity, assume $n := |X|$ is a power of 2; if it is 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: $\mathsf{T}.\mathsf{Build}(X)$. 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).

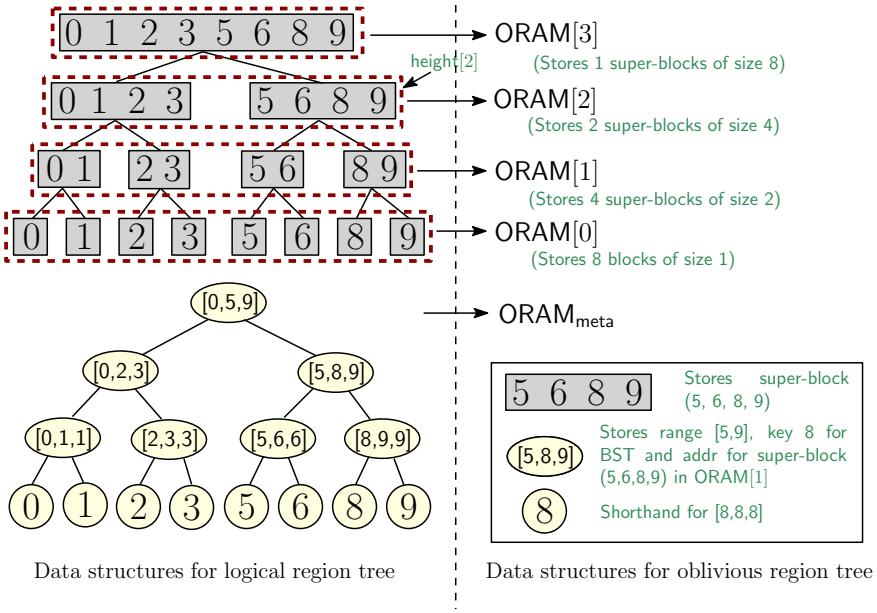

Figure 2: An oblivious Range Tree with Locality.

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 $\mathsf{H}_{\text{meta}}.\mathsf{Access}(\cdot)$ L 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 $\mathsf{B} = \mathsf{Dedup}(\mathsf{B}_1 \,||\, \mathsf{B}_2, \mathsf{len})$.

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* $n \cdot \mathsf{poly}\log(n+\lambda)$ *work and* $(2, \mathsf{poly}\log(n+\lambda))$ *locality,*

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

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

Obliviousness. We first claim the existence of adaptive simulators $\mathsf{Sim}_0, \ldots, \mathsf{Sim}_L$, where Sim_j corresponds to ORAM H_j. In addition, there exists a simulator $\mathsf{Sim}_{\mathsf{meta}}$, corresponding to $\mathsf{H}_{\mathsf{meta}}$, $\mathsf{Sim}_{\mathsf{Dedup}}$ for the algorithm Dedup, and $\mathsf{Sim}_{\mathsf{sort}}$ for the oblivious sorting algorithm. We construct a simulator for satisfying Definition 3.2 where the function $f, \mathsf{leakage}$ are as defined above.

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

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

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

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

- $\mathbf{Hyb_0(\lambda)}$: This is exactly the real execution. With each instruction I received from the adversary, we hand it to the real construction to receive the memory addresses. In addition, the construction interacts with the real memory and generates the output out_i in each stage, which is also given to the adversary.
- $\mathbf{Hyb_1(\lambda)}$: Same as $\mathsf{Hyb}_1(\lambda)$, where now the we use the Range Tree functionality in order to produce the output out_i in each step.
- $\mathbf{Hyb_{2,k}(\lambda)}$ with $k \in [L]$: In this execution, upon receiving some instruction I from the adversary, we proceed as follows:
  1. $\mathsf{Build}(X)$: Perform Steps 1–3 in Algorithm 5.1. Then,
     - For all $i \le k$, call to $\mathsf{Sim}_i(|X|/2^i)$ as in the simulation.
     - For all $i > k$, perform $\mathsf{H}_i.\mathsf{Build}(\mathsf{height}[i])$.
  2. $\mathsf{Access}(\mathtt{read}, [s, t], \perp)$: (with $t - s = 2^i$), perform the following steps:
     (a) Call $\mathsf{H}_{\mathsf{meta}}$ to obliviously search for the metadata $\mathsf{addr}_1, \mathsf{addr}_2$ as in the real execution.
     (b) If $i \le k$, call to the simulator Sim_i for simulating two accesses.
     (c) If $i > k$, then call to the real oblivious RAM H_i to access both addr_1 and addr_2.

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

- **$\mathsf{Hyb_3}(\lambda)$:** Same as $\mathsf{Hyb}_{2,L}(\lambda)$, except that the metadata ORAM is replaced with $\mathsf{Sim}_{\mathsf{meta}}$.
- **$\mathsf{Hyb_4}(\lambda)$:** Same as $\mathsf{Hyb}_3(\lambda)$ except that we replace Dedup with $\mathsf{Sim}_{\mathsf{Dedup}}$.
- **$\mathsf{Hyb_5}(\lambda)$:** Same as $\mathsf{Hyb}_4(\lambda)$ except that we replace the oblivious sort with $\mathsf{Sim}_{\mathsf{sort}}$. As a result, the experiment uses only the leakage of the instruction, and this is exactly the simulator Sim.

We show that for every adversary $\mathcal{A}$, its view in each one of the hybrid experiment is indistinguishable. Specifically, $\mathsf{Hyb}_5(\lambda)$ is indistinguishable from $\mathsf{Hyb}_4(\lambda)$ due to the security of the oblivious sorting algorithm. The view of the adversary in $\mathsf{Hyb}_4(\lambda)$ is indistinguishable from its view in $\mathsf{Hyb}_3(\lambda)$, due to the security of the Dedup function.

The view of the adversary $\mathsf{Hyb}_3(\lambda)$ is indistinguishable from $\mathsf{Hyb}_{2,L}(\lambda)$ due to the security of the metadata ORAM. In a more detail, assume by contradiction that there exists an adversary $\mathcal{A}$ that succeeds to distinguish between $\mathsf{Hyb}_3(\lambda)$ and $\mathsf{Hyb}_{2,L}(\lambda)$. We show the existence of an adversary $\mathcal{A}'$ that succeeds to distinguish between $\mathsf{Expt}_{\mathcal{A}'}^{\mathrm{real},M_{\mathsf{ORAM}}}(1^\lambda)$ and $\mathsf{Expt}_{\mathcal{A}',\mathsf{Sim}_{\mathsf{meta}}}^{\mathrm{ideal},f_{\mathsf{ORAM}}}(1^\lambda)$ as follows:

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

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

5.3 Range ORAM from Oblivious Range Tree

In this section, we show how to construct a Range ORAM from an oblivious Range Tree scheme. Since the underlying oblivious Range Tree has good efficiency/locality, so will the resulting Range ORAM. The idea behind our construction is similar to that of the standard hierarchical ORAM [25, 27]. 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., $\mathsf{all} := \cup_{j=0}^{i-1} \mathsf{T}_j$. This can be easily done by fetching its root. Each block in all is tagged with its level number j as a secondary key so that later after calling Dedup, 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 $\mathsf{all} = \mathsf{all} \cup \mathsf{T}_j.\mathsf{Access}(\mathtt{read}, [s,t], \perp)$.
3. Let $\mathsf{data}^* := \mathsf{Dedup}(\mathsf{all}, 2^i)$. If $\mathsf{op} = \mathtt{read}$, return data^* to client, Else, $\mathsf{data}^* := \mathsf{data}$.
4. If level i is marked empty, 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.

We prove the following Theorem:

Theorem 5.5 (Range ORAM with locality)**.** *Assuming one-way functions exist, there exists a computationally secure oblivious Range ORAM consuming $O(N \log N)$ space with $\mathsf{negl}(N)$ failure probability, and $\mathsf{len} \cdot \mathsf{poly}\log N$ work and $(2, \mathsf{poly}\log N)$ locality for accessing a range of size len.*

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

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

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

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

The simulator Sim**.** Upon initialization, initialize $L+1$ bits corresponding to whether a level is ready or empty. Mark all levels as empty, except for the last level. Invoke Sim_L with leakage 2^L.
Access: Upon receiving $\mathsf{leakage}(I)$ with $\mathsf{leakage}(I) = 2^i$ for some integer i

1. For $j = 0, \ldots, i-1$, access the memory locations devoted to Sim_j.
2. For $j = i, \ldots, L$, if the level j is marked ready, invoke the simulator Sim_j on simulating an access with leakage 2^i.

3. Invoke the simulator $\mathsf{Sim}_{\mathsf{Dedup}}(2^i)$.
4. If level i is marked empty, then invoke Sim_i with Build and leakage 2^i. Otherwise,
   (a) Let ℓ denote the smallest level greater than i that is empty. If no such level exists, let $\ell = L$.
   (b) Call $\mathsf{Sim}_{\mathsf{Dedup}}(2^\ell)$. Terminate all running simulators $\mathsf{Sim}_0, \ldots, \mathsf{Sim}_\ell$ and mark all corresponding bits as empty. Restart Sim_ℓ with Build on leakage 2^ℓ, and Sim_i with leakage 2^i, and mark corresponding bits as ready.

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

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

- $\mathsf{Hyb}_0(\lambda)$: This is exactly the real execution. Upon receiving instruction $I = (\mathsf{op}, [s,t], \mathsf{data})$ from the adversary, we invoke Algorithm 5.4 and output the memory addresses it produces, and out_i.
- $\mathsf{Hyb}_1(\lambda)$: Same as $\mathsf{Hyb}_0(\lambda)$ but the adversary receives in each step the output of the Range ORAM functionality and not the output of the construction. The memory addresses are still according to the construction.
- $\mathsf{Hyb}_{2,k}(\lambda)$ with $k \in \{0, \ldots, L\}$: In this hybrid, we replace all range trees $0, \ldots, k-1$ with simulators $\mathsf{Sim}_0, \ldots, \mathsf{Sim}_{k-1}$.
  Upon receiving some instruction $I = (\mathsf{op}, [s,t], \mathsf{data})$ with $t - s + 1 = 2^i$ for some integer i, we follow Algorithm 5.4. Whenever the algorithm performs $\mathsf{T}_j.\mathsf{Build}(X)$ for some $j < k$ and some X, we replace it with an invocation of Sim_j for Build instruction with leakage $|X|$. Whenever the Algorithm performs $\mathsf{T}_j.\mathsf{Access}$, we invoke Sim_j for Access with leakage 2^i.
- $\mathsf{Hyb}_3(\lambda)$: Same as $\mathsf{Hyb}_{2,L}(\lambda)$, where here also the Dedup algorithm is replaced with $\mathsf{Sim}_{\mathsf{Dedup}}$. As a result, we do not use any information in the instruction I beyond $\mathsf{leakage}(I)$, and this is exactly the simulator Sim.

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

5.4 Asymptotical Improvements and Stronger Security

Statistically secure, asymptotically faster Range ORAM. So far, 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. We will next present non-trivial techniques that asymptotically improve the range ORAM's work and locality by poly-logarithmic factors and moreover, achieve statistical security. In the interest of space, below we outline the core ideas and present the results as theorems while deferring the technical details to the appendices.

- *Locally initializable oblivious binary search trees.* Our first observation is that each oblivious range tree needs an oblivious binary search tree for metadata. Instead of using a generic

ORAM, we can employ a more efficient oblivious binary search tree (OBST) using oblivious data structure techniques described in earlier works [23, 38, 54]. For the same reason why we needed ORAMs to support locality-friendly initialization, our OBST must also be locally initializable. In Appendix B.1, we will describe how to extend an OBST to support locality-friendly initialization.

- *Tree-based ORAMs.* Tree-based ORAMs [45, 48, 52] provide an $O(\log N)$ factor work and locality improvement over best known hierarchical ORAMs. Meanwhile, they are also statistically secure. Similarly, we need to make tree-based ORAMs locally initializable, which we describe in Appendix B.1.
  Now that we have to perform an OBST search, we can insert another optimization for free. We additionally store in the OBST each (super-)block's position label in addition to its address in the corresponding height ORAM. A couple of earlier works have observed that in tree-based ORAMs, if correct position labels can be provided for free upon each query, we can asymptotically reduce the ORAM's bandwidth overhead by avoiding the need of ORAM recursion [23, 38, 54]. To capture this intuition, these works have coined the term "position-based ORAMs".

Putting it all together, we obtain asymptotically faster Range ORAM constructions with statistical security.

Corollary 5.6 (Statistically secure Range ORAM with locality)**.** *There exists a statistically secure Range ORAM such that except with* $\mathsf{negl}(N)$ *probability, accessing any contiguous memory range of size* len *requires* $O(\alpha\mathsf{len} \cdot \log^3 N \log\log^2 N)$ *work and* $(3, O(\log^3 N \log\log^2 N))$ *locality, where* N *denotes both the total memory size as well as the security parameter.*

Perfectly secure Range ORAM. Similarly, if we replace our building blocks with a perfectly secure, locally initializable ORAM and use the locality-friendly implementation of bitonic sort as our oblivious sorting candidate, we can obtain a perfectly secure Range ORAM scheme with polylogarithmic runtime blowup and polylogarithmic locality blowup. We describe how to obtain a perfectly secure, locally initializable ORAM in Appendix C by modifying the perfectly secure ORAM scheme by Damgård et al. [18].

This immediately gives rise to the following corollary:

Corollary 5.7 (Perfectly secure Range ORAM with locality)**.** *There exists a perfectly secure Range ORAM scheme such that accessing any contiguous memory range of size* len *requires* $O(\mathsf{len} \cdot \mathsf{poly}\log N)$ *work and* $(O(1), O(\mathsf{poly}\log N))$ *locality, where* N *denotes both the total memory size.*

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.

Online Range ORAM: problem definition. We now consider an *online* variant, where the memory requests arrive one by one just as in normal ORAM, and we wish to have an ORAM scheme that *preserves locality*, i.e., if the logical request sequence consists of l contiguous memory regions, then the ORAM's physical access sequence visits only $l \cdot \mathsf{poly}\log N$ contiguous regions. More specifically, suppose that the original program's logical request sequence satisfies $(1, l)$-locality, we

want that the ORAM's physical access sequence satisfies $(O(1), l \cdot \mathsf{poly}\log N)$-locality. Similar as before, we want that an online Range ORAM to have only polylogarithmic work.

The security of online range ORAM can be defined in the same manner as range ORAM, where the adaptive leakage function $\mathsf{leakage}(\mathbf{I})$ outputs a bit, indicating whether the last request in $\mathbf{I}$ is contiguous w.r.t. to the last but second.

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. 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. Thus we have the following theorem.

Theorem 5.8 (Online Range ORAM)**.** *Assuming one-way functions exist, there exists a computationally secure online Range ORAM with* $\mathsf{negl}(N)$ *failure probability, which on receiving* len *consecutive memory locations online performs* $\mathsf{len} \cdot \mathsf{poly}\log N$ *work and achieves* $(O(1), \mathsf{poly}\log N)$ *locality.*

6 File ORAM

In this section, we show how to construct a File ORAM scheme. We first define it in Section 6.1. In Section 6.2, we construct a non-recurrent read-only File hashing primitive, which is a File ORAM that supports only read accesses and achieves obliviousness if no file is accessed more than once. Finally, we show how to rely on core ideas behind the hierarchical ORAM framework to obtain a full-fledged File ORAM from non-recurrent read-only File Hashing schemes (see Section 6.3).

6.1 Definition

Compared to a Range ORAM which supports accesses to any contiguous memory region, a File ORAM provides a more constrained functionality that supports accesses to a set of files of predefined

ranges. More specifically, in Range ORAM every memory range $[s, t]$ $(t > s)$ can be accessed, while in File ORAM, each file is the atomic unit of access. A File ORAM should leak only the length of the file requested, but should hide any additional information, such as the number of files of each size, and whether the same file is requested twice or two files of the same size are requested. The File ORAM functionality is defined as follows, and its oblivious simulation is defined using Definition 3.2 where the allowed leakage is as follows:

Functionality: The requests are in the form $\mathsf{Access}(\mathsf{op}, \mathsf{fid}, \mathsf{data})$ where $\mathsf{op} \in \{\texttt{read}, \texttt{write}\}$, $\mathsf{fid} \in [k]$ and $\mathsf{data} \in (\{0,1\}^b)^{\mathsf{len}(\mathsf{fid})}$. If $\mathsf{op} = \texttt{read}$, then the content of file fid is returned. If file fid does not exist (has not been written before), $\perp$ is returned. If $\mathsf{op} = \texttt{write}$, then data denotes the updated value of file fid.

Leakage: Let $\mathsf{len}(\mathsf{fid})$ be the size of file fid. For simplicity, we assume the size of each file $\mathsf{len}(\mathsf{fid})$ is a power of 2 (otherwise, simply pad each file to the next power of 2). Let N be the sum of sizes of all k files. The leakage is simply the length of each accessed file. Formally, given a sequence of instructions

$$\begin{aligned} \mathbf{I} &= ((\mathsf{op}_1, \mathsf{fid}_1, \mathsf{data}_1), \ldots, (\mathsf{op}_m, \mathsf{fid}_m, \mathsf{data}_m)) \text{ , we define} \\ \mathsf{leakage}(\mathbf{I}) &:= (N, \mathsf{len}(\mathsf{fid}_1), \ldots, \mathsf{len}(\mathsf{fid}_m)) \text{ .} \end{aligned}$$

In particular, File ORAM does not reveal how many files there are of each size, and does not reveal whether we access the same file twice or two different files of the same size.

6.2 Non-Recurrent File Hashing Scheme with Locality

A File Hashing scheme T builds a hash table when given a set of files, such that later, file read request can be served quickly. We consider a relaxed notion of obliviousness, where obliviousness is only guaranteed when the adversary is constrained to access requests where each (non-dummy) file can be requested only once.

Functionality: The functionality supports the following instructions:

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

Leakage. The allowed leakage is as follows. For a sequence

$$\begin{aligned} \mathbf{I} &= (\mathsf{Build}(X), \mathsf{Access}(\texttt{read}, \mathsf{fid}_1, \perp, \mathsf{len}_1), \ldots, \mathsf{Access}(\texttt{read}, \mathsf{fid}_m, \perp, \mathsf{len}_m)) \text{ , we define} \\ \mathsf{leakage}(\mathbf{I}) &:= (|X|, \mathsf{len}_1, \ldots, \mathsf{len}_m) \text{ .} \end{aligned}$$

Obliviousness under non-recurrent requests. We say that a non-recurrent File Hashing scheme satisfies obliviousness, iff Definition 3.2 is respected when the adversary $\mathcal{A}$ is constrained to submit request sequences *where the same file identifier* $\mathsf{fid} \neq \perp$ *is never requested twice.*

Construction. Below, we present our non-recurrent File hashing scheme construction by extending the standard balls-and-bins hashing [6]. The output of $\mathsf{Build}(X)$ is a PRF key k and an array of size $2|X|$.

Algorithm 6.1: $\mathsf{T.Build}(X)$

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

Algorithm 6.2: $\mathsf{T.Access}(\mathtt{read}, \mathsf{fid}, \perp, \mathsf{len})$

1. If $\mathsf{fid} \neq \perp$, compute the starting bin number $g := \mathsf{PRF}_k(\mathsf{fid})$. Else, choose g uniformly at random from $[0, \ldots, B-1]$.
2. Retrieve all blocks from bins $g, g+1, \ldots, g+\mathsf{len}-1$ (each mod B). Let all denote the concatenated blocks.
3. In a single linear scan, mark all blocks whose file identifiers are not fid with $\perp$. Use an oblivious sort to remove these blocks and truncate the array length to len.

We then obtain the following theorem:

Theorem 6.3 (Oblivious non-recurrent File hashing scheme)**.** *Let n be an upper bound on the number of blocks of all files combined. Assuming one-way functions exist, for any super constant function $\alpha := \omega(1)$, there exists a computationally secure non-recurrent File hashing scheme that requires $O(n)$ space, and except with $\mathsf{negl}(\lambda)$ probability*

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

Proof. Correctness is clear. We first show obliviousness, then proceed to efficiency and locality and end up with a claim regarding overflow analysis.

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

- Upon receiving a leakage $|X|$ for $\mathsf{Build}(X)$, the simulator performs few linear scans of the memory and invoke the simulator of the underlying oblivious sort whenever necessary.
- Upon receiving a leakage len for simulating some $\mathsf{T.Access}(\mathtt{read}, \mathsf{fid}, \perp, \mathsf{len})$ instruction, the simulator chooses a random bin $g \in [B]$, and accesses $g, g+1, \ldots, g+\mathsf{len}-1$ (modulo B).

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

- Instruction $\mathsf{T.Build}(X)$ is clearly the same in both executions.
- Upon instruction $\mathsf{T.Access}(\mathtt{read}, \mathsf{fid}, \bot, \mathsf{len})$ with $\mathsf{fid}_i = \bot$, then Algorithm 6.2 chooses a random bin $g \in [B]$ and access the bins $g, \ldots, g_{\mathsf{len}-1}$ (modulo B). This is exactly as the simulation.
- Upon instruction $\mathsf{T.Access}(\mathtt{read}, \mathsf{fid}, \bot, \mathsf{len})$ with $\mathsf{fid}_i \neq \bot$, the non-recurrence property guarantees that there was no previous access to this fid. Algorithm 6.2 computes $g = \mathsf{PRF}_k(\mathsf{fid})$ and accesses bins $g, \ldots, g_{\mathsf{len}-1}$ (module B), whereas the simulator chooses g at random. Assuming the pseudorandom property of the PRF, and relying on the fact that fid was not queried before, these two distributions are the same.

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

Claim 6.4 (Overflow analysis)**.** *According to the assignment used in* Build*, the probability of* $\mathsf{overflow}$ *within each bin is negligible in* λ*.*

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

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

Moreover, the random variables $Y_\beta[1], \ldots, Y_\beta[k]$ are *independent* and are taking values in $\{0, 1\}$. By Chernoff's bound we have that the probability to exceed Z (and output $\mathsf{overflow}$) is negligible in λ, and the claim is obtained by a simple union bound on the number of bins B. The analysis can also be adapted for the case where we have a file with size $n_i > B$. In that case, each bin receives at least $\lfloor n_i/B \rfloor$ real elements of that file, and random $[n_i \bmod B]$ consecutive bins receive one more element in addition, and therefore this case is reduced to the case where all files are of size $< B$ (see also [6]). □

This complete the proof of Theorem 6.3. □

6.3 Constructing File ORAM from Non-Recurrent File Hashing Scheme

In this section, we show how to construct a File ORAM with locality in linear space, in a blackbox manner from non-recurrent file hashing scheme. Below we use N to denote both the total size of logical memory as well as the security parameter.

Data structure. There are $\log N + 1$ *levels* numbered $0, 1, \ldots, L$ where $L := \lceil \log_2 N \rceil$ is the largest level. Each level is a non-recurrent file hashing scheme and its data structure is denoted $\mathsf{H} := (\mathsf{H}_0, \mathsf{H}_1, \ldots, \mathsf{H}_L)$ where H_i has capacity 2^i. At any time, each table H_i can be in two possible states, *ready* or *empty*.

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

Algorithm 6.5: $\mathsf{H}.\mathsf{Access}(\mathsf{op}, \mathsf{fid}, \mathsf{data}, \mathsf{len})$**,** where $\mathsf{len} = 2^i$ for some non-negative integer i:

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

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

Theorem 6.6 (File ORAM in linear space with locality)**.** *Assume that one-way functions exist. There exists a computationally secure File ORAM scheme that achieves* $O(\mathsf{len} \cdot \log^2 N \log\log^2 N)$ *work and* $(3, O(\log N \log\log^2 N))$ *locality for accessing a file of* len*.*

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

Amortized work for access. Each access of a file of size len involves looking up in all levels $\log \mathsf{len}, \ldots, \log N$, each costing $O(\alpha \cdot \mathsf{len} \cdot \log N)$. Thus, the cost to retrieve a file of size len is $O(\alpha \cdot \mathsf{len} \cdot \log^2 N)$.

For rebuild, we can view the concatenations of bits indicating whether a level is empty or ready as binary counter. When accessing a file with size $\mathsf{len} = 2^i$, we perform the following rebuild:

- If level i is empty (i.e., the i-th position in the counter is 0), we just rebuild level i (and the i-th position in the counter is switched to 1, equivalent to incrementing it by 2^i).
- Otherwise, we build some level $\ell > i$ from the context of the levels $0, \ldots, \ell - 1$ and the returned file, and these levels are full. Specifically, the counter is 1 in positions $i, \ldots, \ell - 1$, and 0 in position ℓ. By incrementing the counter by 2^i, all positions $i, \ldots, \ell - 1$ becomes 0, and position ℓ becomes 1. Thus, this is equivalent to increment the counter by 2^i.
  At this point, level i is empty, and we rebuild it with the data D. This is equivalent to incrementing the counter by another 2^i.

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

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

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

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

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

The simulator Sim. The simulator for the File ORAM initializes $L+1$ bits representing whether the levels are ready or empty. Upon initialization, all the bits are empty, except for the level L which is marked as ready, and it also invokes the simulator Sim_L on Build instruction with input 2^L.

$\mathsf{Simulating\ Access}(\mathsf{op}, \mathsf{fid}, \mathsf{data}, \mathsf{len})$, using only leakage $\mathsf{len} = 2^i$ where $\mathsf{len} = \mathsf{len}(\mathsf{fid})$:

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

The internal state of the simulator is all its internal bits, and the internal states of the underlying simulators

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

- $\mathbf{Hyb_0(\lambda)}$: This is exactly the real execution. That is, upon receiving instruction $I_i = \mathsf{Access}(\mathsf{op}, \mathsf{fid}, \mathsf{data}, \mathsf{len})$ from the adversary, we invoke Algorithm 6.5 and output the memory addresses it produces, and the outputs out_i in each step.
- $\mathbf{Hyb_1(\lambda)}$: This is as before where the out_i is now being computed by the File ORAM functionality.
- $\mathbf{Hyb_{2,k}(\lambda)}$ with $k \in [L]$: Same as $\mathsf{Hyb}_1(\lambda)$, where we replace all non-recurrent file hashing scheme, $\mathsf{T}_0, \ldots, \mathsf{T}_{k-1}$ with simulators $\mathsf{Sim}_0, \ldots, \mathsf{Sim}_{k-1}$.
  Upon receiving some instruction $I = \mathsf{Access}(\mathsf{op}, \mathsf{fid}, \mathsf{data}, \mathsf{len})$, we follow Algorithm 6.5. Whenever the Algorithm performs $\mathsf{T}_j.\mathsf{Build}(X)$ for some $j < k$ and some X, we replace it with an invocation of Sim_j for Build instruction with leakage $|X|$. Whenever the Algorithm performs $\mathsf{T}_j.\mathsf{Access}$ for $j < k$, we invoke Sim_j for Access with leakage len.
- $\mathbf{Hyb_3(\lambda)}$**:** Same as $\mathsf{Hyb}_{2,L}(\lambda)$ where the only difference is as follows: whenever calling to $\mathsf{Dedup}(S, 2^t)$ for some integer t and set S, we replace it with the $\mathsf{Sim}_{\mathsf{Dedup}}(2^t)$.
- $\mathbf{Hyb_4(\lambda)}$**:** Same as $\mathsf{Hyb}_3(\lambda)$ where every invocation of $\mathsf{H}_{\mathsf{meta}}$ is replaced with an access to $\mathsf{Sim}_{\mathsf{meta}}$. As such, the only information that we use in the $I = \mathsf{Access}(\mathsf{op}, \mathsf{fid}, \mathsf{data}, \mathsf{len})$ is the value len. This is exactly the simulation execution.

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

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

It is easy to see that $\mathsf{Hyb}_{2,0}(\lambda)$ is identical to $\mathsf{Hyb}_1(\lambda)$, which is the real execution. Finally, $\mathsf{Hyb}_1(\lambda)$ and $\mathsf{Hyb}_0(\lambda)$ are indistinguishable due to the correctness of the construction. □

7 A New Oblivious Sorting Algorithm with Locality

In this section, we will show how to perform locality-friendly oblivious sorting with $O(n \log n \log\log^2 \lambda)$ work and $(3, O(\log n \log\log^2 \lambda)$ locality, where λ is the security parameter.

Intuitively speaking, we observe that the composition of the following two algorithms yields an oblivious sorting algorithm:

- First, we *obliviously* permute the input array at random;
- Second, we apply a *non-oblivious*, comparison-based sorting algorithm on the permuted array.

We show how to implement each one of the steps efficiently and with good data locality. In fact, we do not fully implement an oblivious permutation with good locality, and instead we implement a weaker primitive such that the composition still works and results in an oblivious algorithm. In particular, we implement an "oblivious random bin assignment" algorithm defined in Section 7.1. Then, in Section 7.2 we prove the composition the theorem, and conclude our locality-friendly oblivious sorting algorithm.

7.1 Oblivious Random Bin Assignment

7.1.1 The Functionality

We start with defining the oblivious random bin assignment functionality, which we denote by f_{randbin}. In a nutshell, given some input array A we consider an output array which as twice the size of the input array, and we consider the output array as B consecutive bins. We assign each "real" element of the input array into a random bin in the output array, and pad each bin with dummy elements. We then show how to implement this functionality obliviously.

Random bin assignment.

- **Functionality:** Consider random bin assignment functionality $f_{\text{randbin}}(X, Z)$:
  - Upon receiving an array X of length n, and a bin size Z, choose a random bin for each element of X among a total of $B := 2n/Z$ bins. If some bin receives more than Z elements, abort.
  - Output an array Y of size $B \cdot Z$ representing the B output bins, where each bin contains its assigned elements padded with dummy elements such that 1) all real elements appear before dummies; and 2) real elements are ordered in each bin.
- **Leakage:** $|X|, Z$.

Henceforth, we say that an algorithm M is an oblivious random bin assignment algorithm, if it obliviously simulates f_{randbin} with $\mathsf{negl}(\lambda)$ statistical failure. As the functionality is randomized, we use Definition 3.1.

7.1.2 A Locality-Friendly Random Bin Assignment Algorithm

We now describe a locality-friendly oblivious $\mathsf{RandomBinAssignment}$ procedure, inspired by the Bucket ORAM algorithm [20]. The algorithm combines ideas from Bucket sort and Radix sort [16].

The algorithm first chooses a bucket size Z, and constructs $B = \lceil 2|X|/Z \rceil$ buckets each of size Z. Each element in X is then assigned a random key $\mathsf{key}' \in [0, B-1]$ which represents a destination bucket. Next, the algorithm exchanges elements between bucket pairs in $\log B$ steps to distribute elements into their destination buckets, using the $\mathsf{MergeSplit}$ operation. In a more detail, the operation $(C_0, C_1) \leftarrow \mathsf{MergeSplit}(A_0, A_1, i)$ works on four buckets at the time, distributing the keys of the two input buckets A_0, A_1 into two output buckets C_0, C_1 according to the $i+1$th most significant bit (MSB) of the keys (where C_0 receives all the keys with $i+1$th MSB as 0, and C_1 receives all the keys with $i+1$th MSB as 1). Each bucket in each step contains some dummy elements that serve as a form of "slack" which will be important later for our stochastic analysis to work.

We now describe our algorithm. Without loss of generality, assume that B is a power of 2. We also assume that elements in the input array X are all distinct — if not, we can use the indices of the elements within X to break ties.

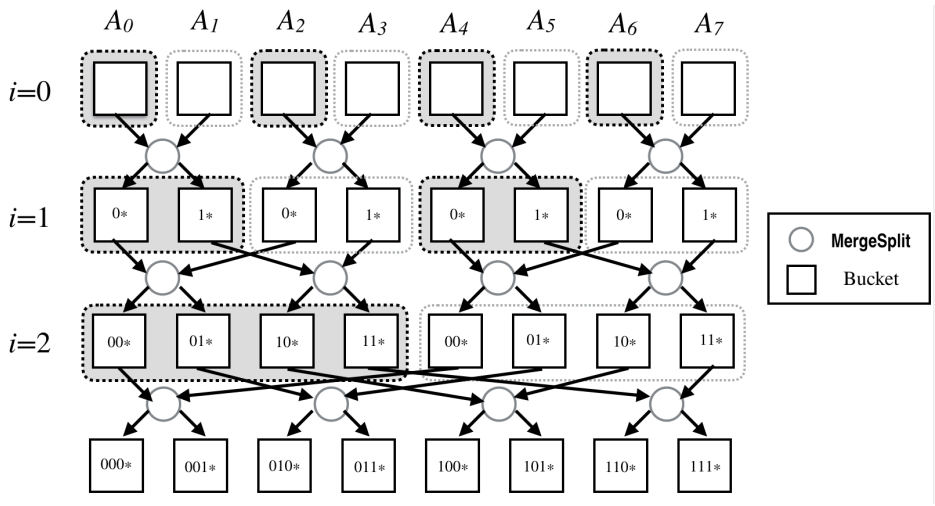

Figure 3: Radix-Bucket Shuffle for 8 input buckets. The MergeSplit procedure works on four buckets at a time, merging two buckets from level i and splitting them into two buckets of level $i+1$ according to i-th most significant bit of the keys, while preserving the order otherwise. At the end of each level i, each subarray (2^{i+1} consecutive buckets) are semi-sorted according to the i most significant bits of the keys, where the jth bucket of the subarray (for $j = 0, \ldots, 2^{i+1} - 1$), contains all keys with binary prefix $\langle j \rangle$.

Algorithm 7.1: RandomBinAssignment(X)

- Let X be an input array of size n, consisting real elements. For each element $X[i]$ in the array, assign a uniformly random key $\mathsf{key}' \in [0, B-1]$.
- Choose a bucket size $Z := \alpha \log \lambda$ with $\alpha \in \omega(1)$. Define an array A of size $2n$, consisting of $B = 2n/Z$ buckets each of size Z, denoted as $A_0, A_1, \ldots, A_{B-1}$.
- Initialize A_i to contain the i-th consecutive $Z/2$ elements of X and $Z/2$ dummy elements.
- For step $i = 0, \ldots, \log B - 1$:
  For $j = 0, \ldots, B/2^{i+1} - 1$:
  For $k = 0, \ldots, 2^i - 1$:
  $(A_{j \cdot 2^{i+1}+k}, A_{j \cdot 2^{i+1}+k+2^i}) \leftarrow \mathsf{MergeSplit}(A_{j \cdot 2^{i+1}+k}, A_{j \cdot 2^{i+1}+k+2^i}, i)$

Figure 3 demonstrates the structure of the algorithm for 8 input buckets.

The MergeSplit *subroutine.* The algorithm repeatedly calls the basic operation MergeSplit, which exchanges real elements between two buckets. It additionally takes in the step number i.

Procedure 7.2: $(A'_0, A'_1) \leftarrow \mathsf{MergeSplit}(A_0, A_1, i)$

1. A'_0 receives all real elements in $A_0 \cup A_1$ where the $(i+1)$-st MSB of the key is 0.
2. A'_1 receives all real elements in $A_0 \cup A_1$ where the $(i+1)$-st MSB of the key is 1.
3. If either output bucket receives more than Z real elements, then the procedure aborts with overflow.
4. Otherwise, pad output buckets A'_0, A'_1 to size Z with dummy elements, and sort each output bucket such that 1) real elements appear before dummies, and 2) real elements are ordered in each bucket.

It is not hard to see that MergeSplit can be realized with a single invocation of bitonic sort,

which has $O(Z \log^2 Z)$ work and $(2, \log^2 Z)$ locality. More concretely, we first scan the two buckets to count how many elements are supposed to go to A'_0 and A'_1, respectively, then tag the right number of dummy elements to go to either direction, and finally perform the bitonic sort. Using the concurrent bitonic sort procedure in Appendix A, we can realize all $B/2$ concurrent instances of MergeSplit in $O(BZ \log^2 Z)$ work and with locality $(2, O(\log^2 Z))$. Note that the MergeSplit operation may be implemented in place.

We now analyze the overflow probability, and show that the probability of overflow is negligible in λ. Formally, in we prove the following claim:

Claim 7.3 (Overflow probability)**.** *There is no* overflow *during the execution of the algorithm, except with probability at most* $B \cdot \log B \cdot \mathsf{negl}(\lambda)$*,*

Proof. Let T be the set of real elements. For $t \in T$, $i \in \{1, \ldots, \log B\}$ and $b \in [B]$, let $X^t_{i,b}$ be an indicator that receives 1 if the real element t is assigned to the bth bucket at the end of the $(i-1)$-st iteration. Let $Y_{i,b}$ denote the load of the bucket (b, i). It holds that $Y_{i,b} = \sum_{t \in T} X^t_{i,b}$, and we bound the probability that a bucket exceeds its limit.

The algorithm ensures that each bucket contains at most $Z/2$ real elements. Fix i and a bucket b as above. Observe that this bucket can receive real elements potentially from 2^i initial buckets, each of which contains at most $Z/2$ real elements initially. Hence, potentially at most $2^i \cdot Z/2$ real elements can be assigned to that bucket.

For each such element, we choose a new key uniformly at random. The probability that the chosen key corresponds to the bucket b is exactly 2^{-i}. As a result, for at most $2^i \cdot Z/2$ of the random indicators $\{X^t_{i,b}\}_t$, the probability to be 1 is 2^{-i}, whereas all the rest of the indicators we have probability 0 to receive 1, and the probabilities are independent since the keys are independent. This implies that the expected load of each bucket is $Z/2$ and this load corresponds to $\sum_{t \in T} X^t_{i,b}$. Chernoff's bound implies that the probability that after some fixed iteration, a particular fixed bucket overflows with probability at most $e^{-Z/4} = \mathsf{negl}(\lambda)$. Hence, a union bound over all iterations and all buckets implies that except with $B \log B \cdot \mathsf{negl}(\lambda)$ probability, no bucket overflows throughout the algorithm. □

Lemma 7.4. *Let* $Z = \alpha \log \lambda$ *for any super-constant function* $\alpha(\lambda)$*. Algorithm 7.1 obliviously simulates* f_{randbin} *with* $\mathsf{negl}(\lambda)$ *statistical failure.*

Proof. The random string consumed by the real-world algorithm RandomBinAssignment includes a random bin number assigned to each element. There are $\mathsf{negl}(\lambda)$ fraction of them that cause RandomBinAssignment to overflow by Claim 7.3. Similarly, except with $\mathsf{negl}(\lambda)$ probability, the random assignment sampled by the ideal f_{randbin} will not lead to overflow.

Consider all random strings that do not lead to overflow in the real world or in the ideal world. Then it is obvious that the real-world algorithm correctly places each element into the targeted bin just like what f_{randbin} does. Further, the access patterns of the real-world algorithm is deterministic and independent of the input and the random choices of the algorithm. Thus the lemma follows in a straightforward manner from here. □

We can now analyze the efficiency and locality of the construction assuming no overflow. There are $\log B$ iterations, and in each iteration, we have $B/2$ executions of MergeSplit procedure. This results in a total of $O(B \log B)$ executions of the MergeSplit procedure, which translates to $O(B \log B \cdot Z \log^2 Z)$ work and locality $(3, O(\log B \cdot \log^2 Z))$. Putting $B = 2n/Z$ and $Z = \alpha \log \lambda$ with $\alpha \in \omega(1)$, this results in $O(n(\alpha \log \lambda + \log n) \log \log^2 \lambda)$ work with $(3, O(\log n \log \log^2 \lambda))$ locality.

With this, we can conclude with the following theorem.

Theorem 7.5 (Oblivious random bin assignment)**.** *There exists an oblivious random bin assignment algorithm which, except with* $\mathsf{negl}(\lambda)$ *probability, completes in* $O(n \log n \log\log^2 \lambda)$ *work and with* $(3, O(\log n \log\log^2 \lambda))$ *locality, as long as* $n = \omega(\log \lambda)$*.*

7.2 The Oblivious Sort

Oblivious sort from random bin assignment algorithm and non-oblivious sort. Given a random bin assignment algorithm $\mathsf{ObliviousRandomBin}$ and a non-oblivious comparison-based sorting algorithm $\mathsf{NonObliviousSort}$ (e.g., Merge-Sort), one can easily construct an oblivious sorting algorithm as follows.

1. Given an input array X, invoke the random bin assignment algorithm to receive $Y := \mathsf{ObliviousRandomBin}(X)$. Note that in each "bin" of Y, the real elements appear before the dummy elements, and are sorted.
2. Now, sort Y using a (non-oblivious) comparison-based sorting algorithm. That is, invoke $W := \mathsf{NonObliviousSort}(Y)$, while preferring real elements over dummy elements. Formally speaking, a sorting algorithm is comparison-based if the physical access pattern depends only on the relative ranking of elements in the input. A technical condition we need is that no two elements have the same rank. This can be ensured by resolving any tie consistently by the initial location of the elements in the array Y.
3. All dummy elements appear at the n last locations of W. Truncate the last n elements of W.

If both the random bin assignment $\mathsf{RandomBin}$ and the non-oblivious sorting algorithm has good data locality, then so will the resulting oblivious sorting algorithm. We use Merge-Sort as the non-oblivious sorting candidate. Let $\mathsf{Truncate}$ denote the last step of the above algorithm. Then, our sorting algorithm can be described as the following simple composition

$$\mathsf{Truncate}(\mathsf{NonObliviousSort}(\mathsf{ObliviousRandomBin}(X)))$$

Why is it oblivious? A natural question is why these composition is still oblivious, as we do not fully permute the input array. Essentially, the compositions still holds since there is a 1-to-1 mapping between any input of the array and every possible output of the f_{randbin} functionality, i.e., every possible input of the non-oblivious sorting algorithm. This mapping is exactly the random assignment of the destination bins. Therefore, every access pattern in the non-oblivious sorting part of the algorithm, can be justified with some specific random assignment on the input array, for every possible ordering of the input array. However, some of the assignments are "invalid" due to overflows; however, overflows occurs with negligible probability, and therefore we get statistical security. We proceed to the formal proof of the security, which is even simpler than the above argument.

Lemma 7.6 (From oblivious random bin to oblivious sort)**.** *Suppose that* $\mathsf{ObliviousRandomBin}$ *is a statistically (or perfectly resp.) oblivious random bin assignment algorithm. Then, the above algorithm is a statistically (or perfectly resp.) secure oblivious sorting algorithm.*

Proof. Correctness of the algorithm is trivial. We next prove the obliviousness of the algorithm. We prove for the case of perfect security, since statistical security is similar, except that we replace "identically distributed" with "statistically close".

Let $\mathsf{SimRandBin}$ be the simulator algorithm for the underlying oblivious random bin assignment. Consider any given input X of length n, and let $(Y, \mathsf{addressesRandBin})$ denote the joint distribution of the outcome array Y and the addresses accessed during an execution of the oblivious random bin assignment. We have that

$$(Y, \mathsf{addressesRandBin}) \equiv (f_{\text{randbin}}(X), \mathsf{SimRandBin}(1^n, |X|)) \ .$$

Let $\mathsf{SortAddresses}(Y)$ denote the addresses observed during an execution of the truncation algorithm and the non-oblivious, comparison-based sorting algorithm upon receiving input array Y. We thus have that

$$(\mathsf{SortAddresses}(Y), \mathsf{addressesRandBin}) \equiv (\mathsf{SortAddresses}(f_{\text{randbin}}(X)), \mathsf{SimRandBin}(1^n, |X|)) \ .$$

For any comparison-based sort, its access patterns depend only on the relative ranking of the input elements. Since without loss of generality, we assumed that the input array X always has distinct values, and we carefully defined how to avoid any ties, we have that

$$\mathsf{SortAddresses}(f_{\text{randbin}}(X)) \equiv \mathsf{SortAddresses}(f_{\text{randbin}}([|X|])) \ ,$$

where $[|X|] = \{1, \ldots, |X|\}$. Now, observe that we can construct a simulator $\mathsf{SimObliviousSort}$ that simply outputs

$$\mathsf{SortAddresses}(f_{\text{randbin}}([|X|])), \mathsf{SimRandBin}(1^n, |X|)$$

This simulator's output is identically distributed as the real-world access patterns of executing the aforementioned sorting algorithm. □

The above performance and security analysis gives rise to the following theorem and corollary which are the main results of this section.

Corollary 7.7 (Locality-friendly oblivious sort)**.** *There exists a statistically secure oblivious sort algorithm which, except with* $\mathsf{negl}(\lambda)$ *probability, completes in* $O(n \log n \log\log^2 \lambda)$ *work and with* $(3, O(\log n \log\log^2 \lambda))$ *locality, as long as* $n = \log^3 \lambda$.

8 Lower Bound for More Restricted Leakage

In the introduction, we mentioned that an ORAM scheme cannot preserve locality without leaking the sizes of the accessed ranges. In this section, we formalize this claim.

Recall that in Section 5, we define the leakage of a sequence of instructions $I = ((\mathsf{op}_1, [s_1, t_1], \mathsf{data}_1), \ldots, (\mathsf{op}_T, [s_T, t_T], \mathsf{data}_T))$ to include the number N of logical blocks being considered, as well as the length $t_i - s_i + 1$ associated with each operation (which means the number T of operations is leaked too). In this section, we show that if we restrict the leakage and do not allow the adversary to learn the length of each operation or even the number of operations, the lower bound for bandwidth overhead 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 sum of the lengths of the operations, i.e, $S = \sum_{i=1}^{T}(t_i - s_i + 1)$. Formally, the distribution of physical accesses observed by the adversary is the same as long as N and S are fixed.

2. Just like earlier ORAM lower bounds [8,25,27]), 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. For simplicity, we shall first assume that the CPU has only 1 block of local cache — however, later in Section 8.3, we show how to remove this assumption in our lower bound proof.
4. We assume that the algorithm is partially online, i.e., there is a phase where one is allowed to preprocess memory without seeing future requests; however, after the preprocessing, all request come in all at once, and the ORAM algorithm is allowed to see all request upfront before serving them.

Notation. Recall that we use h to denote the number of concurrent heads, l to denote the locality blowup, m to denote the memory size blowup (i.e., the ORAM consumes mN memory for N logical memory), and β to denote the bandwidth blowup. We shall show statements of the following form for different choices of the parameters:

Any Range ORAM satisfying the restricted leakage that has (h, l)-locality and m memory blowup will incur β bandwidth blowup.

8.1 Warmup: Special Case of $h = l = 1$

We first prove this special case where $h = l$ are exactly 1 (and not even $O(1)$).

Lemma 8.1. *For any $\epsilon > 0$, for sufficiently large N, any ORAM satisfying the restricted leakage that has $(1, 1)$-locality, a single block of CPU cache, and memory blowup of $m = \mathsf{poly} \log N$ will incur bandwidth blowup of $\Omega(N^{1-\epsilon})$.*

Proof. We assume that before seeing the requests, the algorithm can arrange the N blocks in the memory in any order. Suppose we fix some T (say $T = \log N$ is enough). We consider the following two request scenarios.

1. A single request operation consisting of T contiguous blocks.
2. T request operations, each of which consists of a single block, where the T blocks are distinct.

Because of $(1, 1)$-locality, under the first scenario, the physical access pattern must be a linear scan of one contiguous region, which has length at most βT, because the bandwidth blowup is at most β . Moreover, because of restricted leakage, the same access pattern must also be observed under the second scenario.

We first count that given a particular access pattern of βT contiguous blocks, how many scenarios of the second type it can support. Since the CPU has memory 1, a sequential scan of βT blocks can support at most $\binom{\beta T}{T}$ scenarios of the second type. Since the memory blowup is $m = \mathsf{poly} \log N$, there are at most mN starting positions for the sequential scan. Hence, a very loose upper bound on the number of scenarios of the second type that can be supported is $mN \cdot \binom{\beta T}{T} \leq mN \cdot (\beta e)^T$, where e is the natural number.

On the other hand, the number of scenarios of type 2 is at least $\binom{N}{T} \geq (\frac{N}{T})^T$.

Hence, relating the two bounds, we must have $mN \cdot (\beta e)^T \geq (\frac{N}{T})^T$, which implies that $\beta \geq \Omega(N^{1-\epsilon})$, for sufficiently large N. □

8.2 General $h = l$

We next consider general choices of $h = l$ and $m \leq N$. Specifically, we prove the following.

Lemma 8.2. *For any $\epsilon > 0$, for sufficiently large N, any ORAM satisfying the restricted leakage that has (h, l)-locality with $h = l$, a single block of CPU cache, and memory blowup of $m \leq N$ will incur bandwidth blowup of β such that $\beta l \geq \Omega(N^{1-\epsilon})$.*

Proof. We follow the same proof strategy as in Lemma 8.1. We choose $T = l \log N$, and consider the scenarios of the following types.

1. A single request operation consisting of T contiguous blocks.
2. T request operations, each of which consists of a single block, where the T blocks are *not necessarily* distinct.

Consider the first scenario. Because of (h, l)-locality and β bandwidth, the physical access pattern must satisfy the following: (i) linear scans of at most l (not necessarily disjoint) contiguous blocks, and (ii) the sum of the lengths of the contiguous blocks is at most βT.

Upper bound on the number of type 2 scenarios supported. We next give an upper bound on the number of type 2 scenarios that can be supported by a physical access pattern that can be produced by a type 1 scenario. Since the memory blow up is $m = \mathsf{poly} \log N$, there are at most $(mN)^{2l}$ ways to specify l contiguous regions $(S_1, S_2, \ldots, S_l)$.

For each fixed choice of $(S_1, S_2, \ldots, S_l)$, we know that the sum of their lengths is at most βT. Hence, there are at most $\sum_{i=1}^{T} \binom{\beta T}{i} \leq (\beta e)^T$ ways to specify at most T blocks from the l contiguous regions.

Finally, since there are $h = l$ heads, the T blocks chosen from the l contiguous regions can be interleaved to satisfy different request scenarios. Observe that since each contiguous block is scanned linearly and the CPU has storage of size 1, this means the order of blocks from the same contiguous region must be preserved in the request sequence. Note that there are l contiguous regions, and the number of request operations is T. Hence, for each of the T operations, we can choose from which of the l regions to read the next block, and there is a choice of whether the same block will be requested in future operations. Therefore, there are at most $(2l)^T$ ways to interleave at most T chosen blocks from the l regions.

Hence, a loose upper bound on the number of type 2 scenarios that can be supported is: $(mN)^{2l} \cdot (\beta e)^T \cdot (2l)^T$.

Observe that the T blocks in type 2 scenarios need not be distinct. Hence, the number of type 2 scenarios is at least N^T. Hence, combining the two bounds and observing that $m \leq N$, we have $(mN)^{2l} \cdot (\beta e)^T \cdot (2l)^T \geq N^T$, which implies that $\beta l \geq \Omega(N^{1-\epsilon})$, for sufficiently large N. □

8.3 More General CPU Cache Size

So far, we have focused on the scenario where the CPU can store only a single block. We now explain how to generalize our lower bound for more general CPU cache size. Henceforth, let r denote the number of blocks the CPU can store.

A relaxed model. Henceforth, we make the following assumption: the memory contains a special array of arbitrary length, such that upon a read or write operation, the read head moves forward, and when the read head moves across the end of the array, it wraps around to the beginning of the

array — this wrapping around does not get charged to the cost of locality. It is not hard to observe that Lemma 8.2 holds even under this relaxed memory model. Note that for a lower bound to hold under this relaxed memory model makes it a stronger (i.e., better) lower bound.

Fact 8.3. *A machine $M^{[r]}$ with r blocks of CPU cache can be simulated by a machine $M^{[1]}$ with a single block of CPU cache with the following simulation overheads: suppose that $M^{[r]}$ accesses B blocks and L contiguous regions, then $M^{[1]}$ accesses rB blocks and $L + r$ contiguous regions of memory.*

Further, if $M^{[r]}$ oblivious simulates a functionality f with perfect/statistical/computational security respectively, then $M^{[1]}$ oblivious simulates f with perfect/statistical/computational security respectively.

Proof. The simulation works as follows: $M^{[1]}$ creates a special array (henceforth denoted cache in memory (supporting free wrap-around) of r blocks to simulate $M^{[r]}$'s CPU cache. Further, $M^{[1]}$ introduces r additional read/write heads, each pointing to a distinct location in this special array cache.

Upon every memory access of $M^{[r]}$, all r additional heads of $M^{[1]}$, will read and/or write one block.

- If $M^{[r]}$ wants to write a block to the i-th location in cache, the head pointing to $\mathsf{cache}[i]$ will write the block there, and all other heads will read and write the same block back.
- If $M^{[r]}$ does not want to write any block to cache, then all of the r additional read heads will read and write the same block back.

Clearly, if $M^{[r]}$ accesses B blocks and L contiguous regions, then $M^{[1]}$ accesses rB blocks and $L + r$ contiguous regions of memory. Further, it is not hard to see that the above simulation of $M^{[r]}$ with $M^{[1]}$ is obliviousness preserving. □

Theorem 8.4 (Any efficient ORAM scheme with restricted leakage must necessarily suffer from poor locality)**.** *For any $\epsilon > 0$, for sufficiently large N, any ORAM scheme satisfying the restricted leakage that has (h, l)-locality with $h = l$ and memory blowup of $m \leq N$, and consuming r CPU cache will incur bandwidth blowup of β such that $r\beta(l + r) \geq \Omega(N^{1-\epsilon})$.*

In particular, when both r and l are $\mathsf{poly}\log N$, for any $\epsilon > 0$, for large enough N, the bandwidth blowup $\beta \geq \Omega(N^{1-\epsilon})$.

Proof. Suppose that there is an ORAM scheme with the aforementioned restricted leakage where the CPU has r blocks of cache, satisfying β bandwidth (i.e., work) blowup and l locality blowup. Then, by Fact 8.3, there must exist an ORAM scheme with a single block of CPU cache, satisfying $\beta' := r\beta$ bandwidth (i.e., work) blowup and at most $l' := l + r$ locality blowup. By Lemma 8.2, for any constant $\epsilon > 0$, for large enough N, $\beta' l' \geq \Omega(N^{1-\epsilon})$. Thus, we conclude that it must be the case that $r\beta(l + r) \geq \Omega(N^{1-\epsilon})$. □

Acknowledgments

This work was partially supported by a Junior Fellow award from the Simons Foundation to Gilad Asharov. This work is 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, a Baidu Faculty Research Award and a Google Ph.D. Fellowship Award.

References

[1] Bitonic sorter. `https://en.wikipedia.org/wiki/Bitonic_sorter`. Online; accessed August 2017.

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

[3] Miklós Ajtai, János Komlós, and Endre Szemerédi. Sorting in c log n parallel sets. *Combinatorica*, 3(1):1–19, 1983.

[4] Daniel Apon, Jonathan Katz, Elaine Shi, and Aishwarya Thiruvengadam. Verifiable oblivious storage. In *Public Key Cryptography (PKC'14)*, pages 131–148, 2014.

[5] Lars Arge, Paolo Ferragina, Roberto Grossi, and Jeffrey Scott Vitter. On sorting strings in external memory (extended abstract). In *ACM Symposium on the Theory of Computing (STOC '97)*, pages 540–548, 1997.

[6] Gilad Asharov, Moni Naor, Gil Segev, and Ido Shahaf. Searchable symmetric encryption: optimal locality in linear space via two-dimensional balanced allocations. In *ACM Symposium on Theory of Computing (STOC '16)*, pages 1101–1114, 2016.

[7] Kenneth E. Batcher. Sorting networks and their applications. In *AFIPS '68*, pages 307–314, 1968.

[8] Elette Boyle and Moni Naor. Is there an oblivious RAM lower bound? In *ACM Conference on Innovations in Theoretical Computer Science (ITCS '16)*, pages 357–368, 2016.

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

[10] David Cash, Paul Grubbs, Jason Perry, and Thomas Ristenpart. Leakage-abuse attacks against searchable encryption. In *ACM SIGSAC Conference on Computer and Communications Security (CCS '15)*, pages 668–679, 2015.

[11] David Cash, Stanislaw Jarecki, Charanjit S. Jutla, Hugo Krawczyk, Marcel-Catalin Rosu, and Michael Steiner. Highly-scalable searchable symmetric encryption with support for boolean queries. In *Advances in Cryptology - CRYPTO 2013. Proceedings, Part I*, pages 353–373, 2013.

[12] David Cash and Stefano Tessaro. The locality of searchable symmetric encryption. In *Advances in Cryptology - EUROCRYPT 2014*, volume 8441 of *LNCS*, pages 351–368. Springer, 2014.

[13] T-H. Hubert Chan and Elaine Shi. Circuit OPRAM: A (somewhat) tight oblivious parallel RAM. `http://eprint.iacr.org/curr/`.

[14] Melissa Chase and Seny Kamara. Structured encryption and controlled disclosure. In *Asiacrypt*, pages 577–594. Springer, 2010.

[15] Kai-Min Chung, Zhenming Liu, and Rafael Pass. Statistically-secure ORAM with $\tilde{O}(\log^2 n)$ overhead. In *Asiacrypt*, 2014.

[16] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2009.

[17] Reza Curtmola, Juan A. Garay, Seny Kamara, and Rafail Ostrovsky. Searchable symmetric encryption: improved definitions and efficient constructions. In *ACM Conference on Computer and Communications Security (CCS '06)*, pages 79–88, 2006.

[18] Ivan Damgård, Sigurd Meldgaard, and Jesper Buus Nielsen. Perfectly secure oblivious RAM without random oracles. In *Theory of Cryptography - TCC*, pages 144–163, 2011.

[19] Srinivas Devadas, Marten van Dijk, Christopher W Fletcher, Ling Ren, Elaine Shi, and Daniel Wichs. Onion ORAM: a constant bandwidth blowup oblivious RAM. In *Theory of Cryptography Conference (TCC '16)*, pages 145–174. Springer, 2016.

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

[21] Christopher W. Fletcher, Ling Ren, Albert Kwon, Marten van Dijk, Emil Stefanov, and Srinivas Devadas. RAW Path ORAM: A low-latency, low-area hardware ORAM controller with integrity verification. *IACR Cryptology ePrint Archive*, 2014:431, 2014.

[22] Christopher W. Fletcher, Ling Ren, Xiangyao Yu, Marten van Dijk, Omer Khan, and Srinivas Devadas. Suppressing the oblivious RAM timing channel while making information leakage and program efficiency trade-offs. In *HPCA*, pages 213–224, 2014.

[23] Craig Gentry, Kenny A. Goldman, Shai Halevi, Charanjit S. Jutla, Mariana Raykova, and Daniel Wichs. Optimizing ORAM and using it efficiently for secure computation. In *Privacy Enhancing Technologies Symposium (PETS)*, 2013.

[24] Craig Gentry, Shai Halevi, Steve Lu, Rafail Ostrovsky, Mariana Raykova, and Daniel Wichs. Garbled ram revisited. In *EUROCRYPT*, pages 405–422, 2014.

[25] O. Goldreich. Towards a theory of software protection and simulation by oblivious RAMs. In *STOC*, 1987.

[26] Oded Goldreich. *The Foundations of Cryptography - Volume 2, Basic Applications*. Cambridge University Press, 2004.

[27] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious RAMs. *J. ACM*, 1996.

[28] Michael T. Goodrich. Randomized shellsort: A simple data-oblivious sorting algorithm. *J. ACM*, 58(6):27:1–27:26, December 2011.

[29] Michael T. Goodrich. Zig-zag sort: a simple deterministic data-oblivious sorting algorithm running in o(n log n) time. In *Symposium on Theory of Computing, STOC*, pages 684–693, 2014.

[30] Michael T. Goodrich and Michael Mitzenmacher. Privacy-preserving access of outsourced data via oblivious RAM simulation. In *ICALP*, 2011.

[31] S. Dov Gordon, Jonathan Katz, Vladimir Kolesnikov, Fernando Krell, Tal Malkin, Mariana Raykova, and Yevgeniy Vahlis. Secure two-party computation in sublinear (amortized) time. In *CCS*, 2012.

[32] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *Network and Distributed System Security Symposium, NDSS 2012*. The Internet Society, 2012.

[33] Seny Kamara and Charalampos Papamanthou. Parallel and dynamic searchable symmetric encryption. In *Financial Cryptography and Data Security*, pages 258–274. Springer, 2013.

[34] Georgios Kellaris, George Kollios, Kobbi Nissim, and Adam O'Neill. Generic attacks on secure outsourced databases. In *ACM CCS*, pages 1329–1340, 2016.

[35] Georgios Kellaris, George Kollios, Kobbi Nissim, and Adam O'Neill. Accessing data while preserving privacy. *CoRR*, abs/1706.01552, 2017.

[36] Kaoru Kurosawa and Yasuhiro Ohtaki. How to update documents verifiably in searchable symmetric encryption. In *International Conference on Cryptology and Network Security*, pages 309–328. Springer, 2013.

[37] Eyal Kushilevitz, Steve Lu, and Rafail Ostrovsky. On the (in)security of hash-based oblivious RAM and a new balancing scheme. In *SODA*, 2012.

[38] Chang Liu, Xiao Shaun Wang, Kartik Nayak, Yan Huang, and Elaine Shi. Oblivm: A programming framework for secure computation. In *S & P*, 2015.

[39] Steve Lu and Rafail Ostrovsky. How to garble ram programs. In *EUROCRYPT*, 2013.

[40] Martin Maas, Eric Love, Emil Stefanov, Mohit Tiwari, Elaine Shi, Kriste Asanovic, John Kubiatowicz, and Dawn Song. Phantom: Practical oblivious computation in a secure processor. In *CCS*, 2013.

[41] Kartik Nayak, Xiao Shaun Wang, Stratis Ioannidis, Udi Weinsberg, Nina Taft, and Elaine Shi. GraphSC: Parallel Secure Computation Made Easy. In *IEEE S & P*, 2015.

[42] Benny Pinkas and Tzachy Reinman. Oblivious RAM revisited. In *CRYPTO*, 2010.

[43] Ling Ren, Xiangyao Yu, Christopher W. Fletcher, Marten van Dijk, and Srinivas Devadas. Design space exploration and optimization of path oblivious RAM in secure processors. In *ISCA*, pages 571–582, 2013.

[44] Chris Ruemmler and John Wilkes. An introduction to disk drive modeling. *IEEE Computer*, 27(3):17–28, 1994.

[45] Elaine Shi, T.-H. Hubert Chan, Emil Stefanov, and Mingfei Li. Oblivious RAM with $O((\log N)^3)$ worst-case cost. In *ASIACRYPT*, 2011.

[46] Emil Stefanov and Elaine Shi. Multi-cloud oblivious storage. In *ACM Conference on Computer and Communications Security (CCS)*, 2013.

[47] Emil Stefanov and Elaine Shi. Oblivistore: High performance oblivious cloud storage. In *IEEE Symposium on Security and Privacy (S & P)*, 2013.

[48] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM – an extremely simple oblivious ram protocol. In *CCS*, 2013.

[49] Peter Van Liesdonk, Saeed Sedghi, Jeroen Doumen, Pieter Hartel, and Willem Jonker. Computationally efficient searchable symmetric encryption. In *Workshop on Secure Data Management*, pages 87–100. Springer, 2010.

[50] Jeffrey Scott Vitter. External memory algorithms and data structures. *ACM Comput. Surv.*, 33(2):209–271, 2001.

[51] Jeffrey Scott Vitter. Algorithms and data structures for external memory. *Foundations and Trends in Theoretical Computer Science*, 2(4):305–474, 2006.

[52] Xiao Shaun Wang, T-H. Hubert Chan, and Elaine Shi. Circuit ORAM: On Tightness of the Goldreich-Ostrovsky Lower Bound. In *ACM CCS*, 2015.

[53] Xiao Shaun Wang, Yan Huang, T-H. Hubert Chan, Abhi Shelat, and Elaine Shi. SCORAM: Oblivious RAM for Secure Computation. In *CCS*, 2014.

[54] Xiao Shaun Wang, Kartik Nayak, Chang Liu, T-H. Hubert Chan, Elaine Shi, Emil Stefanov, and Yan Huang. Oblivious Data Structures. In *CCS*, 2014.

[55] Peter Williams and Radu Sion. Usable PIR. In *Network and Distributed System Security Symposium (NDSS)*, 2008.

[56] Peter Williams and Radu Sion. Round-optimal access privacy on outsourced storage. In *ACM Conference on Computer and Communication Security (CCS)*, 2012.

[57] Peter Williams, Radu Sion, and Bogdan Carbunar. Building castles out of mud: practical access pattern privacy and correctness on untrusted storage. In *CCS*, pages 139–148, 2008.

[58] Peter Williams, Radu Sion, and Alin Tomescu. Privatefs: A parallel oblivious file system. In *ACM Conference on Computer and Communications Security (CCS)*, 2012.

[59] Yupeng Zhang, Jonathan Katz, and Charalampos Papamanthou. All your queries are belong to us: The power of file-injection attacks on searchable encryption. In *USENIX Security Symposium (USENIX '16)*, pages 707–720, 2016.

A 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 $\mathsf{bitonic}$ 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 $\widehat{S}_1 = \big\langle \min(a_0, a_{n/2}), \min(a_1, a_{n/2+1}), \ldots, \min(a_{n/2-1}, a_{n-1}) \big\rangle$.
- Let $\widehat{S}_2 = \big\langle \max(a_0, a_{n/2}), \max(a_1, a_{n/2+1}), \ldots, \max(a_{n/2-1}, a_{n-1}) \big\rangle$.

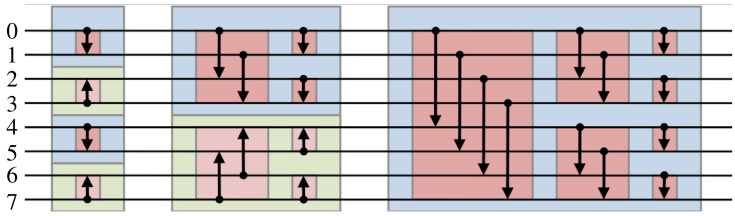

Figure 4: Bitonic sorting network for 8 inputs. Input come in from the left end, and outputs are on the right end. When two numbers are joined by an arrow, they are compared, and if necessary are swapped such that the arrow points from the smaller number toward the larger number. This figure is modified from [1].

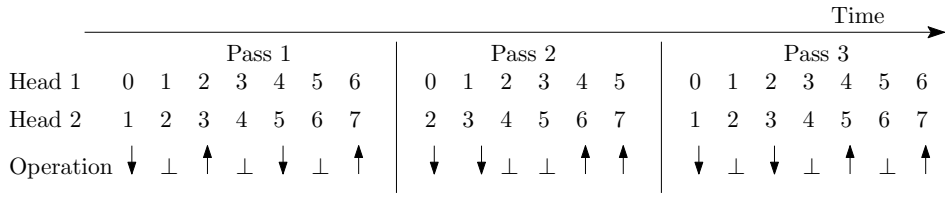

Figure 5: Locality of Bitonic Sort for 8 elements. The figure shows the movement of heads and the operation performed for an 8 element array. For each input, either a compare-and-swap operation is performed in the specified direction or the input is ignored as denoted by $\perp$. For each pass, each head scans through the data blocks once. Bitonic Sort performs $O(\log^2 N)$ passes and hence, has locality $(2, O(\log^2 N))$. The figure shows the first 3 passes out of the required 6 passes for 8 elements (see Figure 4).

– $\overrightarrow{S}_1 = \overrightarrow{\mathsf{Split}}(\widehat{S}_1)$, $\overrightarrow{S}_2 = \overrightarrow{\mathsf{Split}}(\widehat{S}_2)$ 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 [7] for details.

To sort an array of 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: 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 to the actual data. As of 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 4 gives a graphic representation of the algorithm for input size 8 and Figure 5 illustrates its locality. The $(2, O(\log^2 n))$ locality of Bitonic sort is also obvious from the figure.

A.1 Concurrent Executions of Bitonic Sorts

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

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

Specifically, the $k = n$ case is Theorem 4.1.

B Locally Initializable, Statistically Secure Oblivious Memory and Data Structures

In this section, we formally define two building blocks, a locally initializable position-based ORAM and an locally initializable oblivious binary search tree. Both building blocks adopt the core technique of manipulating of position labels in ORAM constructions. Both abstractions have been described in earlier works [23, 38, 54] — however, locality-friendly initialization was not of concern in those works.

B.1 Locally Initializable Position-Based ORAM

Roughly speaking, a position-based ORAM is an ORAM where one can store and access a position map for free — specifically, a position map is a data structure that stores the *position label* of each block in the ORAM, which indicates the block's physical location in memory. In our formal abstraction below, each data access request will supply a correct position label as input, and besides the requested block, the access algorithm will additionally output the requested block's new position label, which is used to update the physical location of the block after the access. Similar to Section 4.3, we need a position-based ORAM that has a $\mathsf{Initialize}$ algorithm that has good locality.

Functionality: A position-based ORAM scheme T (for non-contiguous memory addresses) is the following functionality:

- $\mathsf{posmap} \leftarrow \mathsf{T.Build}(X)$: takes an input array X of blocks of the form $(\mathsf{addr}, \mathsf{data})$, and outputs a position map posmap that contains a position label for every block in X.
- $(\mathsf{pos}', \mathsf{data}) \leftarrow \mathsf{T.Access}(\mathsf{op}, \mathsf{addr}, \mathsf{data}, \mathsf{pos})$: takes in a possibly dummy request addr with its position label pos, outputs the requested block if $\mathsf{op} = \texttt{read}$ or update the block if $\mathsf{op} = \texttt{write}$. It also outputs the updated position map pos' for addr.

T reveals the number of elements in X and the total number of Access operations in the sequence, i.e., $\mathsf{leakage}(\mathbf{I}) := (|X|, m)$. Obliviousness is then defined as in Definition 3.2.

Construction. Let α denote any super-constant function. The idea is to have a Circuit ORAM tree, but truncate the tree at a height with $\frac{n}{\alpha \log \lambda}$ nodes — in this way, the Circuit ORAM tree has $\frac{n}{\alpha \log \lambda}$ leaves. We make the capacity of all non-leaf nodes a suitably large constant (e.g., 4), and the capacity of all leaf nodes $4 \cdot \alpha \log \lambda$. Just like in Circuit ORAM, we have a stash of $\alpha \log \lambda$ in size. For accesses, we adopt exactly the same algorithm as Circuit ORAM.

The Build algorithm proceeds in the most obvious manner. We assign each non-dummy block in the input array X to a random leaf node. By a simple application of the Chernoff bound, it is not hard to see that each leaf node is at most half full except with $\mathsf{negl}(\lambda)$ probability. The set of all label assignments will eventually be output by the algorithm as posmap. We now invoke an instance of our oblivious sorting algorithm in Section 7) to place the blocks into the leaf nodes. This immediately gives rise to the following theorem.

Theorem B.1 (Locally initializable, statistically secure position-based ORAM)**.** *For any super-constant function $\alpha = \omega(1)$, there exists a statistically secure, position-based ORAM scheme that except with $\mathsf{negl}(\lambda)$ probability, costs $O(n \log n \log\log^2 \lambda)$ work and $(3, O(\log n \log\log^2 \lambda))$ locality for initialization with n elements, and $O(\log n \log\log^2 \lambda)$ work and $(2, O(\log n \log\log^2 \lambda)$ locality for each access.*

B.2 Locally Initializable Oblivious Binary Search Tree

We now define a useful building block called an oblivious binary search tree (OBST). It is initialized from an input key-value pairs, and then supports binary search operations on the array. We would like initialization and binary search operations to be efficient and have good locality.

Functionality: An oblivious binary search tree scheme T has the following functionality:

- $\mathsf{T}.\mathsf{Build}(X)$: takes an input array X of key-value pairs in the form (k, v), and initializes its internal structure that will later facilitate requests.
- $v \leftarrow \mathsf{T}.\mathsf{Access}(\mathsf{op}, k, v')$: takes in an operation, a key k to search for, and possibly an updated value v'. If $\mathsf{op} = \texttt{read}$, T returns the value v of the node with key k (if it exists and returns $\perp$ otherwise). If $\mathsf{op} = \texttt{write}$, the node's value is updated to v'.

An OBST reveals the number of elements in X and the total number of accesses operations in the sequence, i.e., $\mathsf{leakage}(\mathbf{I}) := (|X|, m)$. Obliviousness is then defined as in Definition 3.2.

Construction. Wang et al. [54] show how to construct an OBST from any position-based ORAM in a blackbox manner. At a high level, the idea is to have a sequence of index ORAMs where a node in each index ORAM stores the position labels of two children nodes in the next ORAM. In this way, each binary-tree search operation can be performed through $\log n$ position-ORAM accesses. It is not hard to see that if the underlying position-based ORAM can be initialized with locality, so can the resulting OBST. Therefore, we immediately obtain the following theorems.

Theorem B.2 (Statistically secure oblivious binary search tree)**.** *For any super constant function $\alpha = \omega(1)$, there exists an oblivious binary search tree scheme that has $O(n \log n \log\log^2 \lambda)$ work and $(3, O(\log n \log\log^2 \lambda))$ to initialize with n elements, and $O(\log^2 n \log\log^2 \lambda)$ work and $(2, O(\log^2 n \log\log^2 \lambda)$ locality for each request to achieve $\mathsf{negl}(N)$ statistical failure probability.*

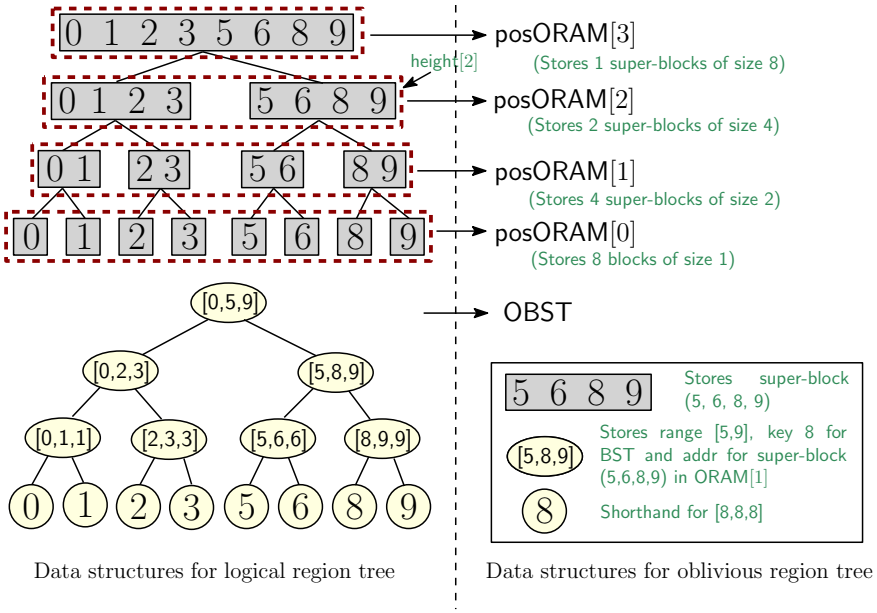

Figure 6: Oblivious range tree with locality.

Proof. Due to Theorem B.1 and due to the blackbox construction of an oblivious binary-tree from a position-based ORAM [54]. Note that the construction preserves initialization locality of the underlying position-based ORAM. We propose a further improvement to Wang et al. [54]'s algorithm. As observed by Chan et al. [13], we may merge the stashes of all index and data ORAMs, and the combined stash has at most $\alpha \log \lambda$ utilization except with $\mathsf{negl}(\lambda)$ probability. With the merged stash, each binary-tree search operation perform a single lookup of the combined stash upfront, rather than having to look up the stashes of all index and data ORAMs. □

B.3 Better Oblivious Range Trees and Range ORAMs

Using the primitives in previous sections, we can construct more efficient oblivious Range Trees and Range ORAMs. We simply replace each initializable hierarchical ORAM with an locally initializable position-based ORAM, and replace the metadata ORAM with an OBST, as shown in Figure 6.

Due to our modular approach of presentation, the more efficient oblivious range tree constructions can be naturally adapted from Section 5.2 with minimum changes. The only change is that we need to manipulate position labels. Specifically, the OBST stores the positional labels for all (super-)blocks in all heights, and queries to OBST returns not only the two (super-)blocks that intersect $[s, t]$, but also their position labels, which will be fed into each position-based ORAM to retrieve the (super-)blocks. This immediately gives rise to the following results.

Theorem B.3 (Statistically secure oblivious Range Tree)**.** *There exists a statistically secure oblivious Range Tree that except with* $\mathsf{negl}(\lambda)$ *probability, achieves* $O(n \log^2 n \log\log^2 \lambda)$ *work and* $(3, O(\log^2 n \log\log^2 \lambda))$ *locality for preprocessing* n *primitives blocks during the build phase, and* $O((\mathsf{len} + \log n) \cdot \log n \log\log^2 \lambda)$ *work and* $(2, O(\log^2 n \log\log^2 \lambda)$ *locality for fetching a range of length* len*.*

Theorem B.4 (Statistically secure Range ORAM with locality)**.** *There exists a statistically secure Range ORAM such that except with* $\mathsf{negl}(N)$ *probability, accessing any contiguous memory range*

of size len *requires* $O(\alpha \mathsf{len} \cdot \log^3 N \log\log^2 N)$ *work and* $(3, O(\log^3 N \log\log^2 N))$ *locality, where* N *denotes both the total memory size as well as the security parameter.*

C Locally Initializable, Perfectly Secure ORAM

We observe that the perfectly secure ORAM scheme by Damgård et al. [18] can be modified into a perfectly secure, locally initializable ORAM. Since Damgård et al. [18]'s perfectly secure ORAM also follows the hierarchical ORAM paradigm, we can make it locally initializable in a similar manner as how we made Goldreich and Ostrovsky's ORAM [25] locally initializable.

Background on perfectly secure ORAM. Damgård et al. [18]'s perfectly secure ORAM basically follows the hierarchical ORAM framework originally proposed by Goldreich and Ostrovsky [25]. The main difference is the following: in Goldreich and Ostrovsky's orginal hierarchical ORAM, each level of the hierarchy is an oblivious hash table (i.e., a hash table with an oblivious rebuilding procedure), where each element is mapped to a bin in the hash table using a pseudorandom function (PRF) whose key is kept private by the client. Now, in order to make the ORAM perfectly secure, we can no long rely on a PRF. Thus, Damgård et al. [18]'s idea is to obliviously and randomly permute each level of the hierarchy when it is rebuilt — such oblivious random permutation can be realized through the help of oblivious sorting as Damgård et al. [18] showed. This, however, gives rise to a new problem. When a block is being requested from the level, the client must know where to look in this permuted level. The most obvious way to do this is for the client to store a large position map that remembers where each block is — however, this would incur a linear amount of client storage.

Thus, Damgård et al. [18]'s idea is to introduce a special (recursive) position map for hierarchical ORAM — effectively, instead of storing this position map locally, the client recurses and stores it in an ORAM on the server. In every position map ORAM, each block stores the positions of at least 2 adjacent addresses in the next recursion depth — this guarantees that the number of blocks will reduce by at least a half every time we recurse; and thus after logarithmically many recursion depths, the amount of metadata will be $O(1)$ blocks. This recursive position map is described as a rather involved binary tree structure in Damgård et al. [18]'s paper, but here we illustrate it in a simpler but equivalent manner to aid understanding. The idea of adopting a recursive position map was commonly used in tree-based ORAM constructions [45]. In fact the idea is similar here although some special treatments are necessary for position maps for hierarchical ORAMs. Specifically, each recursive ORAM that stores the position map is also a perfectly secure hierarchical ORAM. Over all recursion levels, the following invariant must be satisfied:

> (*Invariant for a hierarchical ORAM's position map.*) If in the final data ORAM, a block at logical address is stored in level ℓ of the hierarchical ORAM, then the position map for the block must be stored in its position map ORAM at level ℓ or smaller, and the position map for the position map for this block must also be stored at level ℓ or smaller, and so on.

If this invariant is satisfied, the benefit is the following: when levels $1, 2, \ldots, \ell$ are being rebuilt for the data ORAM, then all position ORAMs will rebuild in sync for the same levels $1, 2, \ldots, \ell$, such that the position map (and the position map of the position map, etc) for the updated blocks are updated in sync.

Locality-friendly initialization. It is not difficult to see that given a list of data blocks, we can initiailize Damgård et al. [18]'s perfectly secure ORAM using poly-logarithmically many oblivious sorting operations. Basically, we first rebuild the data ORAM level just like how we rebuild Goldreich and Ostrovsky's ORAM but now assigning each block to a random location in a level rather than pseudorandom. Once the data ORAM has been rebuilt, we can rebuild its position map level correspondingly through at most logarithmically many oblivious sorts as well; and then we can rebuild the position map of the position map, etc.

Thus we immediately obtain the following theorem where oblivious sort is realized using our locality-friendly implementation of bitonic sort.

Theorem C.1 (Perfectly secure, locally initializable ORAM)**.** *There exists a perfectly secure ORAM scheme that can be initialized with n blocks using $n \cdot \mathsf{poly}\log n$ work and $(2, \mathsf{poly}\log n)$ locality, and can serve an access using $\mathsf{poly}\log n$ work and $(2, \mathsf{poly}\log n)$ locality.*

In our main body of the paper, we can leverage such a perfectly secure, locally initializable ORAM to obtain a perfectly secure Range ORAM (using bitonic sort as the locality-friendly sorting algorithm).