OptORAMa: Optimal Oblivious RAM

Gilad Asharov* Ilan Komargodski† Wei-Kai Lin‡ Kartik Nayak§
Enoch Peserico¶ Elaine Shi‖

Abstract

Oblivious RAM (ORAM), first introduced in the ground-breaking work of Goldreich and Ostrovsky (STOC '87 and J. ACM '96) is a technique for provably obfuscating programs' access patterns, such that the access patterns leak no information about the programs' secret inputs. To compile a general program to an oblivious counterpart, it is well-known that $\Omega(\log N)$ amortized blowup is necessary, where N is the size of the logical memory. This was shown in Goldreich and Ostrovksy's original ORAM work for statistical security and in a somewhat restricted model (the so called *balls-and-bins* model), and recently by Larsen and Nielsen (CRYPTO '18) for computational security.

A long standing open question is whether there exists an *optimal* ORAM construction that matches the aforementioned logarithmic lower bounds (without making large memory word assumptions, and assuming a constant number of CPU registers). In this paper, we resolve this problem and present the first secure ORAM with $O(\log N)$ amortized blowup, assuming one-way functions. Our result is inspired by and non-trivially improves on the recent beautiful work of Patel et al. (FOCS '18) who gave a construction with $O(\log N \cdot \log\log N)$ amortized blowup, assuming one-way functions.

One of our building blocks of independent interest is a linear-time deterministic oblivious algorithm for tight compaction: Given an array of n elements where some elements are marked, we permute the elements in the array so that all marked elements end up in the front of the array. Our $O(n)$ algorithm improves the previously best known deterministic or randomized algorithms whose running time is $O(n \cdot \log n)$ or $O(n \cdot \log\log n)$, respectively.

Keywords: Oblivious RAM, randomized algorithms, compaction.

*Cornell Tech, asharov@cornell.edu
†Cornell Tech, komargodski@cornell.edu
‡Cornell University, wklin@cs.cornell.edu
§VMware and Duke University, nkartik@vmware.com
¶Univ. Padova, enoch@dei.unipd.it
‖Cornell University, runting@gmail.com

Contents

1 Introduction **1**
- 1.1 Our Results: Optimal Oblivious RAM . . . 1
- 1.2 Our Results: Optimal Oblivious Tight Compaction . . . 2

2 Technical Roadmap **3**
- 2.1 Oblivious RAM . . . 3
- 2.2 Tight Compaction . . . 8

3 Preliminaries **10**
- 3.1 Oblivious Machines . . . 11

4 Oblivious Building Blocks **14**
- 4.1 Oblivious Sorting Algorithms . . . 15
- 4.2 Oblivious Random Permutations . . . 16
- 4.3 Oblivious Bin Placement . . . 18
- 4.4 Oblivious Hashing . . . 19
- 4.5 Oblivious Cuckoo Hashing . . . 21
- 4.6 Oblivious Dictionary . . . 24
- 4.7 Oblivious Balls-into-Bins Sampling . . . 25

5 Oblivious Tight Compaction **26**
- 5.1 Reducing Tight Compaction to Loose Compaction . . . 26
- 5.2 Loose Compaction . . . 29
- 5.3 Oblivious Distribution . . . 35

6 Interspersing Randomly Shuffled Arrays **36**
- 6.1 Interspersing Two Arrays . . . 36
- 6.2 Interspersing Multiple Arrays . . . 37
- 6.3 Interspersing Reals and Dummies . . . 38
- 6.4 Perfect Oblivious Random Permutation (Proof of Theorem 4.6) . . . 39

7 BigHT: Oblivious Hashing for Non-Recurrent Lookups **40**

8 SmallHT: Oblivious Hashing for Small Bins **43**
- 8.1 Step 1 – Add Dummies and Shuffle . . . 43
- 8.2 Step 2 – Evaluate Assignment with Metadata Only . . . 44
- 8.3 SmallHT Construction . . . 45
- 8.4 CombHT: Combining BigHT with SmallHT . . . 47

9 Oblivious RAM **48**

References **51**

A Comparison with Prior Works **55**

B Details on Oblivious Cuckoo Assignment **56**

C Deferred Proofs **57**
- C.1 Proof of Theorem 5.2 . . . 57
- C.2 Deferred Proofs from Section 6 . . . 58
- C.3 Proof of Security of BigHT (Theorem 7.2) . . . 60
- C.4 Proof of Security of SmallHT (Theorem 8.6) . . . 65
- C.5 Proof of Security of CombHT (Theorem 8.8) . . . 68
- C.6 Proof of Security of ORAM (Theorem 9.2) . . . 69

1 Introduction

Oblivious RAM (ORAM), first proposed by Goldreich and Ostrovsky [28, 30], is a technique to compile *any* program into a functionally equivalent one, but whose memory access patterns are independent of the program's secret inputs. The overhead of an ORAM is defined as the (multiplicative) blowup in runtime of the compiled program. Since Goldreich and Ostrovsky's seminal work, ORAM has received much attention due to its applications in cloud computing, secure processor design, multi-party computation, and theoretical cryptography (for example, [6, 24, 25, 27, 44–46, 50, 56, 58, 59, 63, 66, 67])

For more than three decades, the biggest open question in this line of work is regarding the *optimal* overhead of ORAM. Goldreich and Ostrovsky's original work [28, 30] showed a construction with $O(\log^3 N)$ blowup in runtime, assuming the existence of one-way functions, where N denotes the memory size consumed by the original non-oblivious program. On the other hand, they proved that any ORAM scheme must incur at least $\Omega(\log N)$ overhead, but their lower bound is restricted to schemes that treat the contents of each memory word as "indivisible" (see Boyle and Naor [7]) and make no cryptographic assumptions. In a recent work, Larsen and Nielsen [40] showed that $\Omega(\log N)$ overhead is necessary for all *online* ORAM schemes,[1] even ones that use cryptographic assumptions and might perform non-trivial encodings on the contents of the memory. Since Goldreich and Ostrovsky's work, a long line of research has been dedicated to improving the asymptotic efficiency of ORAM [12, 32, 39, 57, 60, 62]. Prior to our work, the best known scheme, allowing computational assumptions, is the elegant work by Patel et al. [52]: they showed the existence of an ORAM with $O(\log N \cdot \log\log N)$ overhead, assuming one-way functions. In comparison with Goldreich and Ostrovksy's original $O(\log^3 N)$ result, Patel's result seems tantalizingly close to matching the lower bound, but unfortunately we are still not there yet and the construction of an optimal ORAM continues to elude us even after more than 30 years.

1.1 Our Results: Optimal Oblivious RAM

We resolve this long-standing problem by showing a matching upper bound to Larsen and Nielsen's [40] lower bound: an ORAM scheme with $O(\log N)$ overhead and negligible security in λ, where N is the size of the memory and λ is the security parameter, assuming one-way functions. More concretely, we show: [2]

Theorem 1.1. *Assume that there is a PRF family that is secure against any probabilistic polynomial-time adversary except with a negligible small probability in λ. Assume that $\lambda \le N \le T \le \mathsf{poly}(\lambda)$ for any fixed polynomial $\mathsf{poly}(\cdot)$, where T is the number of accesses. Then, there is an ORAM scheme with $O(\log N)$ overhead and whose security failure probability is upper bounded by a suitable negligible function in λ.*

In the aforementioned results and throughout this paper, unless otherwise noted, we shall assume a standard word-RAM where each memory word has at least $w = \log N$ bits, i.e., large enough to store its own logical address. We assume that word-level addition and boolean operations can be done in unit cost. We assume that the CPU has constant number of private registers. For our ORAM construction, we additionally assume that a single evaluation of a pseudorandom function

[1] An ORAM scheme is *online* if it supports accesses arriving in an online manner, one by one. Almost all known schemes have this property.

[2] Note that for the (sub-)exponential security regime, e.g., failure probability of $2^{-\lambda}$ or $2^{-\lambda^\epsilon}$ for some $\epsilon \in (0, 1)$, perfectly secure ORAM schemes [14, 19] asymptotically outperform known statistically or computationally secure constructions assuming that $N = \mathsf{poly}(\lambda)$.

(PRF), resulting in at least word-size number of pseudo-random bits, can be done in unit cost.[3] Note that all earlier computationally secure ORAM schemes, starting with the work of Goldreich and Ostrovsky [28, 30], make the same set of assumptions. Additionally, we remark that our result can be made statistically secure if one assumes a private random oracle to replace the PRF (the known logarithmic ORAM lower bound [28, 30, 40] still hold in this setting). Finally, we note that our construction suffers from huge constants due to the use of certain expander graphs; improving the concrete constant is left for future work.

In Appendix A we provide a comparison with previous works, where we make the comparison more accurate and meaningful by explicitly stating the dependence on the error probability (which was assumed to be some negligible functions in previous works).

1.2 Our Results: Optimal Oblivious Tight Compaction

Closing the remaining $\log \log N$ gap for ORAM turns out to be highly challenging. Along the way, we actually construct an important building block, that is, a *deterministic*, linear-time, oblivious *tight compaction* algorithm. This result is an important contribution on its own, and has intimate connections to classical algorithms questions, as we explain below.

Tight compaction is the following task: given an input array of size n containing either real or dummy elements, output a permutation of the input array where all real elements appear in the front. Tight compaction can be considered as a restricted form of sorting, where each element in the input array receives a 1-bit key, indicating whether it is real or dummy. One naïve solution for tight compaction, therefore, is to rely on oblivious sorting to sort the input array [1, 31]; unfortunately, due to recent lower bounds [21, 43], we know that any oblivious sorting scheme must incur $\Omega(n \cdot \log n)$ time on a word-RAM, either assuming that the algorithm treats each element as "indivisible" [43] or assuming that the famous Li-Li network coding conjecture [42] is true [21].

A natural question, therefore, is whether we can do asymptotically better than just naïvely sorting the input. It turns out that this question is related to a line of work in the classical algorithms literature, that is, the design of switching networks and routing on such networks [1, 4, 5, 22, 54, 55]. First, a line of combinatorial works showed the existence of linear-sized super-concentrators [53, 54, 61], i.e., switching networks with n inputs and n outputs such that vertex-disjoint paths exist from any k elements in the inputs to any k positions in the outputs. One could leverage a linear-sized super-concentrator construction to *obliviously* route all the real elements in the input to the front of the output array deterministically and in linear time (by routing elements along the routes), but it is not clear yet how to find routes (i.e., a set of vertex-disjoint paths) from the real input positions to the front of the output array.

In an elegant work in 1996, Pippenger [55] showed a deterministic, linear-time algorithm for route-finding but unfortunately the algorithm is *not oblivious*. Shortly afterwards, Leighton et al. [41] showed a probabilistic algorithm that tightly compacts n elements in $O(n \cdot \log \log \lambda)$ time with $1 - \mathsf{negl}(\lambda)$ probability — their algorithm is *almost oblivious* except for leaking the number of reals and dummies. After Leighton et al. [41], this line of work remained somewhat stagnant for almost two decades. Only recently, did we see some new results: Mitchell and Zimmerman [48] as well as Lin et al. [43] showed how to achieve the same asymptotics as Leighton et al. [41] but now making the algorithm fully oblivious.

In this paper, we give an explicit construction of a deterministic, oblivious algorithm that tightly compacts any input array of n elements in linear time, as stated in the following theorem:

[3] Alternatively, if we use number of IOs as an overhead metric, we only need to assume that the CPU can evaluate a PRF internally without writing to memory, but the evaluation need not be unit cost.

Theorem 1.2 (Linear-time oblivious tight compaction)**.** *There is a deterministic, oblivious tight compaction algorithm that compacts n elements in $O(\lceil D/w \rceil \cdot n)$ time on a word-RAM where D is the bit-width for encoding each element and $w \geq \log n$ is the word size.*

Our algorithm is *not comparison-based* and *not stable* and this is inherent. Specifically, Lin et al. [43] recently showed that any stable, oblivious tight compaction algorithm (that treats elements as indivisible) must incur $\Omega(n \cdot \log n)$ runtime, where stability requires that the real elements in the output must appear in the same order as the input. Further, due to the well-known 0-1 principle [18,65], any comparison-based tight compaction algorithm must incur at least $\Omega(n \cdot \log n)$ runtime as well.[4]

Not only our ORAM construction relies on the above compaction algorithm in several key points, but it is a useful primitive independently. For example, we use our compaction algorithm to give a perfectly oblivious algorithm that randomly permutes arrays of n elements in (worst-case) $O(n \cdot \log n)$ time. All previously known such constructions have some probability of failure.

2 Technical Roadmap

We give a high-level overview of our results. In Section 2.1 we provide a high-level overview of our ORAM construction which uses an oblivious tight compaction algorithm. In Section 2.2 we give a high-level overview of the techniques underlying our tight compaction algorithm.

2.1 Oblivious RAM

In this section we present a high-level description of the main ideas and techniques underlying our ORAM construction. Full details are given later in the corresponding technical sections.

Hierarchical ORAM. The hierarchical ORAM framework, introduced by Goldreich and Ostrovsky [28, 30] and improved in subsequent works (e.g., [12, 32, 39]), works as follows. For a logical memory of N blocks, we construct a hierarchy of hash tables, henceforth denoted $T_1, \ldots, T_L$ where $L = \log N$. Each T_i stores 2^i memory blocks. We refer to table T_i as the i-th level. In addition, we store next to each table a flag indicating whether the table is *full* or *empty*. When receiving an access request to read/write some logical memory address addr, the ORAM proceeds as follows:

- **Read phase.** Access each non-empty levels $T_1, \ldots, T_L$ in order and perform Lookup for addr. If the item is found in some level T_i, then when accessing all non-empty levels $T_{i+1}, \ldots, T_L$ look for dummy.
- **Write back.** If this operation is read, then store the found data in the read phase and write back the data value to T_0. If this operation is write, then ignore the associated data found in the read phase and write the value provided in the access instruction in T_0.
- **Rebuild:** Find the first empty level ℓ. If no such level exists, set $\ell := L$. Merge all $\{T_j\}_{0 \leq j \leq \ell}$ into T_ℓ. Mark all levels $T_1, \ldots, T_{\ell-1}$ as empty and T_ℓ as full.

For each access, we perform $\log N$ lookups, one per hash table. Moreover, after t accesses, we rebuild the i-th table $\lceil t/2^i \rceil$ times. When implementing the hash table using the best known oblivious hash table (e.g., oblivious Cuckoo hashing [12, 32, 39]), building a level with 2^k items obliviously requires $O(2^k \cdot \log(2^k)) = O(2^k \cdot k)$ time. This building algorithm is based on oblivious

[4] Although the algorithm of Leighton et al. [41] appears to be comparison-based, it is in fact not since the algorithm must tally the number of reals/dummies and make use of this number.

sorting, and its time overhead is inherited from the time overhead of the oblivious sort procedure (specifically, the best known algorithm for obliviously sorting n elements takes $O(n \cdot \log n)$ time [1, 31]). Thus, summing over all levels (and ignoring the $\log N$ lookup operations across different levels for each access), t accesses require $\sum_{i=1}^{\log N} \lceil \frac{t}{2^i} \rceil \cdot O(2^i \cdot i) = O(t \cdot \log^2 N)$ time. On the other hand, lookup takes essentially constant time per level (ignoring searching in stashes which introduce an additive factor) and this $O(\log N)$ per access. Thus, there is an asymmetry between build time and lookup time, and the main overhead is the build.

The work of Patel et al. [52]. Classically (e.g., [12, 28, 30, 32, 39]), oblivious hash tables were built to support (and be secure for) *every* input array. This required expensive oblivious sorting, causing the extra logarithmic factor. The key idea of Patel et al. [52] is to modify the hierarchical ORAM framework to realize ORAM from a weaker primitive: an oblivious hash table that works only for *randomly shuffled input* arrays. Patel et al. describe a novel oblivious hash table such that building a hash table containing n elements can be accomplished without oblivious sorting and consumes only $O(n \cdot \log\log \lambda)$ total time[5] and lookup consumes $O(\log\log n)$ total time. Patel et al. argue that their hash table construction retains security not necessarily for every input, but when the input array is randomly permuted, and moreover the input permutation must be unknown to the adversary.

To be able to leverage this relaxed hash table in hierarchical ORAM, a remaining question is the following: whenever a level is being rebuilt in the ORAM (i.e., a new hash table is being constructed), how do we make sure that the input array is randomly and secretly shuffled? A naïve answer is to employ an oblivious random permutation to permute the input, but known oblivious random permutation constructions require oblivious sorting which brings us back to our starting point. Patel et al. solve this problem and show that there is no need to completely shuffle the input array. Recall that when building some level T_ℓ, the input array consists of only unvisited elements in tables $T_0, \ldots, T_{\ell-1}$ (and T_ℓ too if ℓ is the largest level). Patel et al. argue that the unvisited elements in tables $T_0, \ldots, T_{\ell-1}$ are already randomly permuted *within each table* and the permutation is unknown to the adversary. Then, they presented a new algorithm, called *multi-array shuffle*, that combines these arrays to a shuffled array within $O(n \cdot \log\log \lambda)$ time, where $n = |T_0| + |T_1| + \ldots + |T_{\ell-1}|$.[6] The algorithm is somewhat involved, randomized, and has a negligible probability of failure.

The blueprint. Our construction builds upon and simplifies the construction of Patel et al. To get better asymptotic overhead, we improve their construction in two different aspects:

1. We show how to implement our variant of multi-array shuffle (called *intersperse*) in $O(n)$ time. Specifically, we show a new reduction from *intersperse* to tight compaction.

2. We develop a hash table that supports build in $O(n)$ time assuming that the input array is randomly shuffled. The lookup is $O(1)$, ignoring time spent on looking in stashes. Achieving this is rather non-trivial: first we use a "packing" style trick to construct oblivious Cuckoo hash tables for small sizes where $n \leq \mathsf{poly}\log \lambda$, achieving linear-time build and constant-time lookup. Relying on the advantage we gain for problems of small sizes, we then show how to solve problems of medium and large sizes, again relying on oblivious tight compaction as a building

[5] λ denotes the security parameter. Since the size of the hash table n may be small, here we separate the security parameter from the hash table's size.

[6] The time overhead is a bit more complicated to state and the above expression is for the case where $|T_i| = 2|T_{i-1}|$ for every i (which is the case in a hierarchical ORAM construction).

block. The bootstrapping step from medium to large is inspired by Patel et al. [52] at a very high level, but our concrete construction differs from Patel et al. [52] in many technical details.

We describe the core ideas behind these improvements next. In Section 2.1.1, we present our multi-array shuffle algorithm. In Section 2.1.2, we show how to construct a hash table for shuffled inputs achieving linear build time and constant lookup.

2.1.1 Interspersing Randomly Shuffled Arrays

Given two arrays, $\mathbf{I}_1$ and $\mathbf{I}_2$, of size n_1, n_2, respectively, where each array is randomly shuffled, our goal is to output a single array that contains all elements from $\mathbf{I}_1$ and $\mathbf{I}_2$ in a randomly shuffled order. Ignoring obliviousness, we could first initialize an output array of size $n = n_1 + n_2$, mark exactly n_1 random locations in the output array, and place the elements from $\mathbf{I}_1$ arbitrarily in these locations. The elements from $\mathbf{I}_2$ are placed in the unmarked locations.[7] The challenge is how to perform this placement obliviously, without revealing the mapping from the input array to the output array.

We observe that this routing problem is exactly the "reverse" problem of oblivious tight compaction, where one is given an input array of size n containing keys that are 1-bit and the goal is to sort the array such that all elements with key 0 appear before all elements with key 1. Intuitively, by running this algorithm "in reverse", we obtain a linear time algorithm for *obliviously* routing marked elements to an array with marked positions (that are not necessarily at the front). Since we believe that this procedure is useful in its own right, we formalize it independently and call it *oblivious distribution*. The full details appear in Section 6.

2.1.2 An Optimal Hash Table for Shuffled Inputs

In this section, we first describe a warmup construction that can be used to build a hash table in $O(n \cdot \mathsf{poly} \log\log \lambda)$ time and supports lookups in $O(\mathsf{poly} \log\log \lambda)$ time. We will then get rid of the additional $\mathsf{poly} \log\log \lambda$ factor in both the build and lookup phases.

Warmup: oblivious hash table with $\mathsf{poly} \log\log \lambda$ slack. Intuitively, to build a hash table, the idea is to randomly distribute the n elements in the input into $B := n/\mathsf{poly} \log \lambda$ bins of size $\mathsf{poly} \log \lambda$ in the clear. The distribution is done according to a pseudorandom function with some secret key K, where an element with address addr is placed in the bin with index $\mathsf{PRF}_K(\mathsf{addr})$. Whenever we lookup for a real element addr', we access the bin $\mathsf{PRF}_K(\mathsf{addr}')$; in which case, we might either find the element there (if it was originally one of the n elements in the input) or we might not find it in the accessed bin (in the case where the element is not part of the input array). Whenever we perform a dummy lookup, we just access a random bin.

Since we assume that the n balls are secretly and randomly distributed to begin with, the build procedure does not reveal the mapping from original elements to bins. However, a problem arises in the lookup phase. Since the total number of elements in each bin is revealed, accessing in the lookup phase all real keys of the input array would produce an access pattern that is identical to that of the build process, whereas accessing n dummy elements results in a new, independent balls-into-bins process of n balls into B bins.

To this end, we first throw the n balls into the B bins as before, revealing loads $n_1, \ldots, n_B$. Then, we sample new *secret* loads $L_1, \ldots, L_B$ corresponding to an independent process of throwing

[7] Note that the number of such assignments is $\binom{n}{n_1, n_2}$. Assuming that each array is already permuted, the number of possible outputs is $\binom{n}{n_1, n_2} \cdot n_1! n_2! = n!$.

$n' := n \cdot (1 - 1/\mathsf{poly}\log\lambda)$ balls into B bins. By a Chernoff bound, with overwhelming probability $L_i < n_i$ for every $i \in [B]$. We extract from each bin arbitrary $n_i - L_i$ elements obliviously and move them to an overflow pile (without revealing the L_i's). The overflow pile contains only $n/\mathsf{poly}\log\lambda$ elements so we use a standard Cuckoo hashing scheme such that it can be built in $O(m \cdot \log m) = O(n)$ time and supports lookups effectively in $O(1)$ time (ignoring the stash).[8] The crux of the security proof is showing that since the secret loads $L_1, \ldots, L_B$ are never revealed, they are large enough to mask the access pattern in the lookup phase so that it looks independent of the one leaked in the build phase.

We glossed over many technical details, the most important ones being how the bin sizes are truncated to the secret loads $L_1, \ldots, L_B$, and how each bin is being implemented. For the second question, since the bins are of $O(\mathsf{poly}\log\lambda)$ size, we support lookups using a perfectly secure ORAM constructions that can be built in $O(\mathsf{poly}\log\lambda \cdot \mathsf{poly}\log\log\lambda)$ and looked up in $O(\mathsf{poly}\log\log\lambda)$ time [14, 19] (this is essentially where our $\mathsf{poly}\log\log$ factor comes from in this warmup). The first question is slightly more tricky and here we employ our linear time tight compaction algorithm to extract the number of elements we want from each bin.

The full details of the construction appear in Section 7.

Remark 2.1 (Comparison of the warmup construction with Patel et al. [52])**.** *Our warmup construction borrows the idea of revealing loads and then sampling new secret loads from Patel et al. However, our concrete instantiation is different and this difference is crucial for the next step where we get an optimal hash table. Particularly, the construction of Patel et al. has* $\log\log\lambda$ *layers of hash tables of decreasing sizes, and one has to look for an element in each one of these hash tables, i.e., searching within* $\log\log\lambda$ *bins. In our solution, by tightening the analysis (that is, the Chernoff bound), we show that a single layer of hash tables suffices; thus, lookup accesses only a single bin. This allows us to focus on optimizing the implementation of a bin towards the optimal construction.*

Oblivious hash table with linear build time and constant lookup time. In the warmup construction, (ignoring the lookup time in the stash of the overflow pile[9]), the only super-linear operation that we have is the use of a perfectly secure ORAM, which we employ for bins of size $O(\mathsf{poly}\log\lambda)$. In this step, we replace this with a data structure with linear time build and constant time lookup: a Cuckoo hash table for lists of polylogarithmic size.

Recall that in a Cuckoo hash table each element receives two random bin choices (e.g., determined by a PRF) among a total of $\mathsf{c}_{\mathsf{cuckoo}} \cdot n$ bins where $\mathsf{c}_{\mathsf{cuckoo}} > 1$ is a suitable constant. During build-time, the goal is for all elements to choose one of the two assigned bins, such that every bin receives at most one element. At this moment it is not clear how to accomplish this build process, but suppose we can obliviously build such a Cuckoo hash table in linear time, then the problem would be solved. Specifically, once we have built such a Cuckoo hash table, lookup can be accomplished in constant time by examining both bin choices made by the element (ignoring the issue of the stash for now). Since the bin choices are (pseudo-)random, the lookup process retains security as long as each element is looked up at most once. At the end of the lookups, we can extract the unvisited elements through oblivious tight compaction in linear time — it is not hard to see that if the input array is randomly shuffled, the extracted unvisited elements appear in a random order too.

[8] We refer to Section 4.5 for background information on Cuckoo hashing.

[9] For the time being, the reader need not worry about how to perform lookup in the stash. Later, when we use our oblivious Cuckoo hashing scheme in the bigger hash table construction, we will merge the stashes of all Cuckoo hash tables into a single one and treat the merged stash specially.

Therefore the crux is how to build the Cuckoo hash table for polylogarithmically-sized, randomly shuffled input arrays. Our observation is that classical oblivious Cuckoo hash table constructions can be split into three steps: (1) assigning two possible bin choices per element, (2) assigning either one of the bins or the stash for every element, and (3) routing the elements according to the Cuckoo assignment. We delicately handle each step separately:

1. For step (1) the $n = \mathsf{poly}\log\lambda$ elements in the input array can each evaluate the PRF on its associated key, and write down its two bin choices (this takes linear time).
2. Implementing step (2) in linear time is harder as this step is dominated by a sequence of oblivious sorts. To overcome this, we use the fact that the problem size n is of size $\mathsf{poly}\log\lambda$. As a result, the index of each item and its two bin choices can be expressed using $O(\log\log\lambda)$ bits which means that a single memory word (which is $\log\lambda$ bits long) can hold $O\left(\frac{\log\lambda}{\log\log\lambda}\right)$ many elements' metadata. We can now apply a "packed sorting" type of idea [2, 13, 17, 35] where we use the RAM's word-level instructions to perform SIMD-style operations. Through this packing trick, we show that oblivious sorting and oblivious random permutation (of the elements' metadata) can be accomplished in $O(n)$ time!
3. Step (3) is classically implemented using oblivious bin distribution which again uses oblivious sorts. Here, we cannot use the packing trick since we operate on the elements themselves, so we use the fact that the input array is randomly shuffled and just route the elements in the clear.

There are many technical issues we glossed over, especially related to the fact that the Cuckoo hash tables are of size $\mathsf{c}_{\mathsf{cuckoo}} \cdot n$ bins, where $\mathsf{c}_{\mathsf{cuckoo}} > 1$. This requires us to pad the input array with dummies and later to use them to fill the empty slots in the Cuckoo assignment. Additionally, we also need to get rid of these dummies when extracting the set of unvisited element. All of these require several additional (packed) oblivious sorts or our oblivious tight compaction.

We refer the reader to Section 8 for the full details of the construction.

2.1.3 Additional Technicalities

The above description, of course, glossed over many technical details. To obtain our final ORAM construction, there are still a few concerns that have not been addressed. First, recall that we need to make sure that the unvisited elements in a hash table appear in a (pseudo-)random order such that we can make use of this residual randomness to re-initialize new hash tables faster. To guarantee this for the Cuckoo hash table that we employ for $\mathsf{poly}\log\lambda$-sized bins, we need that the underlying Cuckoo hash scheme we employ satisfy an additional property called the "indiscriminating bin assignment" property: specifically, we need that the two pseudo-random Cuckoo-bin choices for each element do not depend on the order in which they are added, their keys, or their positions in the input array. In our technical sections later, this property will allow us to do a coupling argument and prove that the residual unvisited elements in the Cuckoo hash table appear in random order.

Additionally, some technicalities remain in how we treat the smallest level of the ORAM and the stashes. The smallest level in the ORAM construction cannot use the hash table construction described earlier. This is because elements are added to the smallest level as soon as they are accessed and our hash table does not support such an insertion. We address this by using an oblivious dictionary built atop a perfectly secure ORAM for the smallest level of the ORAM. This incurs an additive $O(\mathsf{poly}\log\log\lambda)$ blowup. Finally, the stashes for each of the Cuckoo hash tables (at every level and every bin within the level) incur $O(\log\lambda)$ time. We leverage the techniques from

Kushilevitz et al. [39] to merge all stashes into a common stash of size $O(\log^2 \lambda)$, which is added to the smallest level when it is rebuilt.

On deamortization. As the overhead of our ORAM is amortized over several accesses, it is natural to ask whether we can deamortize the construction to achieve the same overhead in the worst case, per access. Historically, Ostrovsky and Shoup [50] deamortized the hierarchical ORAM of Goldreich and Ostrovsky [30], and related techniques were later applied on other hierarchical ORAM schemes [12, 33, 39]. Unfortunately, the technique fails for our ORAM as we explain below (it fails for Patel et al. [52], as well, by the same reason).

Recall that in the hierarchical ORAM, the i-th level hash table stores 2^i keys and is rebuilt every 2^i accesses. The core idea of existing deamortization techniques is to spread the rebuilding work over the next sequence of 2^i ORAM accesses. That is, copy the 2^i keys (to be rebuilt) to another working space while performing lookup on the same level i to fulfill the next 2^i accesses. However, plugging such copy-while-accessing into our ORAM, an adversary can access a key in level i right after the same level is fully copied (as the copying had no way to foresee future accesses). Then, in the adversarial eyes, the copied keys are no longer randomly shuffled, which breaks the security of the hash table (which assumes that the inputs are shuffled). Indeed, in previous works, where hash tables were secure for *every* input, such deamortization works. Deamortizing our construction is left as an open problem.

2.2 Tight Compaction

Recall that tight compaction can be considered as a restricted form of sorting, where each element in the input array receives a 1-bit key, indicating whether it is real or dummy. The goal is to move all the real elements in the array to the front obliviously, and without leaking how many elements are reals. We show a deterministic algorithm for this task.

Reduction to loose compaction. Pippenger's self-routing super-concentrator construction [55] proposes a technique that reduces the task of tight compaction to that of loose compaction. Informally speaking, loose compaction receives as input a sparse array, containing a few real elements and many dummy elements. The output is a compressed output array, containing all real elements but the procedure does not necessarily remove all the dummy elements. More concretely, we care about a specific form of loose compactor (parametrized by n): consider a suitable bipartite expander graph that has n vertices on the left and $n/2$ vertices on the right where each node has constant degree. At most $1/128$ fraction of the vertices on the left will receive a real element, and we would like to route *all* real elements over vertex-disjoint paths to the right side such that every right vertex receives at most 1 element. The crux is to find a set of satisfying routes in linear time and obliviously. Once a set of feasible routes have been identified, it is easy to see that performing the actual routing can be done obliviously in linear time (and for obliviousness we need to route a dummy element over an edge that bears 0 load). During this process, we effectively compress the sparse input array (represented by vertices on the left) by $1/2$ without losing any element.

Using Pippenger's techniques [55] and with a little extra work, we can derive the following claim — at this point we simply state the claim while deferring algorithmic details to subsequent technical sections. Below D denotes the number of bits it takes to encode an element and w denotes the word size:

Claim: There exist appropriate constants $C, C' > 6$ such that the following holds: if we can solve the aforementioned loose compaction problem obliviously in time $T(n)$ for all $n \leq n_0$, then we can

construct an oblivious algorithm that tightly compacts n elements in time $C \cdot T(n) + C' \cdot \lceil D/w \rceil \cdot n$ for all $n \leq n_0$.

As mentioned, the crux is to find satisfying routes for such a "loose compactor" bipartite graph obliviously and in linear time. Achieving this is non-trivial: for example, the recent work of Chan et al. [14] attempted to do this but their route-finding algorithm requires $O(n \log n)$ runtime — thus Chan et al. [14]'s work also implies a loose compaction algorithm that runs in time $O(n \log n + \lceil D/w \rceil \cdot n)$. To remove the extra $\log n$ factor, we introduce two new ideas, *packing*, and *decomposition* — in fact both ideas are remotely reminiscent of a line of works in the core algorithms literature on (non-comparison-based, non-oblivious) integer sorting on RAMs [2, 17, 35] but obviously we apply these techniques to a different context.

Packing: linear-time compaction for small instances. We observe that the offline route-finding phase operates only on metadata. Specifically, the route-finding phase receives the following as input: an array of n bits where the i-th bit indicates whether the i-th input position is real or dummy. If the problem size n is small, specifically, if $n \leq w/\log w$ where w denotes the width of a memory word, we can pack the entire problem into a single memory word (since each element's index can be described in $\log n$ bits). In our technical sections we will show how to rely on word-level addition and boolean operations to solve such small problem instances in $O(n)$ time. At a high level, we follow the slow route-finding algorithm by Chan et al. [14], but now within a single memory word, we can effectively perform SIMD-style operations and we exploit this to speed up Chan et al. [14]'s algorithm by a logarithmic factor for small instances.

Relying on the above Claim that allows us to go from loose to tight, we now have an $O(n)$-time oblivious *tight* compaction algorithm for small instances where $n \leq w/\log w$; specifically, if the loose compaction algorithm takes $C_0 \cdot n$ time, then the runtime of the tight compaction would be upper bounded by $C \cdot C_0 \cdot n + C' \cdot \lceil D/w \rceil \cdot n \leq C \cdot C_0 \cdot C' \cdot \lceil D/w \rceil \cdot n$.

Decomposition: bootstrapping larger instances of compaction. With this logarithmic advantage we gain in small instances, our hope is to bootstrap larger instances by decomposing larger instances into smaller ones.

Our bootstrapping is done in two steps — as we calculate below, each time we bootstrap, the constant hidden inside the $O(n)$ runtime blows up by a constant factor; thus it is important that the bootstrapping is done for only $O(1)$ times.

1. *Medium instances:* $n \leq (w/\log w)^2$. For medium instances, our idea is to divide the input array into $\sqrt{n}$ segments each of $B := \sqrt{n}$ size. As long as the input array has only $n/128$ or fewer real elements, then at most $\sqrt{n}/4$ segments can be dense, i.e., each containing more than $\sqrt{n}/4$ real elements ($1/4$ is loose but sufficient). We rely on tight compaction for small segments to move the dense segments in front of the sparse ones. For each of the $3\sqrt{n}/4$ segments, we next compress away $3/4$ of the space for using tight compaction for small instances. Clearly, the above procedure is a loose compaction and consumes at most $2 \cdot C \cdot C' \cdot C_0 \cdot \lceil D/w \rceil \cdot n + 6\lceil D/w \rceil \cdot n \leq 2.5 \cdot C \cdot C' \cdot C_0 \cdot \lceil D/w \rceil \cdot n$ runtime.

   So far we have constructed a loose compaction algorithm for medium instances. Using the aforementioned Claim, we can in turn construct an algorithm that obliviously and *tightly* compacts a medium-sized instance of size $n \leq (w/\log w)^2$ in time at most $3C^2 \cdot C' \cdot C_0 \cdot \lceil D/w \rceil \cdot n$.

2. *Large instances: arbitrary* n. We can now bootstrap to arbitrary choices of n by dividing the problem into $m := n/(\frac{w}{\log w})^2$ segments where each segment contains at most $(\frac{w}{\log w})^2$

elements. Similar to the medium case, at most $1/4$ fraction of the segments can have real density exceeding $1/4$ — which we call such segments *dense*. As before, we would like to move the dense segments in the front and the sparse ones to the end. Recall that Chan et al. [14]'s algorithm solves loose compaction for problems of arbitrary size m in time $C_1 \cdot (m \log m + \lceil D/w \rceil m)$ Thus due to the above claim we can solve tight compaction for problems of any size m in time $C \cdot C_1 \cdot (m \log m + \lceil D/w \rceil \cdot m) + C' \cdot \lceil D/w \rceil \cdot m$. Thus, in $O(\lceil D/w \rceil \cdot n)$ time we can move all the dense instances to the front and the sparse instances to the end. Finally, by invoking medium instances of tight compaction, we can compact within each segment in time that is linear in the size of the segment. This allows us to compress away $3/4$ of the space from the last $3/4$ segments which are guaranteed to be sparse. This gives us loose compaction for large instances in $O(\lceil D/w \rceil \cdot n)$ time — from here we can construct oblivious tight compaction for large instances using the above Claim.[10]

Remark 2.2. *In our formal technical sections later, we in fact directly use loose compaction for smaller problem sizes to bootstrap loose compaction for larger problem sizes (whereas in the above version we use* tight *compaction for smaller problems to bootstrap loose compaction for larger problems). The detailed algorithm is similar to the one described above: it requires slightly more complicated parameter calculation but results in better constants than the above more intuitive version.*

3 Preliminaries

Throughout this work, the security parameter is denoted λ, and it is given as input to algorithms in unary (i.e., as 1^λ). A function $\mathsf{negl}\colon \mathbb{N} \to \mathbb{R}^+$ is *negligible* if for every constant $c > 0$ there exists an integer N_c such that $\mathsf{negl}(\lambda) < \lambda^{-c}$ for all $\lambda > N_c$. Two sequences of random variables $X = \{X_\lambda\}_{\lambda \in \mathbb{N}}$ and $Y = \{Y_\lambda\}_{\lambda \in \mathbb{N}}$ are *computationally indistinguishable* if for any probabilistic polynomial-time algorithm $\mathcal{A}$, there exists a negligible function $\mathsf{negl}(\cdot)$ such that $\left|\Pr[\mathcal{A}(1^\lambda, X_\lambda) = 1] - \Pr[\mathcal{A}(1^\lambda, Y_\lambda) = 1]\right| \leq \mathsf{negl}(\lambda)$ for all $\lambda \in \mathbb{N}$. We say that $X \equiv Y$ for such two sequences if they define *identical* random variables for every $\lambda \in \mathbb{N}$. The *statistical distance* between two random variables X and Y over a finite domain Ω is defined by $\mathsf{SD}(X, Y) \triangleq \frac{1}{2} \cdot \sum_{x \in \Omega} |\Pr[X = x] - \Pr[Y = x]|$. For an integer $n \in \mathbb{N}$ we denote by $[n]$ the set $\{1, \ldots, n\}$. By $\|$ we denote the operation of string concatenation.

Definition 3.1 (Pseudorandom functions (PRFs))**.** *Let* PRF *be an efficiently computable function family indexed by keys* $\mathsf{sk} \in \{0,1\}^\lambda$*, where each* $\mathsf{PRF}_{\mathsf{sk}}$ *takes as input a value* $x \in \{0,1\}^{n(\lambda)}$ *and outputs a value* $y \in \{0,1\}^{m(\lambda)}$*. A function family* PRF *is* $\delta^{\mathcal{A}}$-secure *if for every (non-uniform) probabilistic polynomial-time algorithm* $\mathcal{A}$*, it holds that*

$$\left| \Pr_{\mathsf{sk} \leftarrow \{0,1\}^\lambda} \left[\mathcal{A}^{\mathsf{PRF}_{\mathsf{sk}}(\cdot)}(1^\lambda) = 1 \right] - \Pr_{f \leftarrow F_\lambda} \left[\mathcal{A}^{f(\cdot)}(1^\lambda) = 1 \right] \right| \leq \delta^{\mathcal{A}}(\lambda),$$

for all large enough $\lambda \in \mathbb{N}$*, where* F_λ *is the set of all functions that map* $\{0,1\}^{n(\lambda)}$ *into* $\{0,1\}^{m(\lambda)}$*.*

It is known that one-way functions are existentially equivalent to PRFs for any polynomial $n(\cdot)$ and $m(\cdot)$ and negligible $\delta^{\mathcal{A}}(\cdot)$ [36, 49]. Our construction will employ PRFs in several places and we present each part modularly with its own PRF, but note that the whole ORAM construction can be implemented with a single PRF from which we can implicitly derive all other PRFs.

[10] We omit the concrete parameter calculation in the last couple of steps but from the calculations so far, it should be obvious by now that the there is at most a constant blowup in the constants hidden inside the big-O notation.

3.1 Oblivious Machines

We define oblivious simulation of (possibly randomized) functionalities. We provide a unified framework that enables us to adopt composition theorems from secure computation literature (see, for example, Canetti and Goldreich [9, 10, 29]), and to prove constructions in a modular fashion.

Random-access machines. A RAM is an interactive Turing machine that consists of a memory and a CPU. The memory is denoted as $\mathsf{mem}[N, w]$, and is indexed by the logical address space $[N] = \{1, 2, \ldots, N\}$. We refer to each memory word also as a *block* and we use w to denote the bit-length of each block. The CPU has an internal state that consists of $O(1)$ words. The memory supports read/write instructions $(\mathsf{op}, \mathsf{addr}, \mathsf{data})$, where $\mathsf{op} \in \{\mathsf{read}, \mathsf{write}\}$, $\mathsf{addr} \in [N]$ and $\mathsf{data} \in \{0,1\}^w \cup \{\perp\}$. If $\mathsf{op} = \mathsf{read}$, then $\mathsf{data} = \perp$ and the returned value is the content of the block located in logical address addr in the memory. If $\mathsf{op} = \mathsf{write}$, then the memory data in logical address addr is updated to data. We use standard setting that $w = \Theta(\log N)$ (so a word can store an address). We follow the convention that the CPU performs one *word-level operation* per unit time, i.e., arithmetic operations (addition or subtraction), bitwise operations (AND, OR, NOT, or shift), memory accesses (read or write), or evaluating a pseudorandom function [12, 30, 32, 39, 40, 52].

Oblivious simulation of a (non-reactive) functionality. We consider machines that interact with the memory via $\mathsf{read}/\mathsf{write}$ operations. We are interested in defining sub-functionalities such as oblivious sorting, oblivious shuffling of memory contents, and more, and then define more complex primitives by composing the above. For simplicity, we assume for now that the adversary cannot see memory contents, and does not see the data field in each operation $(\mathsf{op}, \mathsf{addr}, \mathsf{data})$ that the memory receives. That is, the adversary only observes $(\mathsf{op}, \mathsf{addr})$. One can extend the constructions for the case where the adversary can also observe data using symmetric encryption in a straightforward way.

We define oblivious simulation of a RAM program. Let $f\colon \{0,1\}^* \to \{0,1\}^*$ be a (possibly randomized) functionality in the RAM model. We denote the output of f on input x to be $f(x) = y$. Oblivious simulation of f is a RAM machine M_f that interacts with the memory, has the same input/output behavior, but its access pattern to the memory can be simulated. More precisely, we let $(\mathsf{out}, \mathsf{Addrs}) \leftarrow M_f(x)$ be a pair of random variable that corresponds to the output of M_f on input x and where Addrs define the sequence of memory accesses during the execution. We say that the machine M_f *implements* the functionality f if it holds that for every input x, the distribution $f(x)$ is identical to the distribution out, where $(\mathsf{out}, \cdot) \leftarrow M_f(x)$. In terms of security, we require oblivious simulation which we formalize by requiring the existence of a simulator that simulates the distribution of Addrs without knowing x.

Definition 3.2 (Oblivious simulation)**.** *Let $f\colon \{0,1\}^* \to \{0,1\}^*$ be a functionality, and let M_f be a machine that interacts with the memory. We say that M_f* obliviously simulates the functionality *f, if there exists a probabilistic polynomial time simulator* Sim *such that for every input $x \in \{0,1\}^*$, the following holds:*

$$\left\{(\mathsf{out}, \mathsf{Addrs}) : (\mathsf{out}, \mathsf{Addrs}) \leftarrow M_f(1^\lambda, x)\right\}_\lambda \approx \left\{\left(f(x), \mathsf{Sim}(1^\lambda, 1^{|x|})\right)\right\}_\lambda .$$

Depending on whether $\approx$ refers to computational, statistical, or perfectly indistinguishability, we say M_f is computationally, statistically, or perfectly oblivious, respectively.

Later in our theorem statements, we often wish to explicitly characterize the security failure probability. Thus we also say that M_f $(1 - \delta^{\mathcal{A}})$-*obliviously simulates the functionality* f, iff no

non-uniform probabilistic polynomial-time $\mathcal{A}(1^\lambda)$ can distinguish the above joint distributions with probability more than $\delta^{\mathcal{A}}(\lambda)$ — note that the failure probability $\delta^{\mathcal{A}}$ is allowed to depend on the adversary $\mathcal{A}$'s algorithm and running time. Additionally, a 1-oblivious algorithm is also called perfectly-oblivious.

Intuitively, the above definition requires indistinguishability of the *joint* distribution of the output of the computation and the access pattern, similarly 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 Canetti and Goldreich [9,10,29]). Note that here we handle correctness and obliviousness in a single definition. As an example, consider an algorithm that randomly permutes some array in the memory, while leaking only the size of the array. Such a task should also hide the chosen permutation. As such, our definition requires that the simulation would output an access pattern that is independent of the output permutation itself.

Parametrized functionalities. In our definition, the simulator receives no input, except the security parameter and the length of the input. While this is very restricting, the simulator knows the description of the functionality and therefore also its "public" parameters. We sometimes define functionalities with explicit public inputs and refer to them as "parameters". For instance, the access pattern of a procedure for sorting of an array depends on the size of the array; a functionality that sorts an array will be parameterized by the size of the array, and this size will also be known by the simulator.

Modeling reactive functionalities. We consider functionalities that are reactive, i.e., proceed in stages, where the functionality preserves an internal state between stages. Such a reactive functionality can be described as a sequence of functions, where each function also receives as input a state, updates it, and outputs an updated state for the next function. We extend Definition 3.2 to deal with such functionalities.

We consider a reactive functionality $\mathcal{F}$ as a reactive machine, that receives commands of the form $(\mathsf{command}_i, \mathsf{inp}_i)$ and produces an output out_i, while maintaining some (secret) internal state. An implementation of the functionality $\mathcal{F}$ is defined analogously, as an interactive machine $M_{\mathcal{F}}$ that receives commands of the same form $(\mathsf{command}_i, \mathsf{inp}_i)$ and produces outputs out_i. We say that $M_{\mathcal{F}}$ is oblivious, if there exists a simulator Sim that can simulate the access pattern produced by $M_{\mathcal{F}}$ while receiving only $\mathsf{command}_i$ but not inp_i. Our simulator Sim is also a reactive machine that might maintain a state between execution.

In more detail, we consider an adversary $\mathcal{A}$ (i.e., the distinguisher or the "environment") that participates in either a real execution or an ideal one, and we require that its view in both execution is indistinguishable. The adversary $\mathcal{A}$ chooses adaptively in each stage the next command $(\mathsf{command}_i, \mathsf{inp}_i)$. In the ideal execution, the functionality $\mathcal{F}$ receives $(\mathsf{command}_i, \mathsf{inp}_i)$ and computes out_i while maintaining its secret state. The simulator is then being executed on input $\mathsf{command}_i$ and produces an access pattern Addrs_i. The adversary receives $(\mathsf{out}_i, \mathsf{Addrs}_i)$. In the real execution, the machine M receives $(\mathsf{command}_i, \mathsf{inp}_i)$ and has to produce out_i while the adversary observes the access pattern. We let $(\mathsf{out}_i, \mathsf{Addrs}_i) \leftarrow M_f(\mathsf{command}_i, \mathsf{inp}_i))$ denote the join distribution of the output and memory accesses pattern produced by M upon receiving $(\mathsf{command}_i, \mathsf{inp}_i)$ as input. The adversary can then choose the next command, as well as the next input, in an adaptive manner according to the output and access pattern it received.

Definition 3.3 (Oblivious simulation of a reactive functionality)**.** *We say that a reactive machine $M_{\mathcal{F}}$ is an oblivious implementation of the reactive functionality $\mathcal{F}$ if there exists a PPT simulator*

Sim, *such that for any non-uniform PPT (stateful) adversary* $\mathcal{A}$*, the view of the adversary* $\mathcal{A}$ *in the following two experiments* $\mathsf{Expt}^{\mathrm{real},M}_{\mathcal{A}}(1^\lambda)$ *and* $\mathsf{Expt}^{\mathrm{ideal},\mathcal{F}}_{\mathcal{A},\mathsf{Sim}}(1^\lambda)$ *is computationally indistinguishable:*

$\underline{\mathsf{Expt}^{\mathrm{real},M}_{\mathcal{A}}(1^\lambda)}$:	$\underline{\mathsf{Expt}^{\mathrm{ideal},\mathcal{F}}_{\mathcal{A},\mathsf{Sim}}(1^\lambda)}$:
Let $(\mathsf{command}_i, \mathsf{inp}_i) \leftarrow \mathcal{A}\left(1^\lambda\right)$	*Let* $(\mathsf{command}_i, \mathsf{inp}_i) \leftarrow \mathcal{A}\left(1^\lambda\right)$
Loop while $\mathsf{command}_i \neq \perp$*:*	*Loop while* $\mathsf{command}_i \neq \perp$*:*
$\mathsf{out}_i, \mathsf{Addrs}_i \leftarrow M\left(1^\lambda, \mathsf{command}_i, \mathsf{inp}_i\right)$	$\mathsf{out}_i \leftarrow \mathcal{F}(\mathsf{command}_i, \mathsf{inp}_i)$.
	$\mathsf{Addrs}_i \leftarrow \mathsf{Sim}\left(1^\lambda, \mathsf{command}_i\right)$.
$(\mathsf{command}_i, \mathsf{inp}_i) \leftarrow \mathcal{A}\left(1^\lambda, \mathsf{out}_i, \mathsf{Addrs}_i\right)$	$(\mathsf{command}_i, \mathsf{inp}_i) \leftarrow \mathcal{A}\left(1^\lambda, \mathsf{out}_i, \mathsf{Addrs}_i\right)$

Definition 3.3 can be extended in a natural way to the cases of statistical security (in which $\mathcal{A}$ is unbounded and its view in both worlds is statistically close), or perfect security ($\mathcal{A}$ is unbounded and its view is identical).

To allow our theorem statements to explicitly characterize the security failure probability, we also say that M_f $(1-\delta^{\mathcal{A}})$-*obliviously simulates the reactive functionality* $\mathcal{F}$, iff no non-uniform probabilistic polynomial-time $\mathcal{A}(1^\lambda)$ can distinguish the above joint distributions with probability more than $\delta^{\mathcal{A}}(1^\lambda)$ — note that the failure probability $\delta^{\mathcal{A}}$ is allowed to depend on the adversary $\mathcal{A}$'s algorithm and running time.

An example: ORAM. An example of a reactive functionality is an ordinary ORAM, implementing logical memory. Functionality 3.4 is a reactive functionality in which the adversary can choose the next command (i.e., either read or write) as well as the address and data according to the access pattern it has observed so far.

Functionality 3.4: $\mathcal{F}_{\mathsf{ORAM}}$

The functionality is reactive, and holds an internal state – N memory blocks, each of size w. Denote the internal state $\mathbf{X}[1,\ldots,N]$. Initially, $\mathbf{X}[\mathsf{addr}] = 0$ for every $\mathsf{addr} \in [N]$.

- $\mathsf{Access}(\mathsf{op}, \mathsf{addr}, \mathsf{data})$: where $\mathsf{op} \in \{\mathsf{read}, \mathsf{write}\}$, $\mathsf{addr} \in [N]$ and $\mathsf{data} \in \{0,1\}^w$.
  1. If $\mathsf{op} = \mathsf{read}$, set $\mathsf{data}^* := \mathbf{X}[\mathsf{addr}]$.
  2. If $\mathsf{op} = \mathsf{write}$, set $\mathbf{X}[\mathsf{addr}] := \mathsf{data}$ and $\mathsf{data}^* := \mathsf{data}$.
  3. Output data^*.

Definition 3.3 requires the existence of a simulator that on each Access command only knows that such a command occurred, and successfully simulates the access pattern produced by the real implementation. This is a strong notion of security since the adversary is adaptive and can choose the next command according to what it have seen so far.

Hybrid model and composition. We sometimes describe executions in a hybrid model. In this case, a machine M interacts with the memory via $\mathsf{read}/\mathsf{write}$-instruction and in addition can also send $\mathcal{F}$-instruction to the memory. We denote this model as $M^{\mathcal{F}}$. When invoking a function $\mathcal{F}$, we assume that it only affects the address space on which it is instructed to operate; this is achieved by first copying the relevant memory locations to a temporary position, running $\mathcal{F}$ there, and finally copying the result back. This is the same whether $\mathcal{F}$ is reactive or not. Definition 3.3 is then modified such that the access pattern Addrs_i also includes the commands sent to $\mathcal{F}$ (but not the inputs to the command). When a machine $M^{\mathcal{F}}$ obliviously implements a functionality $\mathcal{G}$ in

the $\mathcal{F}$-hybrid model, we require the existence of a simulator Sim that produces the access pattern exactly as in Definition 3.3, where here the access pattern might also contain $\mathcal{F}$-commands.

Concurrent composition follows from [10], since our simulations are universal and straight-line. Thus, if (1) some machine M obliviously simulates some functionality $\mathcal{G}$ in the $\mathcal{F}$-hybrid model, and (2) there exists a machine $M_{\mathcal{F}}$ that obliviously simulate $\mathcal{F}$ in the plain model, then there exists a machine M' that obliviously simulate $\mathcal{G}$ in the plain model.

Input assumptions. In some algorithms, we assume that the input satisfies some assumption. For instance, we might assume that the input array for some procedure is randomly shuffled or that it is sorted according to some key. We can model the input assumption $\mathcal{X}$ as an ideal functionality $\mathcal{F}_{\mathcal{X}}$ that receives the input and "rearranges" it according to the assumption $\mathcal{X}$. Since the mapping between an assumption $\mathcal{X}$ and the functionality $\mathcal{F}_{\mathcal{X}}$ is usually trivial and can be deduced from context, we do not always describe it explicitly.

We then prove statements of the form: "The algorithm A with input satisfying assumption $\mathcal{X}$ obliviously implements a functionality $\mathcal{F}$". This should be interpreted as an algorithm that receives x as input, invokes $\mathcal{F}_{\mathcal{X}}(x)$ and then invokes A on the resulting input. We require that this modified algorithm implements $\mathcal{F}$ in the $\mathcal{F}_{\mathcal{X}}$-hybrid model.

4 Oblivious Building Blocks

Our ORAM construction uses many building blocks, some of which new to this work and some of which are known from the literature. The building blocks are listed next. We advise the reader to use this section as a reference and skip it during a first read.

- **Oblivious Sorting Algorithms** (Section 4.1): We state the classical sorting network of Ajtai et al. [1] and present a *new* oblivious sorting algorithm that is more efficient in settings where each memory word can hold multiple elements.
- **Oblivious Random Permutations** (Section 4.2): We show how to perform *efficient* oblivious random permutations in settings where each memory word can hold multiple elements.
- **Oblivious Bin Placement** (Section 4.3): We state the known results for oblivious bin placement of Chan et al. [12, 15].
- **Oblivious Hashing** (Section 4.4): We present the formal functionality of a hash table that is used throughout our work. We also state the resulting parameters of a simple oblivious hash table that is achieved by compiling a non-oblivious hash table inside an existing ORAM construction.
- **Oblivious Cuckoo Hashing** (Section 4.5): We present and overview the state-of-the-art constructions of oblivious Cuckoo hash tables. We state their complexities and also make minor modifications that will be useful to us later.
- **Oblivious Dictionary** (Section 4.6): We present and analyze a simple construction of a dictionary that is achieved by compiling a non-oblivious dictionary (e.g., a red-black tree) inside an existing ORAM construction.
- **Oblivious Balls-into-Bins Sampling** (Section 4.7): We present an oblivious sampling of the approximated bin loads of throwing independently n balls into m bins, which uses the binomial sampling of Bringmann et al. [8].

4.1 Oblivious Sorting Algorithms

The elegant work of Ajtai et al. [1] shows that there is a comparator-based circuit with $O(n \cdot \log n)$ comparators that can sort any array of length n.

Theorem 4.1 (Ajtai et al. [1])**.** *There is a deterministic oblivious sorting algorithm that sorts n elements in $O(\lceil D/w \rceil \cdot n \cdot \log n)$ time where D denotes the number of bits it takes to encode a single element and w denotes the length of a word.*

Packed oblivious sort. We consider a variant of the oblivious sorting problem on a RAM, which is useful when each memory word can hold up to $B > 1$ elements. The following theorem assumes that the RAM can perform only word-level addition, subtraction, and bitwise operations in unit cost (as defined in Section 3.1).

Theorem 4.2 (Packed oblivious sort)**.** *There is a deterministic packed oblivious sorting algorithm that sorts n elements in $O\left(\frac{n}{B} \cdot \log^2 n\right)$ time, where B denotes the number of elements each memory word can pack.*

Proof. We use a variant of bitonic sort, introduced by Batcher [5]. It is well-known that, given a list of n elements, bitonic sort runs in $O\left(n \cdot \log^2 n\right)$ time. The algorithm, viewed as a sorting network, proceeds in $O\left(\log^2 n\right)$ iterations, where each iteration consists of $\frac{n}{2}$ comparators (see Figure 1). In each iteration, the comparators are totally parallelizable, but our goal is to perform the comparators *efficiently using standard word-level operation*, i.e., to perform each iteration in $O\left(\frac{n}{B}\right)$ standard word-level operations. The intuition is to pack sequentially $O(B)$ elements into each word and then apply *SIMD (single-instruction-multiple-data) comparators*, where a SIMD comparator emulates $O(B)$ standard comparators using only constant time. We show the following facts: (1) each iteration runs in $O\left(\frac{n}{B}\right)$ SIMD comparators and $O\left(\frac{n}{B}\right)$ time, and (2) each SIMD comparator can be instantiated by a constant number of word-level subtraction and bitwise operations.

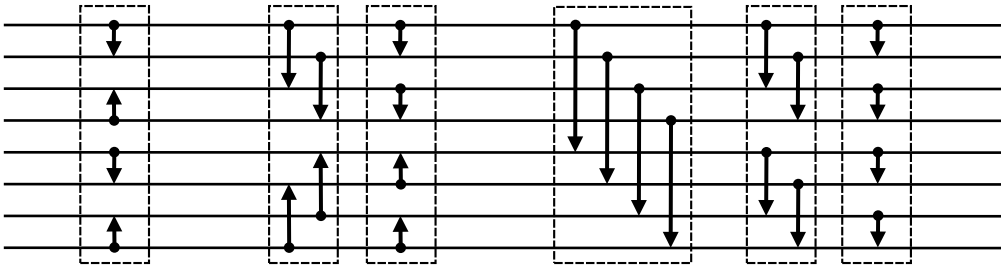

Figure 1: A bitonic sorting network for 8 inputs. Each horizontal line denotes an input from the left end and output to the right end. Each vertical arrow denotes a comparator such that compares two elements and then swaps the greater one to the pointed end. Each dashed box denotes an iteration in the algorithm. The figure is modified from [64].

To show fact (1), we first assume without loss of generality that n and B are powers of 2. We refer to the *packed array* which is the array of $\frac{n}{B}$ words, where each word stores B elements. Then, for each iteration, we want a procedure that takes as input the packed array from the previous iteration, and outputs the packed array that is processed by the comparators prescribed in the standard bitonic sort. To use SIMD comparators efficiently and correctly, for each comparator, the input pair of elements has to be *aligned* within the pair of two words. We say that two packed arrays are *aligned* if and only if the offset between each two words is the same. Hence, it suffices

to show that it takes $O(1)$ time to align $O(B)$ pairs of elements. By the definition of bitonic sort, in the same iteration, the offset between any compared pair is the same power of 2 (see Figure 1). Since B is also a power of 2, one of the following two cases holds:

(a) All comparators consider two elements from *two distinct words*, and elements are always aligned in the input.
(b) All comparators consider two elements from *the same word*, but the offset t between any compared pair is the same power of 2.

In case (a), the required alignment follows immediately. In case (b), it suffices to do the following:

1. Split one word into two words such that elements of the offset t are interleaved, where the two words are called odd and even, and then
2. Shift the even word by t elements so the comparators are aligned to the odd word.

The above procedure takes $O(1)$ time. Indeed, there are two applications of the comparators, and thus it blows up the cost of the operation by a factor of 2. Thus, the algorithm of an iteration aligns elements, applies SIMD comparators, and then reverses the alignment. Every iteration runs $O\left(\frac{n}{B}\right)$ SIMD comparators plus $O\left(\frac{n}{B}\right)$ time.

For fact (2), note that to compare k-bit strings it suffices to perform $(k+1)$-bit subtraction (and then use the sign bit to select one string). Hence, the intuition to instantiate the SIMD comparator is to use "SIMD" subtraction, which is the standard word subtraction but the packed elements are augmented by the sign bit. The procedure is as follows. Let k be the bit-length of an element such that $B \cdot k$ bits fit into one memory word. We write the B elements stored in a word as a vector $\vec{a} = (a_1, \ldots, a_B) \in (\{0,1\}^k)^B$. It suffices to show that for any $\vec{a} = (a_1, \ldots, a_B)$ and $\vec{b} = (b_1, \ldots, b_B)$ stored in two words, it is possible to compute the *mask word* $\vec{m} = (m_1, \ldots, m_B)$ such that

$$m_i = \begin{cases} 1^k & \text{if } a_i \geq b_i \\ 0^k & \text{otherwise.} \end{cases}$$

For binary strings x and y, let xy be the concatenation of x and y. Let $*$ be a wild-card bit. Assume additionally that the elements are packed with additional sign bits, i.e., $\vec{a} = (*a_1, *a_2, \ldots, *a_B)$. This can be done by simply splitting one word into two. Consider two input words $\vec{a} = (1a_1, 1a_2, \ldots, 1a_B)$ and $\vec{b} = (0b_1, 0b_2, \ldots, 0b_B)$ such that $a_i, b_i \in \{0,1\}^k$. The procedure runs as follows:

1. Let $\vec{s'} = \vec{a} - \vec{b}$, which has the format $\left(s_1 *^k, s_2 *^k, \ldots, s_B *^k\right)$, where $s_i \in \{0,1\}$ is the *sign bit* such that $s_i = 1$ iff $a_i \geq b_i$. Keep only sign bits and let $\vec{s} = \left(s_1 0^k, \ldots, s_B 0^k\right)$.
2. Shift $\vec{s}$ and get $\vec{m'} = \left(0^k s_1, \ldots, 0^k s_B\right)$. Then, the mask is $\vec{m} = \vec{s} - \vec{m'} = \left(0 s_1^k, \ldots, 0 s_B^k\right)$.

The above takes $O(1)$ subtraction and bitwise operations. This concludes the proof. □

4.2 Oblivious Random Permutations

We say that an algorithm ORP is a statistically secure oblivious random permutation, iff ORP statistically obliviously simulates the functionality $\mathcal{F}_{\text{perm}}$ which, upon receiving an input array of n elements, chooses a random permutation π from the space of all $n!$ permutations on n elements, uses π to permute the input array, and outputs the result. Note that this definition implies that not only does ORP output an almost random permutation of the input array; moreover, the access patterns of ORP is statistically close for all input arrays and all permutations. As before, we use the notation $(1-\delta)$-oblivious random permutation to explicitly denote the algorithm's failure probability δ.

Theorem 4.3. *Let $n > 100$ and let D denote the number of bits it takes to encode an element. There exists a $(1 - e^{-\sqrt{n}})$-oblivious random permutation for arrays of size n. It runs in time $O(T_{\text{sort}}^{D+\log n}(n) + n)$, where $T_{\text{sort}}^{\ell}(n)$ is an upper bound on the time it takes to sort n elements each of size ℓ bits.*

Later, in our ORAM construction, this version of ORP will be applied to arrays of size $n \geq \log^3 \lambda$, where λ is a security parameter, and thus the failure probability is bounded by a negligible function in λ.

Proof of Theorem 4.3. We apply a similar algorithm as that of Chan et al. [11, Figure 2 and Lemma 10], except with different parameters:

- Assign each element an $8 \log n$-bit random label drawn uniformly from $\{0,1\}^{8 \log n}$. Obliviously sort all elements based on their random labels, resulting in the array $\mathbf{R}$. This step takes $O(T_{\text{sort}}^{D+\log n}(n) + n)$ time.
- In one linear scan, write down two arrays: 1) an array $\mathbf{I}$ containing the indices of all elements that have collisions; and 2) an array $\mathbf{X}$ containing all the colliding elements themselves. This can be accomplished in $O(n)$ time assuming that we can leak the indices of the colliding elements.
- If the number of elements that collide is greater than $\sqrt{n}$, simply abort throwing an Overflow exception. Otherwise, use a naïve quadratic oblivious random permutation algorithm to obliviously and randomly permute the array $\mathbf{X}$, and let $\mathbf{Y}$ be the outcome. This step can be completed in $O(n)$ time where the quadratic oblivious random permutation performs the following: for each of $i \in \{1, 2, \ldots, n\}$, sample a random index r from $\{1, 2, \ldots, n - i + 1\}$, and write the i-th element of the input to the r-th unoccupied position of the output through a linear scan of the output array.
- Finally, for each $j \in |\mathbf{I}|$, write back each element $\mathbf{Y}[j]$ to the position $\mathbf{R}[\mathbf{I}[j]]$ and output the resulting $\mathbf{R}$.

To bound the probability of Overflow, we first prove the following claim:

Claim 4.4. *Let $n > 100$. Fix a subset $S \subseteq \{1, 2, \ldots, n\}$ of size $\alpha \geq 2$. Throw elements $\{1, 2, \ldots, n\}$ to n^8 bins independently and uniformly at random. The probability that every element in S has a collision with any other elements is upper bounded by $\alpha!/n^{2\alpha}$.*

Proof. If all elements in S see collisions for some sample path determined by the choice of all elements' bins denoted ψ, then the following event G^S must be true for the sample path ψ: there is a permutation S' of S such that for every $i \in \{\lceil \alpha/2 \rceil, \ldots, \alpha\}$, $S'[i]$ either collides with some element in S' whose index $j < i$ (i.e., with an element before itself) or with an element outside of S (i.e., from $[n] \setminus S$).

Therefore, the fraction of sample paths for which a fixed subset S of size α all have collision is upper bounded by the fraction of sample paths over which the above event G^S holds. Now, the fraction of sample paths over which the G^S holds is upper bounded by $\alpha! \cdot (n/n^8)^{\lfloor \alpha/2 \rfloor} \leq \alpha!/n^{2\alpha}$. □

We proceed with the proof of Theorem 4.5. The probability that there exists at least α collisions is upper bounded by the following expression since there are at most $\binom{n}{\alpha}$ possible choices for such

a subset S:

$$\binom{n}{\alpha} \cdot \frac{\alpha!}{n^{2\alpha}} = \frac{n!}{(n-\alpha)!\alpha!} \cdot \frac{\alpha!}{n^{2\alpha}} \le \frac{e\sqrt{n}(n/e)^n}{\sqrt{2\pi(n-\alpha)}((n-\alpha)/e)^{n-\alpha} \cdot \sqrt{2\pi\alpha}(\alpha/e)^\alpha} \cdot \frac{\alpha!}{n^{2\alpha}}$$
$$\le \frac{e\sqrt{n}}{2\pi} \cdot \frac{n^n}{(n-\alpha)^{n-\alpha} \cdot \alpha^\alpha} \cdot \frac{\alpha!}{n^{2\alpha}} = \frac{\alpha! \cdot e\sqrt{n}}{2\pi} \cdot \left(\frac{n}{n-\alpha}\right)^{n-\alpha} \cdot \frac{1}{\alpha^\alpha} \cdot \frac{1}{n^\alpha}$$
$$\le \frac{\alpha! \cdot e\sqrt{n}}{2\pi} \cdot \left(1 + \frac{\alpha}{n-\alpha}\right)^n \cdot \frac{1}{\alpha^\alpha} \cdot \frac{1}{n^\alpha}$$

Plugging in $\alpha = \sqrt{n}$, we can upper bound the above expression as follows assuming large $n > 100$:

$$\frac{\sqrt{n}! \cdot e\sqrt{n}}{2\pi} \cdot \left(1 + \frac{\sqrt{n}}{n-\sqrt{n}}\right)^{\sqrt{n}\cdot\sqrt{n}} \cdot \frac{1}{(n\sqrt{n})^{\sqrt{n}}} \le \frac{\sqrt{n}! \cdot e\sqrt{n}}{2\pi} \cdot \left(1 + \frac{1}{0.5\sqrt{n}}\right)^{0.5\sqrt{n}\cdot 2\cdot\sqrt{n}} \cdot \frac{1}{(n\sqrt{n})^{\sqrt{n}}}$$
$$\le \frac{\sqrt{n}! \cdot e\sqrt{n}}{2\pi} \cdot \exp(2\sqrt{n}) \cdot \frac{1}{(n\sqrt{n})^{\sqrt{n}}} \le \exp(-\sqrt{n})$$

Having bounded the Overflow probability, the obliviousness proof can be completed in identical manner to that of Lemma 10 in Chan et al. [11], since our algorithm is essentially the same as theirs but with different parameters. We stress the algorithm is oblivious even though the positions of the colliding elements are revealed. □

Packed oblivious random permutation. The following version of oblivious random permutation has good performance when each memory word is large enough to store many copies of the elements to be permuted tagged with their own indices. The algorithm follows directly by plugging in our oblivious packed sort (Theorem 4.2) into the oblivious random permutation algorithm (Theorem 4.3).

Theorem 4.5 (Packed oblivious random permutation)**.** *Let $n > 100$ and let D denote the number of bits it takes to encode an element. Let $B = \lfloor w/(\log n + D) \rfloor$ be the element capacity of each memory word and assume that $B > 1$. Then, there exists an $(1 - e^{-\sqrt{n}})$-oblivious random permutation algorithm that permutes the input array in time $O\left(\frac{n}{B} \cdot \log^2 n + n\right)$.*

Perfect oblivious random permutation. Note that the permutation of Theorem 4.3 runs in time $O(n \cdot \log n)$ but it may fail w.p. $e^{-\sqrt{n}}$. We construct a *perfectly* oblivious random permutation in this paper. This scheme comes as a by-product of our tight compaction and intersperse algorithms that we construct later in Sections 5 and 6.4.

Theorem 4.6 (Perfectly oblivious random permutation)**.** *For any n, any $m \in [n]$, suppose that sampling an integer uniformly at random from $[m]$ takes unit time. Then, there exists a perfectly oblivious random permutation such that permutes an input array of size n in $O(n \cdot \log n)$ time.*

Proof. We will prove this theorem in Section 6.4. □

4.3 Oblivious Bin Placement

Let $\mathbf{I}$ be an input array containing real and dummy elements. Each element has a tag from $\{1, \ldots, |\mathbf{I}|\} \cup \{\perp\}$. It is guaranteed that all the dummy elements are tagged with $\perp$ and all real elements are tagged with *distinct* values from $\{1, \ldots, |\mathbf{I}|\}$. The goal of *oblivious bin placement* is to create a new array $\mathbf{I}'$ of size $|\mathbf{I}|$ such that a real element that is tagged with the value i will appear

in the i-th cell of $\mathbf{I}'$. If no element was tagged with a value i, then $\mathbf{I}'[i] = \perp$. The values in the tags of real elements can be thought of as "bin assignments" where the elements want to go to and the goal of the bin placement algorithm is to route them to the right location obliviously.

Oblivious bin placement can be accomplished with $O(1)$ number of oblivious sorts (Section 4.1), where each oblivious sort operates over $O(|\mathbf{I}|)$ elements [12,15]. In fact, these works [12,15] describe a more general oblivious bin placement algorithm where the tags may not be distinct, but we only need the special case where each tag appears at most once.

4.4 Oblivious Hashing

An oblivious (static) hashing scheme is a data structure that supports three operations Build, Lookup, and Extract that realizes the following (ideal) reactive functionality. The Build procedure is the constructor and it creates an in-memory data structure from an input array $\mathbf{I}$ containing real and dummy elements where each real element is a (key, value) pair. It is assumed that all real elements in $\mathbf{I}$ have distinct keys. The Lookup procedure allows a requestor to look up the value of a key. A special symbol $\perp$ is returned if the key is not found or if $\perp$ is the requested key. We say a (key, value) pair is *visited* if the key was searched for and found before. Finally, Extract is the destructor and it returns a list containing unvisited elements padded with dummies to the same length as the input array $\mathbf{I}$.

An important property that our construction relies on is that if the input array $\mathbf{I}$ is randomly shuffled to begin with (with a secret permutation), the outcome of Extract is also randomly shuffled (in the eyes of the adversary). In addition, we need obliviousness to hold only when the Lookup sequence is non-recurrent, i.e., the same real key is never requested twice (but dummy keys can be looked up multiple times). The functionality is formally given next.

Functionality 4.7: $\mathcal{F}_{\mathsf{HT}}^n$ – Hash Table Functionality for Non-Recurrent Lookups

- $\mathcal{F}_{\mathsf{HT}}^n$.Build($\mathbf{I}$):
  - **Input:** an array $\mathbf{I} = (a_i, \ldots, a_n)$ containing n elements, where each a_i is either dummy or a (key, value) pair denoted $(k_i, v_i) \in \{0,1\}^D \times \{0,1\}^D$.
  - **Assumption**: throughout the paper, we assume that both the key and the value can be stored in $O(1)$ memory words, i.e., $D = O(w)$ where w denotes the word size.
  - **The procedure:**
    1. Initialize the state state to $(\mathbf{I}, \mathbf{P})$, where $\mathbf{P} = \emptyset$.
  - **Output:** The Build operation has no output.

- $\mathcal{F}_{\mathsf{HT}}^n$.Lookup($k$):
  - **Input:** The procedure receives as input a key k (that might be $\perp$, i.e., dummy).
  - **The procedure:**
    1. Parse the internal state as state $= (\mathbf{I}, \mathbf{P})$.
    2. If $k \in \mathbf{P}$ (i.e., k is a recurrent lookup) then halt and output fail.
    3. If $k = \perp$ or $k \notin \mathbf{I}$, then set $v^* = \perp$.
    4. Otherwise, set $v^* = v$, where v is the value that corresponds to the key k in $\mathbf{I}$.
    5. Update $\mathbf{P} = \mathbf{P} \cup \{(k, v)\}$.
  - **Output:** The element v^*.

- $\mathcal{F}^n_{\mathsf{HT}}.\mathsf{Extract}()$:
  - **Input:** The procedure has no input.
  - **The procedure:**
    1. Parse the internal state $\mathsf{state} = (\mathbf{I}, \mathbf{P})$.
    2. Define an array $\mathbf{I}' = (a'_1, \ldots, a'_n)$ as follows. For $i \in [n]$, set $a'_i = a_i$ if $a_i = (k, v) \notin \mathbf{P}$. Otherwise, set $a'_i = \mathsf{dummy}$.
    3. Shuffle $\mathbf{I}'$ uniformly at random.
  - **Ouptut:** The array $\mathbf{I}'$.

Construction of naïveHT. A naïve, perfectly secure oblivious hashing scheme can be obtained directly [14, 19] from a perfectly secure ORAM construction [14, 19]. Both schemes [14, 19] are Las Vegas algorithms: for any capacity n, it almost always takes $O(\log^3 n)$ time to serve a request — however with negligible in n probability, it may take longer to serve a request. We stress that although the runtime may sometimes exceed the stated bound, there is never any security or correctness failure in the known perfectly secure ORAM constructions [14, 19]. We observe that the scheme of Chan et al. [14] is a Las Vegas algorithm only because the oblivious random permutation they employ is a Las Vegas algorithm. In this paper, we actually construct a perfect oblivious random permutation that runs in $O(n \cdot \log n)$ time with probability 1 (Theorem 4.6). Thus, we can replace the oblivious random permutation in Chan et al. [14] with our own Theorem 4.6. Interestingly, this results in *the first non-trivial perfectly oblivious RAM that is not a Las Vegas algorithm.*

Theorem 4.8 (Perfect ORAM (using [14] + Theorem 4.6))**.** *For any capacity $n \in \mathbb{N}$, there is a perfect ORAM scheme that consumes space $O(n)$ and worst-case time overhead $O\left(\log^3 n\right)$ per request.*

To construct naïveHT using perfectly secure ORAM scheme, we use Theorem 4.8 to compile a standard, balanced binary search tree data structure (e.g., a red-black tree). Finally, Extract can be performed in linear time if we adopt the perfect ORAM of Theorem 4.8 which incurs constant space blowup. In more detail, we flatten the entire in-memory data structure into a single array, and apply oblivious tight compaction (Theorem 1.2) on the array, moving all the real elements to the front. We then truncate the array at length $|\mathbf{I}|$, apply a perfectly random permutation on the truncated array, and output the result. This gives the following construction.

Theorem 4.9 (naïveHT)**.** *Assume that each memory word is large enough to store at least $\Theta(\log n)$ bits where n is an upper bound on the total number of elements that exist in the data structure. There exists a perfectly secure, oblivious hashing scheme that consumes $O(n)$ space; further,*

- Build *and* Extract *each consumes $n \cdot \mathsf{poly}\log n$ time;*
- *Each* Lookup *request consumes $\mathsf{poly}\log n$ time.*

Later in our paper, whenever we need an oblivious hashing scheme for a small ($\mathsf{poly}\log(\lambda)$-sized) bin, we will adopt naïveHT since it is *perfectly secure*. In comparison, schemes whose failure probability is negligible in the problem size ($\mathsf{poly}\log(\lambda)$ in this case) may not yield $\mathsf{negl}(\lambda)$ failure probability. Indeed, almost all known computationally secure [28, 30, 32, 39] or statistically secure [57, 60, 62] ORAM schemes have a (statistical) failure probability that is negligible in the problem's size and are thus unsuited for small, poly-logarithmically sized bins. In a similar vein, earlier works also employed perfectly secure ORAM schemes to treat poly-logarithmic size inputs [57].

4.5 Oblivious Cuckoo Hashing

A Cuckoo hashing scheme [51] is a hashing method with constant lookup cost (ignoring the stash). Imagine that we wish to hash n balls into a table of size $\mathsf{c}_{\mathsf{cuckoo}} \cdot n$, where $\mathsf{c}_{\mathsf{cuckoo}} > 1$ is an appropriate fixed constant. Additionally, there is a stash denoted S of size s for holding a small number of overflowing balls. We also refer to each position of the table as a bin, and a bin can hold exactly one ball. Each ball receives two independent bin choices. During the build phase, we execute a Cuckoo assignment algorithm that picks either a bin-choice for each ball among its two specified choices, or assigns the ball to some position in the stash. It must hold that no two balls are assigned to the same location either in the main table or in the stash. Kirsch et al. [38] showed an assignment algorithm that succeeds with with probability $1 - n^{-\Omega(s)}$ over the random bin choices, where s denotes the stash size.

Without privacy, it is known that such an assignment can be computed in $O(n)$ time. However, it is also known that the standard procedure for building a Cuckoo hash table leaks information through the algorithm's access patterns [12, 32, 59]. Goodrich and Mitzenmacher [32] (see also the recent work of Chan et al. [12][11]) showed that a Cuckoo hash table can be built *obliviously* in $O(n \cdot \log n)$ total time. In our ORAM construction, we will need to apply Chan et al. [12]'s oblivious Cuckoo hashing techniques in a non-blackbox fashion to enable asymptotically more efficient hashing schemes for randomly shuffled input arrays. Below, we present the necessary preliminaries.

4.5.1 Build Phase: Oblivious Cuckoo Assignment

To obliviously build a Cuckoo hash table given an input array, we have two phases: 1) a *metadata* phase in which we select a bin among the two bin choices made by each input ball or alternatively assign the ball to a position in the stash; and 2) the actual (oblivious) routing of the balls into their destined location in the resulting hash-table data structure. The problem solved by the the first phase (i.e., the metadata step), is called the Cuckoo assignment problem, formally defined as below.

Oblivious Cuckoo assignment. Let n be the number of balls to be put into the Cuckoo hash table, let $\mathbf{I} = ((u_1, v_1), \ldots (u_n, v_n))$ be the array of the two bin choices made by each of the n balls, where $u_i, v_i \in [\mathsf{c}_{\mathsf{cuckoo}} \cdot n]$ for $i \in [n]$. In the *Cuckoo assignment* problem, given such an input array $\mathbf{I}$, the goal is to output an array $\mathbf{A} = \{a_1, \ldots a_n\}$, where $a_i \in \{\mathtt{bin}(u_i), \mathtt{bin}(v_i), \mathtt{stash}(j)\}$ denotes that the i-th ball is assigned either to bin u_i or bin v_i, or to the j-th position in the stash. We say that a Cuckoo assignment $\mathbf{A}$ is *correct* iff it holds that (i) each bin and each position in the stash receives at most one ball, and (ii) the number of balls in the stash is bounded by a parameter s.

Given a correct assignment $\mathbf{A}$, a Cuckoo hash table can be built by obliviously placing the balls into the position it is assigned too. A straightforward way to accomplish this is through a standard oblivious bin placement algorithm (Section 4.3)[12].

Theorem 4.10 (Oblivious Cuckoo assignment [12, 32])**.** *Let* $\mathsf{c}_{\mathsf{cuckoo}} > 1$ *be a suitable constant,* $\delta > 0$*,* $n \in \mathbb{N}$*, the stash size* $s \geq \log(1/\delta)/\log n$*, and let* $n_{\mathsf{cuckoo}} := \mathsf{c}_{\mathsf{cuckoo}} \cdot n + s$ *and* $\ell := 8 \log_2(n_{\mathsf{cuckoo}})$*.*

[11] Chan et al. [12] is a re-exposition and slight rectification of the elegant ideas of Goodrich and Mitzenmacher [32]; also note that the Cuckoo hashing appears only in the full version of Chan et al., http://eprint.iacr.org/2017/924.

[12] The description here differs slightly from previous works [12]. In previous works, the Cuckoo assignment $\mathbf{A}$ was allowed to depend not only on the two bin choices $\mathbf{I}$, but also on the balls and keys themselves. In our work, the fact that the Cuckoo assignment is only a function of $\mathbf{I}$ is crucial – see Remark 4.13 for a discussion on this property that we call *indiscrimination*.

Then, there is a deterministic oblivious algorithm denoted $\mathsf{cuckooAssign}$ *that successfully finds a Cuckoo assignment problem with probability* $1 - O(\delta) - \exp\left(-\Omega\left(\frac{n^{5/7}}{\log^4(1/\delta)}\right)\right)$ *over the choice of the random bins* $\mathbf{I}$*. It runs in time* $O\left(n_{\mathsf{cuckoo}} + T^{\ell}_{\mathrm{sort}}(n_{\mathsf{cuckoo}}) + \log\frac{1}{\delta} \cdot T^{\ell}_{\mathrm{sort}}(n^{6/7})\right)$*, where* $T^{\ell}_{\mathrm{sort}}(m)$ *is the time bound for obliviously sorting* m *elements each of length* ℓ*.*

As a special case, suppose that $n \geq \log^8(1/\delta)$*,* $s = \log(1/\delta)/\log n$*, and the word size* $w \geq \Omega(\log n)$*, using the AKS oblivious sorting algorithm (Theorem 4.1), the algorithm runs in time* $O(n \cdot \log n)$ *with success probability* $1 - O(\delta)$*.*

In fact, to obtain the above theorem, we cannot directly apply Chan et al. [12]'s Cuckoo hash-table build algorithm, but need to make some minor modifications. We refer the reader to Remark 4.13 and Appendix B for more details.

A variant of the Cuckoo assignment problem. Later, we will need a variant of the above Cuckoo assignment problem. Imagine that the input array is now of size exactly the same as the total size consumed by the Cuckoo hash-table, i.e., $n_{\mathsf{cuckoo}} := \mathsf{c}_{\mathsf{cuckoo}} \cdot n + s$ where s denotes the stash size; but importantly, at most n balls in the input array are real and the remaining are dummy. Therefore, we may imagine that the input array to the Cuckoo assignment problem is the following metadata array containing either real or dummy bin choices: $\mathbf{I} = \{(u_i, v_i)\}_{i \in [n_{\mathsf{cuckoo}}]}$ where $u_i, v_i \in [n_{\mathsf{cuckoo}}]$ if i corresponds to a real ball and $u_i = v_i = \perp$ if i corresponds to a dummy ball.

We would like to compute a correct Cuckoo assignment for all real balls in the input. This variant can easily be solved as follows: 1) obliviously sort the input array such that the upto n real balls' bin choices are in the front — let $\mathbf{X}$ denote the outcome; 2) apply the aforementioned $\mathsf{cuckooAssign}$ algorithm to $\mathbf{X}[1:n]$, resulting in an output assignment array denoted $\mathbf{A}$ of length n; 3) pad $\mathbf{A}$ to length n_{cuckoo} by adding $n_{\mathsf{cuckoo}} - n$ dummy assignment labels of appropriate length resulting in the array $\mathbf{A}'$ and 4) reverse route $\mathbf{A}'$ back to the input array — this can be accomplished if in step 1 we remembered the per-gate routing decisions in the sorting network. Henceforth, we refer to this slight variant as $\overline{\mathsf{cuckooAssign}}$. Note that Steps 1 and 4 of the above algorithm require oblivious sorting for elements of at most $\ell := 8\log_2(\mathsf{c}_{\mathsf{cuckoo}} \cdot n)$ bits.

Corollary 4.11 (Oblivious Cuckoo assignment variant)**.** *Suppose that* $\delta > 0$*,* $n \geq \log^8(1/\delta)$*, the stash size* $s \geq \log(1/\delta)/\log n$*. Then, there is a deterministic, oblivious algorithm denoted* $\overline{\mathsf{cuckooAssign}}$ *that successfully finds a Cuckoo assignment for the above variant of the problem with probability* $1 - O(\delta)$*, consuming the same asymptotical runtime as Theorem 4.10.*

Later, when we apply Corollary 4.11, if a single memory word can pack $B > 1$ elements of length $\ell := 8\log_2(\mathsf{c}_{\mathsf{cuckoo}} \cdot n)$ — this happens when the Cuckoo hash-table's size is small — we may use packed oblivious sorting to instantiate the sorting algorithm. This gives rise to the following corollary:

Corollary 4.12 (Packed oblivious Cuckoo assignment)**.** *Suppose that* $\delta > 0$*,* $n \geq \log^8(1/\delta)$*, the stash size* $s \geq \log(1/\delta)/\log n$*, and let* $\ell := 8\log_2(\mathsf{c}_{\mathsf{cuckoo}} \cdot n + s)$*. Then, there is a deterministic oblivious algorithm running in time* $O(n_{\mathsf{cuckoo}} + (n_{\mathsf{cuckoo}}/w) \cdot \log^3 n_{\mathsf{cuckoo}})$ *that successfully finds a Cuckoo assignment (for both of the above variants) with probability* $1 - O(\delta)$ *over the choice of the random bins, where* w *denotes the word size.*

Although we state the above corollary for general n, we only gain in efficiency when we apply it to the case of large word size w and small n.

Proof. Let $n_{\mathsf{cuckoo}} = \mathsf{c}_{\mathsf{cuckoo}} \cdot n + s$. If $\ell = \log n_{\mathsf{cuckoo}} \leq w$, we use packed oblivious sorting to instantiate all the oblivious sorting in the $\mathsf{cuckooAssign}$ or $\overline{\mathsf{cuckooAssign}}$ algorithms. By Theorem 4.2,

the running time is upper bounded by $O(n_{\mathsf{cuckoo}} + (n_{\mathsf{cuckoo}}/(w/\log n_{\mathsf{cuckoo}})) \cdot \log^2 n_{\mathsf{cuckoo}} + \log(1/\delta) \cdot \left(n_{\mathsf{cuckoo}}^{6/7}/(w/\log n_{\mathsf{cuckoo}})\right) \cdot \log^2 n_{\mathsf{cuckoo}}) \leq O(n_{\mathsf{cuckoo}} + (n_{\mathsf{cuckoo}}/w) \cdot \log^3 n_{\mathsf{cuckoo}}))$ where the last inequality relies on the fact that $n \geq \log^8(1/\delta)$. Finally, it is not hard to see that the theorem also holds for $\ell \geq w$ too — in this case, we can use the normal AKS sorting network to instantiate the oblivious sorting. □

Remark 4.13 (Indiscriminate hashing)**.** *As mentioned earlier, to get Theorem 4.10, Corollary 4.11, and Corollary 4.12, we cannot directly apply Chan et al. [12]'s oblivious Cuckoo hash-table building algorithm. In particular, their algorithm does not explicitly separate the metadata phase from the actual ball-routing phase; consequently, in their algorithm, the final assignment computed may depend on the element's key, and not just the bin-choice metadata array* $\mathbf{I}$*. In our paper, we need an extra indiscrimination property: after the hash-table is built, the location of any real ball is fully determined by its relative index in the input array as well as the bin-choice metadata array* $\mathbf{I}$*. This property will be needed to prove an extra property for our oblivious hashing schemes, that is, if the input array is randomly shuffled, then all unvisited elements in the hash-table data structure must appear in random order. Note that the way we formulated the Cuckoo assignment problem automatically ensures this indiscriminate property. See Appendix B for details, where we describe a variant of the algorithm of Chan et al. [12] that satisfies our needs.*

4.5.2 Oblivious Cuckoo Hashing

We get an oblivious Cuckoo hashing scheme as follows:

- To perform Build, use the aforementioned $\mathsf{cuckooAssign}$ algorithm, while determining element's (k, v) two bin choices by evaluating a pseudorandom function $\mathsf{PRF}_{\mathsf{sk}}(k)$ where sk is a secret key sampled freshly for this hash-table instance, and stored inside the CPU.
- For Lookup of key k, we evaluate the element's two bin choices using the PRF, and look up the corresponding two bins in the hash-table. Besides these two bins, we also need to scan through the stash (no matter whether the element is found in one of the two bins). After an element has been looked up, it will be marked as removed.
- Finally, to realize $\mathsf{Extract}$ we obliviously shuffle all unvisited elements using a perfect oblivious random permutation (Theorem 4.6) and output the resulting array.

We have the following theorem:

Theorem 4.14 ($\mathsf{cuckooHT}$)**.** *Assume a* $\delta^{\mathcal{A}}_{\mathsf{PRF}}$*-secure PRF. For any* $\delta > 0, n \geq \log^8(1/\delta)$*, there is an oblivious hashing scheme denoted* $\mathsf{cuckooHT}^{\delta,n} = (\mathsf{Build}, \mathsf{Lookup}, \mathsf{Extract})$ *which* $(1 - O(\delta) - \delta^{\mathcal{A}}_{\mathsf{PRF}})$*-obliviously simulates* $\mathcal{F}^n_{\mathsf{HT}}$*. Moreover, the algorithm satisfies the following properties:*

- Build *takes as input* $\mathbf{I}$ *of length* n*, and outputs a Cuckoo table* T *of size* $O(n)$ *and a stash* S *of size* $O(\log(1/\delta)/\log n)$*. It requires* $O(n \cdot \log n)$ *time.*
- Lookup *requires looking up only* $O(1)$ *positions in the table* T *which takes* $O(1)$ *time, and making a linear scan of the stash* S *consuming* $O(\log(1/\delta)/\log n)$ *time.*
- $\mathsf{Extract}$ *performs a perfect oblivious random permutation, consuming* $O(n \cdot \log n)$ *time.*

Remark 4.15. *This above scheme is different from the one of Chan et al. [12] in three aspects: 1) we explicitly separate the assignment phase from the ball-routing phase during* Build*; 2) we satisfy the indiscriminate property mentioned in Remark 4.13, and 3) we additionally support* $\mathsf{Extract}$ *(whereas Chan et al. do not).*

4.6 Oblivious Dictionary

As opposed to the oblivious hash table from Section 4.4, which is a *static* data structure, an oblivious dictionary is an extension of oblivious hashing, which allows to add only one element at a time into the structure using an algorithm Insert, where Insert is called at most n times for a predetermined capacity n. Also, the dictionary supports Lookup and Extract procedures as described in oblivious hashing. Note that there is no specific order in which Insert and Lookup requests have to be made and they could be mixed arbitrarily. Another difference between our hashing notion and the dictionary notion is that the Extract operation outputs all elements, including "visited" elements (while Extract of oblivious hashing outputs only "unvisited" elements). In summary, an oblivious dictionary realizes Functionality 4.16 described below.

Functionality 4.16: $\mathcal{F}^n_{\mathsf{Dict}}$ – Dictionary Functionality

- $\mathcal{F}^n_{\mathsf{Dict}}.\mathsf{Init}()$:
  - **Input:** The procedure has no input.
  - **The procedure:**
    1. Allocate an empty set S and an empty table T.
  - **Output:** The operation has no output.

- $\mathcal{F}^n_{\mathsf{Dict}}.\mathsf{Insert}(k, v)$:
  - **Input:** A key-value pair denoted (k, v).
  - **The procedure:**
    1. If $|S| < n$, add k to the set S and set $T[k] = v$.
  - **Output:** The operation has no output.

- $\mathcal{F}^n_{\mathsf{Dict}}.\mathsf{Lookup}(k)$:
  - **Input:** The procedure receives as input a key k (that might be $\perp$, i.e., dummy).
  - **The procedure:**
    1. Initialize $v^* := \perp$.
    2. If $q \in S$, set $v^* := T[q]$.
  - **Output:** The element v^*.

- $\mathcal{F}^n_{\mathsf{Dict}}.\mathsf{Extract}()$:
  - **Input:** The procedure has no input.
  - **The procedure:**
    1. Initialize an empty array L.
    2. Iterate over S and for each $k \in S$, add $(k, T[k])$ to L.
    3. Pad L to be of size n.
    4. Randomly shuffle L and denote the output by L'.
  - **Output:** The array L'.

Corollary 4.17 (Perfectly secure oblivious dictionary)**.** *For any capacity $n \in \mathbb{N}$, there exists a perfectly-oblivious dictionary* $(\mathsf{Init}, \mathsf{Insert}, \mathsf{Lookup}, \mathsf{Extract})$ *such that the time of* Init *and* $\mathsf{Extract}$ *is* $O\left(n \cdot \log^3 n\right), O\left(n \cdot \log^3 n\right)$*, respectively, the time of* $\mathsf{Insert}, \mathsf{Lookup}$ *are both* $O\left(\log^4 n\right)$*.*

Proof. The realization of the oblivious dictionary is very similar to the $\mathsf{naïveHT}$. Without security, the functionalities can be realized in $O(n)$ or $O(\log n)$ time using a standard, balanced binary search tree data structure (e.g., red-black tree) and the standard linear-time Fisher-Yates shuffle [23]. To achieve obliviousness, it suffices to compile the algorithms and the data structure using the perfect ORAM of Theorem 4.8, which is perfectly-oblivious and incurs $O(\log^3 n)$ overhead per access. □

4.7 Oblivious Balls-into-Bins Sampling

Consider the ideal functionality $\mathcal{F}_{n,m}^{\mathsf{throw\text{-}balls}}$ that throws n balls into m bins uniformly at random and outputs the bin loads. A non-oblivious algorithm for this functionality will throw each ball independently at random and will run in ime $O(n)$. To achieve obliviousness, we need to be able to sample binomials.

Let $\mathsf{Binomial}(n, p)$ be the binomial distribution parameterized by n independent trial with success probability p. Let $\mathcal{F}^{\mathsf{binomial}}$ be an ideal functionality that samples from $\mathsf{Binomial}(n, 1/2)$ and outputs the result. The standard way to implement $\mathcal{F}^{\mathsf{binomial}}$ is to toss n independent coins, but this takes time $O(n)$. Since this is too expensive for our purposes, we settle for an approximation using an algorithm of Bringmann et al. [8] (see also [20]).

Theorem 4.18 (Sampling Binomial Variables [8, Theorem 5])**.** *Assume word RAM arithmetic, logical operations, and sampling a uniformly random word takes $O(1)$ time. For any $n = 2^{O(w)}$, there is a $(1 - n \cdot \delta)$-oblivious RAM algorithm* $\mathsf{SampleApproxBinomial}_\delta$ *that implements the functionality* $\mathcal{F}^{\mathsf{binomial}}$ *in time $O\left(\log^5(1/\delta)\right)$.*

Here is our implementation of $\mathcal{F}^{\mathsf{throw\text{-}balls}}$ using $\mathsf{SampleApproxBinomial}_\delta$.

Algorithm 4.19: $\mathsf{SampleBinLoad}_{m,\delta}(n)$

- **Input:** a secret number of balls $n \in \mathbb{N}$.
- **Public parameters:** the number of bins $m \in \mathbb{N}$, which is a power of 2.
- **The Algorithm:**
  1. (Base case.) If $m = 1$, output n. Otherwise, continue with the following.
  2. Sample a binomial random variable $X \leftarrow \mathsf{SampleApproxBinomial}_\delta(n)$, where X is the total number of balls in the first $m/2$ bins.
  3. Recursively call $\mathsf{SampleBinLoad}_{m/2,\delta}(X)$ and $\mathsf{SampleBinLoad}_{m/2,\delta}(n - X)$, let L_1, L_2 be the results.
  4. Output the concatenated array $L_1 \| L_2$.

If we use $\delta = 0$ in the above algorithm, then $\mathsf{SampleBinLoad}$ perfectly and obliviously implements $\mathcal{F}^{\mathsf{throw\text{-}balls}}$. Using the efficient algorithm for sampling approximated binomials (Theorem 4.18), we get the following theorem.

Theorem 4.20. *For any integer $n = 2^{O(w)}$, m a power of 2,* $\mathsf{SampleBinLoad}_{m,\delta}$ *$(1 - m \cdot n \cdot \delta)$-obliviously implements the functionality $\mathcal{F}_{n,m}^{\mathsf{throw\text{-}balls}}$ in time $O(m \cdot \log^5(1/\delta))$.*

5 Oblivious Tight Compaction

In this section, we describe a deterministic linear-time procedure (in the balls and bins model) which solves the tight compaction problem: given an input array containing n balls each of which marked with a 1-bit label that is either 0 or 1, output a permutation of the input array such that all the 1 balls are moved to the front of the array.

Theorem 5.1 (Restatement of Theorems 1.2)**.** *There exists a deterministic oblivious tight compaction algorithm that takes $O(\lceil D/w \rceil \cdot n)$ time to compact any input array of n elements each can be encoded using D bits, where w is the word size.*

Our approach extends to oblivious *distribution*: Given an array containing n balls and an assignment array A of n bits such that each ball is marked with a 1-bit label that is either 0 or 1 and the number of 0-balls equals the number of 0-bits in A, output a permutation of the input balls such that all the 0-balls are moved to the positions of 0-bits in A. See Section 5.3 for details.

A bipartite expander. Our construction relies on bipartite expander graphs where the entire edge set can be computed in linear time in the number of nodes.

Theorem 5.2. *For any constant $\epsilon \in (0,1)$, there exists a family of bipartite graphs $\{G_{\epsilon,n}\}_{n\in\mathbb{N}}$ and a constant $d_\epsilon \in \mathbb{N}$, such that for every $n \in \mathbb{N}$ being a power of 2, $G_{\epsilon,n} = (L, R, E)$ has $|L| = |R| = n$ vertices on each side, it is d_ϵ-regular, and for every sets $S \subseteq L, T \subseteq R$, it holds that*

$$\left| e(S,T) - \frac{d_\epsilon}{n} \cdot |S| \cdot |T| \right| \leq \epsilon \cdot d_\epsilon \cdot \sqrt{|S| \cdot |T|},$$

where $e(S,T)$ is the set of edges $(s,t) \in E$ such that $s \in S$ and $t \in T$.

Furthermore, there exists a (uniform) linear-time algorithm that on input 1^n outputs the entire edge set of $G_{\epsilon,n}$.

Such graphs are well known (c.f. Margulis [47] and Pippenger [55]) and we provide a proof in Appendix C.1 for completeness. Note that the property that the entire edge set can be computed in linear time is crucial for us (but to the best of our knowledge has not been exploited before).

5.1 Reducing Tight Compaction to Loose Compaction

We first reduce the problem of tight compaction in linear time to loose compaction in linear time. A loose compaction algorithm is parametrized by a sufficiently large constant $\ell > 2$ (which will be chosen in Section 5.2), and the input is an array $\mathbf{I}$ of size n that has *real* and *dummy* balls. It is guaranteed that the number of reals is at most n/ℓ. The expected output of the procedure is an array of size $n/2$ that contains all the real balls.

From SwapMisplaced to TightCompaction. The first observation for our tight compaction algorithm is that some balls already reside in the correct place, and only some balls have to be moved. In fact, the number of 0-balls that are "misplaced" equals exactly the number of 1-balls that are misplaced. Specifically, assume that there are c balls marked 0; all 1 balls in the subarray $\mathbf{I}[1,\ldots,c]$ are misplaced, and all 0 balls in $\mathbf{I}[c+1,\ldots,n]$ are also misplaced. Notice that the number of misplaced 0 balls equals the number of misplaced 1 balls. Therefore, we have reduced the problem of tight compaction to the problem of swapping misplaced 0 balls with the misplaced 1 balls. This reduction is described as Algorithm 5.3.

Algorithm 5.3: TightCompaction($\mathbf{I}$)**:**

- **Input:** an array $\mathbf{I}$ of n balls, each ball is labeled as 0 or 1.
- **The algorithm:**
  1. Count the number of 0-balls in $\mathbf{I}$, let c be the number.
  2. For $i = 1, 2, \ldots, n$, do the following.
     (a) If $\mathbf{I}[i]$ is a 1-ball and $i \leq c$, mark $\mathbf{I}[i]$ as blue.
     (b) If $\mathbf{I}[i]$ is a 0-ball and $i > c$, mark $\mathbf{I}[i]$ as red.
     (c) Otherwise, mark $\mathbf{I}[i]$ as $\perp$.
  3. Run SwapMisplaced($\mathbf{I}$), let $\mathbf{O}$ be the result.
- **Output:** The array $\mathbf{O}$.

From LooseSwapMisplaced and LooseCompaction to SwapMisplaced. In SwapMisplaced, we are given n balls, each is labeled as either red, blue or $\perp$. It is guaranteed that the number of blue balls equals the number of red balls. Our goal is to obliviously swap the locations of the blue balls with the red balls. To implement SwapMisplaced we use two subroutines, $\mathsf{LooseCompaction}_\ell$ and $\mathsf{LooseSwapMisplaced}_\ell$, parametrized with a number $\ell > 2$ that have the following input-output guarantees:

- The algorithm $\mathsf{LooseCompaction}_\ell$ receives as input an array $\mathbf{I}$ consisting of n balls, where at most $1/\ell$ fraction are real and the rest are dummies. The output is an array of size $n/2$ that contains all the real balls. We implement this procedure in Section 5.2.
- The algorithm $\mathsf{LooseSwapMisplaced}_\ell$ receives the same input as SwapMisplaced: n balls, each is labeled as either red, blue or $\perp$, and the number of blues equals the number of reds. This procedure swaps the locations of all the red-blue balls except $1/\ell$ fraction. All the swapped balls are labeled with $\perp$. We implement this procedure below in this subsection.

Using these two procedures, SwapMisplaced works by first running $\mathsf{LooseSwapMisplaced}_\ell$ which makes all the necessary swaps except for at most $1/\ell$ fraction. We then perform $\mathsf{LooseCompaction}_\ell$ on the resulting array, moving all the remaining red and blue balls to the first half of the array. Then, we continue recursively and perform SwapMisplaced on the first half of the array. To be able to facilitate the recursion, we record the original placement of the balls and their movements, and revert them in the end. Given a linear time algorithm for $\mathsf{LooseCompaction}_\ell$ and $\mathsf{LooseSwapMisplaced}_\ell$ (that we will achieve below), the recursive formula for the running time of the algorithm is $T(n) = T(n/2) + O(n)$, and therefore is linear. The description of SwapMisplaced is given in Algorithm 5.5 and we have the following claim.

Claim 5.4. *Let* $\mathbf{I}$ *be any input where the number of balls marked* red *equals the number of balls marked* blue*, and let* $\mathbf{O} = \mathsf{SwapMisplaced}(\mathbf{I})$*. Then,* $\mathbf{O}$ *is a permutation of the balls in* $\mathbf{I}$*, where each* red *ball in* $\mathbf{I}$ *swaps its position with one* blue *ball in* $\mathbf{I}$*. Moreover, the runtime of* $\mathsf{SwapMisplaced}(\mathbf{I})$ *is linear in* $|\mathbf{I}|$*.*

Algorithm 5.5: SwapMisplaced($\mathbf{I}$)

- **Input:** an array $\mathbf{I}$ of n balls, each ball is labeled as red, blue, or $\perp$, where the number of balls marked red equals the number of balls marked blue.
- **The algorithm:**

1. Run $\mathsf{LooseSwapMisplaced}_\ell(\mathbf{I})$, let $\mathbf{I}'$ be the result.
2. Replace all balls marked as $\perp$ in $\mathbf{I}'$ with dummies. Run $\mathsf{LooseCompaction}_\ell(\mathbf{I}')$ while relating to all balls marked with blue or red as real. Let $\mathbf{I}_{\text{real}}$ be the result of the procedure, where $|\mathbf{I}_{\text{real}}| = n/2$, and record all moves in an array Aux.
   At the end of this step, we are guaranteed that all blue and red balls appear in $\mathbf{I}_{\text{real}}$.
3. Run SwapMisplaced recursively on $\mathbf{I}_{\text{real}}$ and let $\mathbf{I}'_{\text{real}}$ be the result. In $\mathbf{I}'_{\text{real}}$, every red ball is swapped with some blue ball and vise versa.
4. Reverse route of all real balls from $\mathbf{I}'_{\text{real}}$ to $\mathbf{I}'$ using Aux, let $\mathbf{O}$ be the resulting array after such routing.

- **Output:** The array $\mathbf{O}$.

Implementing $\mathsf{LooseSwapMisplaced}_\ell$. The access pattern of the algorithm is determined by a (deterministically generated) expander graph $G_{\epsilon,n} = (L, R, E)$, where the allowed swaps are vertices of distance 2. That is, we interpret $L = R = [n]$; if two vertices $i, k \in R$ have some common neighbor $j \in L$, and $\mathbf{I}[i], \mathbf{I}[k]$ are both marked with different colors, then we swap them and change their mark to $\perp$. Choosing the expansion parameters of the graph appropriately guarantees that after performing these swaps, there are at most n/ℓ misplaced balls. As the graph is d-regular, there are at most $\binom{d}{2} \cdot n$ neighbors of distance 2, and since $d = O(1)$, the total running time is $O(n)$.

Algorithm 5.6: $\mathsf{LooseSwapMisplaced}_\ell(\mathbf{I})$

- **Input:** An array $\mathbf{I}$ of n balls, each ball is labeled as red, blue or $\perp$. The number of balls marked as red equals the number of balls marked blue.
- **Parameters:** A parameter $\ell \in \mathbb{N}$.
- **The algorithm:**
  1. Generate a bipartite graph $G_{\epsilon,n} = (L, R, E)$ with vertex set $L = R = [n]$ such that $\epsilon \leq \frac{1}{2\sqrt{\ell}}$ (see Theorem 5.2).
  2. For $j = 1, \ldots, n$, do:
     (a) For all edges $(j, i) \in E$ and $(j, k) \in E$ do the following: If $(\mathbf{I}[i], \mathbf{I}[k])$ are marked (blue, red) or (red, blue), then swap between $\mathbf{I}[i]$ and $\mathbf{I}[k]$. Mark both as $\perp$. Otherwise, perform dummy swap.
- **Output:** The array $\mathbf{I}$

Claim 5.7. *Let $\mathbf{I}$ be an input array in which the number of balls marked* blue *equals the number of balls marked* red*. Denote as $\mathbf{O}$ the output array. Then, $\mathbf{O}$ is a swap of the input array:*

- *There exist pairs of indices $(i_1, j_1), \ldots, (i_k, j_k)$ all distinct such that the following holds: For every $\ell \in [k]$, $\mathbf{I}[i_\ell], \mathbf{I}[j_\ell]$ are marked with different colors ((red, blue) or (blue, red)), and $\mathbf{O}[i_\ell] = \mathbf{I}[j_\ell]$, $\mathbf{O}[j_\ell] = \mathbf{I}[i_\ell]$ and both $\mathbf{O}[i_\ell], \mathbf{O}[j_\ell]$ are marked $\perp$.*
- *For every $i \notin \{i_1, \ldots, i_k, j_1, \ldots, j_k\}$ then $\mathbf{O}[i] = \mathbf{I}[i]$ and both have the same mark.*

In $\mathbf{O}$, the number of balls marked red *equals the number of balls marked* blue*, and there are at most $1/\ell$ fraction of marked balls.*

Proof. The algorithm only performs swaps between red and blue balls and therefore $\mathbf{O}$ is a permutation of $\mathbf{I}$, the three conditions hold, and the number of balls marked red equals the number of balls marked blue. It remain to show that the number of red/blue balls in $\mathbf{O}$ is at most n/ℓ. In the end of the execution of the algorithm, let R_{red} be the set of all vertices in R that are marked red, and let R_{blue} be the set of vertices in R that are marked blue. Then, it must be that $\Gamma(R_{\mathsf{red}}) \cap \Gamma(R_{\mathsf{blue}}) = \emptyset$, as otherwise the algorithm would have swapped an element in R_{red} with an element in R_{blue}. Since the number of balls in R_{red} and in R_{blue} is equal, it suffices to show that for every subset $R' \subset R$ of size greater than $n/(2\ell)$, it holds that $|\Gamma(R')| > n/2$. This implies that $|R_{\mathsf{red}}| = |R_{\mathsf{blue}}| \leq n/(2\ell)$, as otherwise $\Gamma(R_{\mathsf{red}}) \cap \Gamma(R_{\mathsf{blue}}) \neq \emptyset$. The fact that every set of vertices is expanding follows generically by the equivalence between spectral expansion (the definition of expanders we use) and vertex expansion. We give a direct proof below.

Let $R' \subset R$ with $|R'| > n/2\ell$ and let $L' = \Gamma(R')$ be its set of neighbors. Since the graph is d_ϵ-regular for some $d_\epsilon \in O(1)$, it holds that $e(L', R') = d_\epsilon \cdot |R'|$. Thus, by the guarantee on the expander graph (Theorem 5.2) and by $\epsilon \leq \frac{1}{2\sqrt{\ell}}$, it holds that

$$d_\epsilon \cdot \left|R'\right| = e(L', R') \leq \frac{d_\epsilon \left|L'\right| \left|R'\right|}{n} + \frac{d_\epsilon}{2\sqrt{\ell}} \cdot \sqrt{\left|L'\right| \left|R'\right|}.$$

Dividing by $d_\epsilon \cdot |R'|$ and rearranging, we get

$$1 - \frac{|L'|}{n} \leq \sqrt{\frac{|L'|}{4\ell \cdot |R'|}}.$$

Since $|R'| > n/(2\ell)$, we have

$$1 - \frac{|L'|}{n} < \sqrt{\frac{|L'|}{2n}}.$$

Solving the above by squaring and rearranging, $\left(1 - \frac{|L'|}{n}\right)^2 - \frac{|L'|}{2n} < 0$, we have $|L'| > n/2$. □

5.2 Loose Compaction

In Section 5.2.1, we describe the algorithm CompactionFromMatching – compacting an array given the required matching (via "folding"). In Section 5.2.2, we show how to compute the matching, both for the case where m is "big" (SlowMatch) and when m is "small" (FastMatch). In Section 5.2.3, we present the full loose compaction algorithm.

5.2.1 Compaction from Matching

We show that with the appropriate notion of matching (given below), one can "fold" an array A, with density of real balls being small enough, such that all the real balls reside in the output array of size $n/2$.

Definition 5.8 ($(B, B/4)$-Matching)**.** *Let $G = (L, R, E)$ be a bipartite graph, and let $S \subseteq L$ and $M \subseteq E$. Given any vertex $u \in L \cup R$, define $\Gamma_M(u) := \{v \in L \cup R \mid (u, v) \in M\}$ as the subset of neighboring vertices in M. We say that M is a $(B, B/4)$-matching for S, iff (i) for every $u \in S$, $|\Gamma_M(u)| \geq B$, and; (ii) for every $v \in R$, $|\Gamma_M(v)| \leq B/4$.*

In Algorithm 5.9, we show how to compact an array given a $(B, B/4)$-matching for the set of all *dense* bins S, where a bin is said to be dense if it contains more than $B/4$ real balls. We assume that the matching itself is given to us via an algorithm $\mathsf{ComputeMatching}_G(S)$. The implementation of

$\mathsf{ComputeMatching}_G(\cdot)$ is given in Section 5.2.2. Note that for any problem size $m < 2B$, it suffices to perform oblivious sorting (e.g., Theorem 4.1) instead of the following algorithm as B is a constant.

Algorithm 5.9: $\mathsf{CompactionFromMatching}_D(A)$

- **Public parameters:** D is the size of each ball in bits.
- **Input:** An array $\mathbf{I}$ of m balls, in which at most $m/128$ are real.
- **The Procedure:**
  1. Let $\epsilon = 1/64$ and let d_ϵ be the constant regularity of the graph $G_{\epsilon,\star}$ guaranteed by Theorem 5.2, and set $B = d_\epsilon/2$.
  2. Interpret the array $\mathbf{I}$ as m/B bins, where each bin consists of B balls. Let S be the set of indexes i such that $\mathbf{I}[i]$ consists of more than $B/4$ real balls (the "dense" bins in $\mathbf{I}$). Let $\mathbf{I}'$ be an array of m/B empty bins, where the capacity of a bin is B balls.
  3. Let $G_{\epsilon,m/B} = (L, R, E)$ be the d_ϵ-regular bipartite graph guaranteed by Theorem 5.2, where $|L| = |R| = m/B$.
  4. Compute a $(B, B/4)$-matching for S via $M \leftarrow \mathsf{ComputeMatching}_{G_{\epsilon,m/B}}(S)$.
  5. **Distribute:** For all $i \in |E|$, get the edge $(u, v) = E[i]$ (where $u \in L, v \in R$).
     (a) If $M[i] = 1$, move a ball from bin $\mathbf{I}[u]$ to bin $\mathbf{I}'[v]$.
     (b) If $M[i] = 0$, access $\mathbf{I}[u]$ and $\mathbf{I}'[v]$ but move nothing.
  6. **Fold:** Let $\mathbf{O}$ be an array of size $m/(2B)$ empty bins, each of capacity of B balls. For $i \in [m/(2B)]$, move all real balls in $(\mathbf{I}[i], \mathbf{I}[m/(2B) + i])$, $(\mathbf{I}'[i], \mathbf{I}'[m/2B + i])$ to bin $\mathbf{O}[i]$, pad $\mathbf{O}[i]$ with dummy balls if there are less than B real balls.
- **Output:** The array $\mathbf{O}$.

Given that $|E| = O(m)$ and B is a constant, the running time is linear in $\lceil D/w \rceil \cdot m$. Also, there are at most $\frac{m}{B} \cdot \frac{1}{32}$ dense bins as the total number of real balls is at most $\frac{m}{128}$ (i.e., $|S| \leq \frac{m}{32B}$), so M is a $(B, B/4)$-matching by $\mathsf{ComputeMatching}$, as we will show later in Claim 5.16. Hence, correctness holds. As for correctness, the $(B, B/4)$ matching M guarantees that every vertex in the right vertices of G contains at most $B/4$ real elements. As a result, after the distribute phase, all bins in the entire graph contain at most $B/4$ real elements, and we can fold the array without having any overflow. The following claim is immediate.

Claim 5.10. *Let $\mathbf{I}$ be an array of m balls, where each ball is of size D bits, and where at most $m/128$ balls are marked real. Then, the output of $\mathsf{CompactionFromMatching}_D(\mathbf{I})$ is an array of $m/2$ balls that contains all real balls in $\mathbf{I}$. The running time of the algorithm is $O(\lceil D/w \rceil \cdot m)$ plus the running time of $\mathsf{ComputeMatching}_{G_{\epsilon,m/B}}$.*

5.2.2 Computing the Matching

To compute the matching, we have two cases to consider, depending on the size of the input, and each algorithm results in different running time.

Algorithm 5.11: $\mathsf{ComputeMatching}_{G_{\epsilon,m/B}}(S)$

- **Public parameters:** $\epsilon = \frac{1}{64}$, B, m and a graph $G_{\epsilon,m/B} = (L, R, E)$.
- **Input:** a set $S \subset L$ such that $|S| \leq \frac{m}{32B}$.
- **Procedure:**

1. If $\frac{m}{B} > \frac{w}{\log w}$, then let $M \leftarrow \mathsf{SlowMatch}_{G_{\epsilon,m/B}}(S)$.
2. If $\frac{m}{B} \leq \frac{w}{\log w}$, then let $M \leftarrow \mathsf{FastMatch}_{G_{\epsilon,m/B}}(S)$.

- **Output:** M.

Case I: SlowMatch $\left(\frac{m}{B} > \frac{w}{\log w}\right)$. We transform the non-oblivious algorithm described in the overview, that runs in time $O(m)$, into an oblivious algorithm, by performing fake accesses. This results in an algorithm that requires $O(m \cdot \log m)$ time [14]. The **bolded** instructions in $\mathsf{SlowMatch}_{G_{\epsilon,m/B}}(S)$ are the ones where we pay the extra $\log m$ factor in efficiency; these accesses will be avoided in Case II (FastMatch).

Algorithm 5.12: $\mathsf{SlowMatch}_{G_{\epsilon,m/B}}(S)$

- **Public parameters:** $\epsilon = \frac{1}{64}$, B, m and a graph $G_{\epsilon,m/B} = (L, R, E)$.
- **Input:** a set $S \subset L$ such that $|S| \leq \frac{m}{32B}$ and $\frac{m}{B} > \frac{w}{\log w}$.
- **The procedure:**
  1. Let M be a bit-array of length $|E|$ initialized to all 0s.
  2. Let $L' = S$ be the "dense" vertices in L, and let $R' = \Gamma(L')$: initialize R' as an array of m/B 0s; for every edge $(u, v) \in E$, if $L'[u] = 1$ then assign $R'[v] = 1$. Note that each L', R' is stored as an array of m/B indicators.
  3. Repeat the following for $\log(m/B)$ iterations.
     (a) For each vertex $u \in L$, if $u \in L'$, send one *request* to every neighboring vertex $v \in \Gamma(u)$. **(If $u \notin L'$, perform fake accesses)**
     (b) For each vertex $v \in R$, if $v \in R'$, do the following **(perform fake accesses if $v \in R \setminus R'$)**:
        i. Let the number of received requests be c.
        ii. If $c \leq B/4$, then reply *positive* to every request. Otherwise, reply *negative* to every request.
     (c) For each vertex $u \in L$, if $u \in L'$, do the following **(perform fake accesses to $u \in L \setminus L'$).**
        i. Let the number of received positives be c.
        ii. If $c \geq B$, then add to M every edge that replied positive, and remove vertex u from L'. **(Otherwise, perform fake accesses to M).**
     (d) Recompute $R' = \Gamma(L')$ from the updated L'.
- **Output:** The array M.

In the following claim we show that the size of L' decreases by a constant factor in every iteration which implies that the algorithm finishes after $\log(m/B)$ iterations. This means that SlowMatch outputs a correct $(B, B/4)$-matching for S in time $O(m \cdot \log m)$ (see Claim 5.14).

Claim 5.13. *Let $S \subset L$ such that $|S| \leq m/(32B)$, and let $\epsilon = \frac{1}{64}$. In each iteration of Algorithm 5.12, the number of unsatisfied vertices $|L'|$ decreases by a factor of 2.*

Proof. Let L' be the set of unsatisfied vertices at beginning of any given round, let $R'_{\mathsf{neg}} \subseteq R' \subseteq \Gamma(L')$ be the set of neighbors such that reply *negative*, and let $m' = m/B$. Then, $e(L', R'_{\mathsf{neg}}) > |R'_{\mathsf{neg}}| \cdot B/4$. From the expansion property in Theorem 5.2, we obtain $|R'_{\mathsf{neg}}| \cdot B/4 < e(L', R'_{\mathsf{neg}}) \leq d_\epsilon |L'| \left|R'_{\mathsf{neg}}\right| / m' + \epsilon d_\epsilon \sqrt{|L'| \left|R'_{\mathsf{neg}}\right|}$; dividing by $\left|R'_{\mathsf{neg}}\right| d_\epsilon$ and rearranging this becomes $\epsilon \sqrt{|L'| / \left|R'_{\mathsf{neg}}\right|} > B/(4d_\epsilon) - |L'|/m'$. We chose B as the largest power of 2 that is no larger than $d_\epsilon/2$, and so $B/d_\epsilon > 1/4$. Since $|L'|/m' \leq 1/32$ (recall that L' is initially S, and the number of dense vertices, $|S|$, is at most $m/(32B)$), and since $\epsilon = \frac{1}{64}$, we have that:

$$\sqrt{|L'| / \left|R'_{\mathsf{neg}}\right|} > \frac{1}{\epsilon} \cdot \frac{B}{4d_\epsilon} - \frac{1}{\epsilon} \cdot \frac{|L'|}{m'} > \frac{1}{\epsilon} \cdot \left(\frac{1}{16} - \frac{1}{32}\right),$$

then $\sqrt{|L'| / \left|R'_{\mathsf{neg}}\right|} \geq 64/16 - 64/32$, i.e., $\left|R'_{\mathsf{neg}}\right| \leq |L'|/4$.

We conclude that the number of vertices in R' that reply negatively is at most $|L'|/4$. As L' has $d_\epsilon |L'|$ outgoing edges, and R'_{neg} has at most $d_\epsilon \left|R'_{\mathsf{neg}}\right| \leq d_\epsilon |L'|/4$ incoming edges, at most one quarter of edges in L' lead to R'_{neg} and yield a negative reply. Since $d_\epsilon = B/2$ and every vertex in L' sends d_ϵ requests and all negative are from R'_{neg}, there are at most $d_\epsilon |L'|/4$ negative replies, and therefore at most $|L'|/2$ nodes in L' get more than $B = d_\epsilon/2$ negatives. We conclude that at least $|L'|/2$ nodes become satisfied. □

Claim 5.14. *Let $S \subset L$ such that $|S| \leq m/(32B)$, and let $\epsilon = \frac{1}{64}$. Then, $\mathsf{SlowMatch}_{G_{\epsilon,m/B}}$ takes as input S, runs obliviously in time $O(m \cdot \log m)$, and outputs a $(B, B/4)$-matching for S.*

Proof. The runtime follows by there are $\log \frac{m}{B}$ iterations and each iteration takes $O(m)$ time. Obliviousness follows since the access pattern is a deterministic function of the graph $G_{\epsilon,m/B}$, which depends only on the parameter m (but not on the input S). We argue correctness next. By Step 3(c)ii, every removal of vertex u from L' has at least B edges in the output M. Also, the edge added to M must have a vertex in R' that has at most $B/4$ requests at Step 3(b)ii. Observing that the set of received requests at Step 3(b)ii is non-increasing over iterations, it follows that for every $v \in R$, $\Gamma_M(v) \leq B/4$. By Claim 5.13, after $\log(m/B)$ iterations, we have $L' = \emptyset$, and hence every $u \in S$ has at least B edges in M. □

Case II: FastMatch $\left(\frac{m}{B} \leq \frac{w}{\log w}\right)$. Here, we improve the running time by relying on the fact that for instances where m is really small (as above), the number of words needed to encode the whole graph is really small (i.e., constant). While obliviousness is obtained again by accessing the whole graph, this time it will be much cheaper as we will be able to read the whole graph while accessing a small number of words. The algorithm is described in Algorithm 5.15. Note that whenever we write, e.g., "access $\mathsf{Reply}[v]$", we actually access the whole array Reply as it is $O(1)$ words, and do not reveal which vertex we actually access. Thus, each iteration takes time $O(|L'| + |R'|)$, and since the size of $|L'|$ is reduced by a factor of 2 in each iteration (and since $|R'| \leq d_\epsilon \cdot |L'|$), we perform $O(m)$ time in total. Additionally, note that we store the set L' as a set (i.e., a list) and not as a bit vector, as we cannot afford the $O(m)$ time required to run over all balls of L and then ask whether the element is in L'.

Algorithm 5.15: $\mathsf{FastMatch}_{G_{\epsilon,m/B}}(S)$

- **Public parameters:** $\epsilon = \frac{1}{64}$, B, m and a graph $G_{\epsilon,m/B} = (L, R, E)$.
- **Input:** a set $S \subset L$ such that $|S| \leq \frac{m}{32B}$ and $\frac{m}{B} \leq \frac{w}{\log w}$.
- **The procedure:**

1. Represent $G_{\epsilon,m/B}$ and initialize internal variables as follows:
   (a) Represent $L = R = \{0, \ldots, m/B - 1\}$, where each identifier requires $\log w$ bits. The set of edges E is represented as d_ϵ arrays $E_1, \ldots, E_{d_\epsilon}$. For a given node $u \in L$, let $v_1, \ldots, v_{d_\epsilon} \in R$ be its set of neighbors. We write $E_1[u] = v_1, \ldots, E_{d_\epsilon}[u] = v_{d_\epsilon}$. Each array E_i can be stored in a single word.
   (b) Initialize an array of counters $\mathsf{ctr} = (\mathsf{ctr}[0], \ldots, \mathsf{ctr}[m/B - 1])$, i.e., a counter for every $v \in R$. Initialize an array of lists $\mathsf{Req} = (\mathsf{Req}[0], \ldots, \mathsf{Req}[m/B - 1])$, each item $\mathsf{Req}[v]$ for $v \in R$ is a list of at most $B/4$ identifiers of nodes in L that sent a request. Initialize an array of lists $\mathsf{Reply} = (\mathsf{Reply}[0], \ldots, \mathsf{Reply}[m/B - 1])$, where each item $\mathsf{Reply}[v]$ for $v \in L$ is a list of at most d_ϵ identifiers of nodes in R that replied positively to a request. Observe that $\mathsf{ctr}, \mathsf{Req}$ and Reply requires $O(1)$ words.
   (c) Given the set S of marked nodes in L, we put all identifiers in a single word L'. The set R' is also a set of identifiers. Initially, R' is the set of all neighbors of L'. The set M is an array of $|E|$ indicators.
2. Repeat the following for $i = \log(m/B), \ldots, 1$ iterations.
   (a) Initialize $\mathsf{ctr} = 0$ (note that a single access initializes the whole array).
   (b) For each vertex $u \in L'$: (in the i-th iteration, make exactly 2^i accesses; i.e., perform fake accesses to L' if necessary)
      i. Let $(v_1, \ldots, v_{d_\epsilon}) = (E_1[u], \ldots, E_{d_\epsilon}[u])$. Append u to the lists $\mathsf{Req}[v_1], \ldots, \mathsf{Req}[v_{d_\epsilon}]$.
      ii. Increment the counters $\mathsf{ctr}[v_1], \ldots, \mathsf{ctr}[v_{d_\epsilon}]$.
   (c) For each $v \in R'$, do the following: (in the i-th iteration, make exactly $2^i \cdot d_\epsilon$ accesses; i.e., make fake accesses to R' if necessary)
      i. Access $\mathsf{ctr}[v]$. If $\mathsf{ctr}[v] \leq B/4$, then iterate over all identifiers $\mathsf{Req}[v]$ and for each node $u \in \mathsf{Req}[v]$ add v to $\mathsf{Reply}[u]$. This corresponds to answering *positive* to requests.
   (d) For each vertex $u \in L'$, do the following (perform 2^i total accesses).
      i. Let the number of received positive be c.
      ii. If $c \geq B$, then add to M every edge that replied positive, and remove vertex u from L'. Otherwise, do nothing.
   (e) Recompute $R' = \Gamma(L')$ from the updated L', and initialize again the counters.

- **Output:** The array M.

Claim 5.16. *Given the public d_ϵ-regular bipartite graph $G_{\epsilon,m/B} = (L, R, E)$ such that $B = d_\epsilon/2$, let $S \subseteq L$ be a set such that $|S| \leq \frac{m}{32B}$. Then, $\mathsf{ComputeMatching}_{G_{\epsilon,m/B}}(S)$ outputs a $(B, B/4)$-matching for S, where the running time is $O(m)$ if $\frac{m}{B} \leq \frac{w}{\log w}$, and $O(m \cdot \log m)$ otherwise.*

Proof. The correctness follows from Claim 5.14 and the time bound follows from the time of $\mathsf{FastMatch}$ and $\mathsf{SlowMatch}$ in both cases. □

5.2.3 Oblivious Loose Compaction

By combining Claim 5.16 and Claim 5.10 we get the following corollary.

Corollary 5.17. *The total running time of $\mathsf{CompactionFromMatching}_D$ on an array of m elements each of size D bits is $O(\lceil D/w \rceil \cdot m)$ if $m \leq \frac{w}{\log w}$, and $O(\lceil D/w \rceil \cdot m + m \cdot \log m)$ otherwise.*

The next algorithm shows how to compute $\mathsf{LooseCompaction}_\ell(\mathbf{I})$ for any input array $\mathbf{I}$ in which there are at most $|\mathbf{I}|/\ell$ elements that are marked.

Algorithm 5.18: $\mathsf{LooseCompaction}_\ell(\mathbf{I})$

- **Public parameters:** Size of input array n, maximum acceptable sparsity ℓ $(= 2^{38})$.
- **Input:** An array $\mathbf{I}$ with n balls each of size D bits, where at most n/ℓ balls are real and the rest are dummy.
- **The procedure:** Let $p := \frac{w}{\log w}$. We have three cases:

  Case I: Compaction for big arrays, i.e., $n \geq p^2$:

  1. Let $\mu = p^2$. Represent $\mathbf{I}$ as another array A that consists of n/μ blocks: for each $i \in [n/\mu]$, let $A[i]$ be the block consists of all balls $\mathbf{I}[(i-1)\cdot\mu+1], \ldots, \mathbf{I}[i\cdot\mu]$.
  2. For each $i \in [n/\mu]$, label $A[i]$ as *dense* if $A[i]$ consists of more than $\mu/\sqrt{\ell}$ real balls.
  3. Run $\mathbf{O}_1 = \mathsf{CompactionFromMatching}_{\mu\cdot D}(A)$. $\mathbf{O}_1$ is of size $n/2$, and consists of all dense blocks in A.
  4. Repeat the above process, this time on the array $\mathbf{O}_1$: interpret it as $n/2\mu$ blocks, mark dense blocks as before, and let $\mathbf{O}'_1 = \mathsf{CompactionFromMatching}_{\mu\cdot D}(\mathbf{O}_1)$. $\mathbf{O}'_1$ is of size $n/4$.
  5. Replace all dense blocks in A with dummy blocks. For every $i \in [n/\mu]$, run $\mathbf{O}_{2,i} \leftarrow \mathsf{LooseCompaction}_{\sqrt{\ell}/2}(A[i])$, and then run again $\mathbf{O}'_{2,i} \leftarrow \mathsf{LooseCompaction}_{\sqrt{\ell}/2}(\mathbf{O}_{2,i})$. Note that $|A[i]| = \mu$ and $\left|\mathbf{O}'_{2,i}\right| = \mu/4$.
  6. **Output:** $\mathbf{O}'_1 \| \mathbf{O}'_{2,1} \| \ldots \| \mathbf{O}'_{2,n/\mu}$ (which is of total size $n/2$, as $|\mathbf{O}'_1| = n/4$ and $\sum_{i=1}^{n/\mu} \left|\mathbf{O}'_{2,i}\right| = n/4$).

  Case II: Compaction for moderate arrays, i.e., $p \leq n < p^2$:

  Similar to Case I, where this time we work with $\mu = p$ instead of p^2.

  Case III: Compaction for small arrays, i.e., $n < p$:

  1. Run $\mathbf{O} = \mathsf{CompactionFromMatching}_D(A)$.
  2. **Output:** $\mathbf{O}$.

Theorem 5.19. *Let $\ell = 2^{38}$. For any input array $\mathbf{I}$ with n balls such that each ball consists of D bits and with at most n/ℓ real balls, the procedure $\mathsf{LooseCompaction}_\ell(\mathbf{I})$ outputs an array of size $n/2$ consisting of all real balls in $\mathbf{I}$, and runs in time $O(\lceil D/w \rceil \cdot n)$.*

Proof. To show the correctness, we check that in all cases, both $\mathsf{CompactionFromMatching}$ and $\mathsf{LooseCompaction}_\ell$ are called with an input array $\mathbf{I}$ such that at most $|\mathbf{I}|/128$ balls are real so that the correctness is implied by Claim 5.10 (henceforth referred to as the 128-condition). We proceed by checking the 128-condition for each case in $\mathsf{LooseCompaction}_\ell$. Note that $2^{38} = (2\cdot(2\cdot 128)^2)^2$.[13]

- **Case I:** It is always called with $\ell = 2^{38}$. At Step 2, note that the number of dense blocks is at most $\frac{n}{\mu} \cdot \frac{1}{\sqrt{\ell}}$ as the total number of real balls is at most $\frac{n}{\ell}$. Hence, the two $\mathsf{CompactionFromMatching}$ takes as input at most $(\frac{n}{\mu})/\sqrt{\ell}$ and then $(\frac{n}{2\mu})/(\sqrt{\ell}/2)$ dense blocks,

[13] For readability, we recurse with $\sqrt{\ell}$ instead of optimizing the constants (e.g., 2^{38}). Compared to the non-oblivious routing of Pippenger [55], our constant is much larger, and it is an open problem to resolve this issue or prove it is inherent.

and the 128-condition holds as $\sqrt{\ell}/2 = (2 \cdot 128)^2 > 128$. The two $\mathsf{LooseCompaction}_{\sqrt{\ell}/2}$ takes at most $\mu/\sqrt{\ell}$ and $(\frac{\mu}{2})/(\sqrt{\ell}/2)$ real balls as each block is not dense, and then the 128-condition holds for $\sqrt{\ell}$ and $\sqrt{\ell}/2$ similarly.

- **Case II:** It can be either called directly with $\ell = 2^{38}$, or called indirectly from Case I with $\ell = (2 \cdot 128)^2$. Hence, the number of real is at most $n/(2 \cdot 128)^2$. By the same calculation as Case I, the two $\mathsf{CompactionFromMatching}$ and two $\mathsf{LooseCompaction}_{\sqrt{\ell}/2}$ take an input array such that the sparsity is at least $\sqrt{(2 \cdot 128)^2}/2 = 128$, and the 128-condition holds.
- **Case III:** Similar to Case II, it can be called directly, indirectly from Case I, or indirectly from Case II with $\ell = 2^{38}, (2 \cdot 128)^2$, or 128 respectively. Hence, the given sparsity ℓ is at least 128, and hence the 128-condition holds directly for $\mathsf{CompactionFromMatching}$.

To show the time complexity, observe that, except for $\mathsf{CompactionFromMatching}$, all other procedures run in time $O(n)$. By Corollary 5.17, running $\mathsf{CompactionFromMatching}$ on m items of size D bits takes $O(\lceil D/w \rceil \cdot m)$ time if $m \leq \frac{w}{\log w}$, or $O(\lceil D/w \rceil \cdot m + m \cdot \log m)$ otherwise. For any n, the depth of the recursive call to $\mathsf{LooseCompaction}$ is at most 2. Hence, it suffices to show that in each case, every $\mathsf{CompactionFromMatching}$ run in $O(\lceil D/w \rceil \cdot n)$ time. We proceed from Case III back to I.

- **Case III:** Since $n < \frac{w}{\log w}$, the $\mathsf{CompactionFromMatching}_D$ takes an input size $m = n < \frac{w}{\log w}$ runs in $O(\lceil D/w \rceil \cdot n)$ time by Corollary 5.17.
- **Case II:** Given that $n < \left(\frac{w}{\log w}\right)^2$, the subsequent $\mathsf{CompactionFromMatching}_{D \cdot p}$ takes input size $m = \frac{n}{p} < \frac{w}{\log w}$. Hence, its running time is $O(\lceil D \cdot p/w \rceil \cdot m) = (\lceil D/w \rceil \cdot n)$, as in Case III.
- **Case I:** For arbitrary n, the subsequent invocation of $\mathsf{CompactionFromMatching}_{D \cdot p^2}$, in Steps 3 and 4, takes an input size $m = n/p^2$ and then $m/2$. By Corollary 5.17, in both cases, the procedure runs in time $O(\lceil D \cdot p^2/w \rceil \cdot m + m \cdot \log m) = O(\lceil D/w \rceil \cdot n + (n/p^2) \cdot \log n)$. Since Case I is the starting point of the algorithm, by the standard RAM model, we have that $w = \Omega(\log n)$, which implies that $n/p^2 = O(n/\log n)$ as $p = w/\log w$. Thus, the total time is bounded by $O(\lceil D/w \rceil \cdot n)$.

□

Plugging $\mathsf{LooseCompaction}_{2^{38}}$ into $\mathsf{SwapMisplaced}$, by Claim 5.4 and Theorem 5.19, we have the linear-time tight compaction claimed in Theorem 5.1.

5.3 Oblivious Distribution

In oblivious distribution, the input is an array $\mathbf{I}$ of n balls and a set $A \subseteq [n]$ such that each ball in $\mathbf{I}$ is labeled as 0 or 1 and the number of 0-balls is equal to $|A|$. The output is a permutation of $\mathbf{I}$ such that for each $i \in [n]$, the i-th location is a 0-ball if and only if $i \in A$. By marking $\mathsf{red}, \mathsf{blue}$, or $\perp$ correspondingly, $\mathsf{SwapMisplaced}$ achieves oblivious distribution as elaborated in Algorithm 5.20, where the set A is represented as an array of n indicators such that $i \in A$ iff $A[i] = 0$.

Algorithm 5.20: $\mathsf{Distribution}(\mathbf{I}, A)$

- **Input:** an array $\mathbf{I}$ of n balls and an array A of n bits, where each ball is labeled as 0 or 1, and the number of 0-balls in $\mathbf{I}$ equals to the number of 0s in A.
- **The algorithm:**
  1. For each $i \in [n]$, mark the ball $\mathbf{I}[i]$ as follows:
     (a) If $\mathbf{I}[i]$ is tagged with 1 and $A[i] = 0$, mark $\mathbf{I}[i]$ as red.

(b) If $\mathbf{I}[i]$ is tagged with 0 and $A[i] = 1$, mark $\mathbf{I}[i]$ as blue.

(c) Otherwise ($\mathbf{I}[i]$ is tagged with $A[i]$), mark $\mathbf{I}[i]$ as $\perp$.

2. Run SwapMisplaced($\mathbf{I}$) and let $\mathbf{O}$ be the result.

- **Output:** The array $\mathbf{O}$.

The correctness, security, and time complexity of Distribution follow directly from those of SwapMisplaced, and this gives the following theorem.

Theorem 5.21 (Oblivious distribution)**.** *There exists a deterministic oblivious distribution algorithm that takes $O(n)$ time on input arrays of size n.*

6 Interspersing Randomly Shuffled Arrays

In this section, we present the following variants of shuffling on an array of n elements. We suppose that for any $m \in [n]$, sampling an integer uniformly at random from the set $[m]$ takes unit time.[14]

6.1 Interspersing Two Arrays

We first describe a building block called Intersperse that allows us to randomly merge two randomly shuffled arrays. Informally, we would like to realize the following abstraction:

- **Input**: An array $\mathbf{I} := \mathbf{I}_0 \| \mathbf{I}_1$ of size n and two numbers n_0 and n_1 such that $|\mathbf{I}_0| = n_0$ and $|\mathbf{I}_1| = n_1$ and $n = n_0 + n_1$. We assume that each element in the input array fits in $O(1)$ memory words.
- **Output**: An array $\mathbf{B}$ of size n that contains all elements of $\mathbf{I}_0$ and $\mathbf{I}_1$. Each position in $\mathbf{B}$ will hold an element from either $\mathbf{I}_0$ or $\mathbf{I}_1$, chosen uniformly at random and the choices are concealed from the adversary.

Looking ahead, we will invoke the procedure Intersperse with arrays $\mathbf{I}_0$ and $\mathbf{I}_1$ that are *already* randomly and independently shuffled (each with a hidden permutation). So, when we apply Intersperse on such arrays the output array $\mathbf{B}$ is guaranteed to be a random permutation of the array $\mathbf{I} := \mathbf{I}_0 \| \mathbf{I}_1$ in the eyes of an adversary.

The intersperse algorithm. The idea is to first generate a random auxiliary array of 0's and 1's, denoted Aux, such that the number of 0's in the array is exactly n_0 and the number of 1's is exactly n_1. This can be done obliviously by sequentially sampling each bit depending on the number of 0's we sampled so far (see Algorithm 6.1). Aux is used to decide the following: if $\mathsf{Aux}[i] = 0$, then the i-th position in the output will pick up an element from $\mathbf{I}_0$, and otherwise, from $\mathbf{I}_1$.

Next, to obliviously route elements from $\mathbf{I}_0$ (and $\mathbf{I}_1$, respectively) to the i-th position such that $\mathsf{Aux}[i] = 0$ (and $\mathsf{Aux}[i] = 0$, respectively), it is performed using the deterministic oblivious distribution given in Section 5.3 — mark every element in $\mathbf{I}_0$ as 0-balls, mark every element in $\mathbf{I}_1$ as

[14] In the standard RAM model, we assume only a memory word (i.e., fair random bits) can be sampled uniformly at random in unit time. We note that to represent the probability of the shuffling exactly, infinite random bits are necessary, which implies that no shuffling can finish in worst-case finite time. One may approximate the stronger sampling using standard random words and repeating, and then bound the total number of repetition to get a high-probability time (where the total number of repetition is a sum of geometric random variables, which has a very sharp tail due to Chernoff bound). We adopt the stronger sampling for simplicity.

1-balls, and then run Distribution (Algorithm 5.20) on the marked array $\mathbf{I} = \mathbf{I}_0 \| \mathbf{I}_1$ and the auxiliary array Aux.

The formal description of the algorithm for interspersing two arrays is given in Algorithm 6.1. The functionality that it implements (assuming that the two input arrays are randomly shuffled) is given in Functionality 6.2 and the proof that the algorithm implements the functionality is given in Claim 6.3.

Algorithm 6.1: $\mathsf{Intersperse}_n(\mathbf{I}_0 \| \mathbf{I}_1, n_0, n_1)$ – Shuffling an Array via Interspersing Two Randomly Shuffled Subarrays

- **Input:** An array $\mathbf{I} := \mathbf{I}_0 \| \mathbf{I}_1$ that is a concatenation of two arrays $\mathbf{I}_0$ and $\mathbf{I}_1$ of sizes n_0 and n_1, respectively.
- **Public parameters:** $n := n_0 + n_1$.
- **Input assumption:** Each one of the arrays $\mathbf{I}_0, \mathbf{I}_1$ is independently randomly shuffled.
- **The algorithm:**
  1. Sample an auxiliary array Aux uniformly at random among all arrays of of size n with n_0 0's and n_1 1's:
     (a) Initialize $m_0 := n_0$ and $m_1 := n_1$.
     (b) For every position $1, 2, \ldots, n$, flip a random coin that results in heads with probability $\frac{m_1}{m_0+m_1}$. If heads, write down 1 and decrement m_1. Else, write down 0 and decrement m_0.
  2. For every $i \in [n]$, mark $\mathbf{I}[i]$ as 0 if $i \leq n_0$, otherwise mark $\mathbf{I}[i]$ as 1.
  3. Run $\mathsf{Distribution}(\mathbf{I}, \mathsf{Aux})$, let $\mathbf{B}$ be the resulting array.
- **Output:** The array $\mathbf{B}$.

Functionality 6.2: $\mathcal{F}^n_{\mathsf{Shuffle}}(\mathbf{I})$ – Randomly Shuffling an Array

- **Input:** An array $\mathbf{I}$ of size n.
- **Public parameters:** n.
- **The functionality:**
  1. Choose a permutation $\pi \colon [n] \to [n]$ uniformly at random.
  2. Initialize an array $\mathbf{B}$ of size n. Assign $\mathbf{B}[i] = \mathbf{I}[\pi(i)]$ for every $i = 1, \ldots, n$.
- **Output:** The array $\mathbf{B}$.

Claim 6.3. *Let $\mathbf{I}_0$ and $\mathbf{I}_1$ be two arrays of size n_0 and n_1, respectively, that satisfies the input assumption as in the description of Algorithm 6.1. The Algorithm $\mathsf{Intersperse}_n(\mathbf{I}_0 \| \mathbf{I}_1, n_0, n_1)$ obliviously implements functionality $\mathcal{F}^n_{\mathsf{Shuffle}}(\mathbf{I}_0 \| \mathbf{I}_1)$. The implementation has $O(n)$ time.*

The proof of this claim is deferred to Appendix C.2.

6.2 Interspersing Multiple Arrays

We generalize the Intersperse algorithm to work with $k \in \mathbb{N}$ arrays as input. The algorithm is called $\mathsf{Intersperse}^{(k)}$ and it implements the following abstraction:

- **Input**: An array $\mathbf{I}_1 \| \dots \| \mathbf{I}_k$ consisting of k different arrays of lengths $n_1, \dots, n_k$, respectively. The parameters $n_1, \dots, n_k$ are public.
- **Output**: An array $\mathbf{B}$ of size $\sum_{i=1}^{k} n_i$ that contains all elements of $\mathbf{I}_1, \dots, \mathbf{I}_k$. Each position in $\mathbf{B}$ will hold an element from one of the arrays, chosen uniformly at random and the choices are concealed from the adversary.

As in the case of $k = 2$, we will invoke the procedure $\mathsf{Intersperse}^{(k)}$ with arrays $\mathbf{I}_1, \dots, \mathbf{I}_k$ that are *already* randomly and independently shuffled (with k hidden permutations). So, when we apply $\mathsf{Intersperse}^{(k)}$ on such arrays the output array $\mathbf{B}$ is guaranteed to be a random permutation of the array $\mathbf{I} := \mathbf{I}_1 \| \dots \| \mathbf{I}_k$ in the eyes of an adversary.

The algorithm. To intersperse k arrays $\mathbf{I}_1, \dots, \mathbf{I}_k$, we intersperse the first two arrays using $\mathsf{Intersperse}_{n_1+n_2}$, then intersperse the result with the third array, and so on. The precise description is given in Algorithm 6.4.

Algorithm 6.4: $\mathsf{Intersperse}^{(k)}_{n_1,\dots,n_k}(\mathbf{I}_1 \| \dots \| \mathbf{I}_k)$ – Shuffling an Array via Interspersing k Randomly Shuffled Subarrays

- **Input:** An array $\mathbf{I} := \mathbf{I}_1 \| \dots \| \mathbf{I}_k$ consisting of k arrays of sizes $n_1, \dots, n_k$, respectively.
- **Public parameters:** $n_1, \dots, n_k$.
- **Input assumption:** Each input array is independently randomly shuffled.
- **The algorithm:**
  1. Let $\mathbf{I}'_1 := \mathbf{I}_1$.
  2. For $i = 2, \dots k$, do:
     (a) Execute $\mathsf{Intersperse}_{\sum_{j=1}^{i} n_j}(\mathbf{I}'_{i-1} \| \mathbf{I}_i, \sum_{j=1}^{i-1} n_j, n_i)$. Denote the result by $\mathbf{I}'_i$.
  3. Let $\mathbf{B} := \mathbf{I}'_k$.
- **Output:** The array $\mathbf{B}$.

We prove that this algorithm obliviously implements a uniformly random shuffle.

Claim 6.5. *Let $k \in \mathbb{N}$ and let $\mathbf{I}_1, \dots, \mathbf{I}_k$ be k arrays of $n_1, \dots, n_k$ elements, respectively, that satisfy the input assumption as in the description of Algorithm 6.4. The Algorithm $\mathsf{Intersperse}^{(k)}_{n_1,\dots,n_k}(\mathbf{I}_1 \| \dots \| \mathbf{I}_k)$ obliviously implements the functionality $\mathcal{F}^{n}_{\mathsf{Shuffle}}(\mathbf{I})$. The implementation requires $O\left(n + \sum_{i=1}^{k-1}(k-i) \cdot n_i\right)$ time.*

The proof of this claim is deferred to Appendix C.2.

6.3 Interspersing Reals and Dummies

We describe a related algorithm, called $\mathsf{IntersperseRD}$, which will also serve as a useful building block. Here, the abstraction we implement is the following:

- **Input**: An array $\mathbf{I}$ of n elements, where each element is tagged as either *real* or *dummy*. The real elements are distinct. We assume that if we extract the subset of all real elements in the array, then these elements appear in random order. However, there is no guarantee of the relative positions of the real elements with respect to the dummy ones.

- **Output**: An array $\mathbf{B}$ of size $|\mathbf{I}|$ containing all real elements in $\mathbf{I}$ and the same number of dummy elements, where all elements in the array are randomly permuted.

In other words, the real elements are randomly permuted, but there is no guarantee regarding their order in the array with respect to the dummy elements. In particular, the dummy elements can appear in arbitrary (known to the adversary) positions in the input, e.g., appear all in the front, all at the end, or appearing in all the odd positions. The output will be an array where all the real and dummy elements are randomly permuted, and the random permutation is hidden from the adversary.

The implementation of $\mathsf{IntersperseRD}$ is done by first running the deterministic tight compaction procedure on the input array such that all the real balls appear before the dummy ones. Next, we count the number of real elements in this array run the $\mathsf{Intersperse}$ procedure from Algorithm 6.1 on this array with the calculated sizes. The formal implementation appears as Algorithm 6.6.

Algorithm 6.6: $\mathsf{IntersperseRD}_n(\mathbf{I})$ – Shuffling an Array via Interspersing Real and Dummy

- **Input:** An array $\mathbf{I}$ of n elements, where each element is tagged as either *real* or *dummy*. The real elements are distinct. We assume that each element fits in $O(1)$ memory words.
- **Public parameters:** n.
- **Input assumption:** The input $\mathbf{I}$ restricted to the real elements is randomly shuffled.
- **The algorithm:**
  1. Run the deterministic oblivious tight compaction algorithm on $\mathbf{I}$ (see Section 4), such that all the real balls appear before the dummy ones. Let $\mathbf{I}'$ denote the output array of this step.
  2. Count the number of reals in $\mathbf{I}'$ by a linear scan. Let n_R denote the result.
  3. Invoke $\mathsf{Intersperse}_n(\mathbf{I}', n_R, n - n_R)$ and let $\mathbf{B}$ be the output.
- **Output:** The array $\mathbf{B}$.

We prove that this algorithm obliviously implements a uniformly random shuffle.

Claim 6.7. *Let $\mathbf{I}$ be an array of n elements that satisfies the input assumption as in the description of Algorithm 6.6. The Algorithm $\mathsf{IntersperseRD}_n(\mathbf{I})$ obliviously implements the functionality $\mathcal{F}^n_{\mathsf{Shuffle}}(\mathbf{I})$. The implementation has $O(n)$ time.*

The proof of this claim is deferred to Appendix C.2.

6.4 Perfect Oblivious Random Permutation (Proof of Theorem 4.6)

Recall that an oblivious random permutation shuffles an input array of n elements using a secret permutation $\pi : [n] \to [n]$ uniformly at random (Section 4.2). The following (perfect) oblivious random permutation, $\mathsf{PerfectORP}$, is constructed with standard divide-and-conquer technique using $\mathsf{Intersperse}$ for merging.

Algorithm 6.8: $\mathsf{PerfectORP}(\mathbf{I})$ – Perfect Oblivious Random Permutation

- **Input:** An array $\mathbf{I}$ of n elements.
- **The algorithm:**

1. (Base case.) If $n = 1$, output $\mathbf{I}$ directly (skip all following steps).
2. Let A_1 be the front $\lceil n/2 \rceil$ elements of $\mathbf{I}$, A_2 be the back $\lfloor n/2 \rfloor$ elements.
3. Recursively run $\mathsf{PerfectORP}(A_1), \mathsf{PerfectORP}(A_2)$, let A'_1, A'_2 be the results respectively.
4. Run $\mathsf{Intersperse}_n(A'_1 \| A'_2, \lceil n/2 \rceil, \lfloor n/2 \rfloor)$, let $\mathbf{O}$ be the result.

- **Output:** The array $\mathbf{O}$.

We argue that PerfectORP runs in $O(n \cdot \log n)$ time and permutes $\mathbf{I}$ uniformly at random. The time bound follows since Intersperse runs in $O(n)$ time (Claim 6.3) and the recursion consists of 2 sub-problems, each of half the size. The fact that the permutation is uniformly random follows by induction and that Intersperse perfectly-obliviously implements $\mathcal{F}^n_{\mathsf{Shuffle}}$ (Claim 6.3).

7 BigHT: Oblivious Hashing for Non-Recurrent Lookups

The hash table construction we describe in this section suffers from $\mathsf{poly} \log \log \lambda$ extra multiplicative factor in Build and Lookup (which lead to similar overhead in the implied ORAM construction). Nevertheless, this hash table serves as a first step and we will get rid of the extra factor in Section 8. Hence, the parameter of expected bin load $\mu = \log^9 \lambda$ is seemingly loose in this section but is necessary later in Section 8 (to apply Cuckoo hash). Additionally, note that this hash table captures and simplifies many of the ideas in the oblivious hash table of Patel et al. [52] and can be used to get an ORAM with similar overhead to theirs.

Construction 7.1: Hash Table for Shuffled Inputs

Procedure BigHT.Build(I):

- **Input:** An array $\mathbf{I} = (a_1, \ldots, a_n)$ containing n elements, where each a_i is either dummy or a (key, value) pair denoted (k_i, v_i), where both the key k and the value v are D-bit strings where $D := O(1) \cdot w$.
- **Input assumption:** The elements in the array are uniformly shuffled.
- **The algorithm:**
  1. Let $\mu := \log^9 \lambda$, $\epsilon := \frac{1}{\log^2 \lambda}$, $\delta := e^{-\log \lambda \cdot \log \log \lambda}$, and $B := \lceil n/\mu \rceil$.
  2. *Sample PRF key.* Sample a random PRF secret key sk.
  3. *Directly hash into major bins.* Throw the real $a_i = (k_i, v_i)$ into B bins using $\mathsf{PRF}_{\mathsf{sk}}(k_i)$. If $a_i = \mathsf{dummy}$, throw it to a uniformly random bin. Let $\mathsf{Bin}_1, \ldots, \mathsf{Bin}_B$ be the resulted bins.
  4. *Sample independent smaller loads.* Execute Algorithm 4.19 to obtain $(L_1, \ldots, L_B) \leftarrow \mathsf{SampleBinLoad}_{B,\delta}(n')$, where $n' = n \cdot (1 - \epsilon)$. If there exists $i \in [B]$ such that $||\mathsf{Bin}_i| - \mu| > 0.5 \cdot \epsilon\mu$ or $\left|L_i - \frac{n'}{B}\right| > 0.5 \cdot \epsilon\mu$, then abort.
  5. *Create major bins.* Allocate new arrays $(\mathsf{Bin}'_1, \ldots, \mathsf{Bin}'_B)$, each of size μ. For every i, iterate in parallel on both Bin_i and Bin'_i, and copy the first L_i elements in Bin_i to Bin'_i. Fill the empty slots in Bin'_i with dummy. (L_i is not revealed during this process, by continuing to iterate over Bin_i after we cross the threshold L_i.)
  6. *Create overflow pile.* Obliviously merge all of the last $|\mathsf{Bin}_i| - L_i$ elements in each bin $\mathsf{Bin}_1, \ldots, \mathsf{Bin}_B$ into an overflow pile:

- For each $i \in [B]$, replace the first L_i positions with dummy.
- Concatenate all of the resulting bins and perform oblivious tight compaction on the resulting array such that the real balls appear in the front. Truncate the outcome to be of length ϵn.

7. Prepare an oblivious hash table for elements in the overflow pile by calling the Build algorithm of the $(1 - O(\delta) - \delta^{\mathcal{A}}_{\mathsf{PRF}})$-oblivious Cuckoo hashing scheme (Theorem 4.14) parameterized by δ (recall that $\delta = e^{-\Omega(\log \lambda \cdot \log\log \lambda)}$) and the stash size $\log(1/\delta)/\log n$. Let $\mathsf{OF} = (\mathsf{OF}_{\mathsf{T}}, \mathsf{OF}_{\mathsf{S}})$ denote the outcome data structure. Henceforth, we use $\mathsf{OF.Lookup}$ to denote a lookup operation to this oblivious Cuckoo hashing scheme.
8. *Prepare data structure for efficient lookup.* For $i = 1, \ldots, B$, call $\mathsf{naïveHT.Build}(\mathsf{Bin}'_i)$ on each major bin to construct an oblivious hash table, and let OBin_i denote the outcome for the i-th bin.

- **Output:** The algorithm stores in the memory a state that consists of $(\mathsf{OBin}_1, \ldots, \mathsf{OBin}_B, \mathsf{OF}, \mathsf{sk})$.

Procedure BigHT.Lookup(k):

- **Input:** The secret state $(\mathsf{OBin}_1, \ldots, \mathsf{OBin}_B, \mathsf{OF}, \mathsf{sk})$, and a key k to look for (that may be $\perp$, i.e., dummy).
- **The algorithm:**
  1. Call $v \leftarrow \mathsf{OF.Lookup}(k)$.
  2. If $k = \perp$, choose a random bin $i \xleftarrow{\$} [B]$ and call $\mathsf{OBin}_i.\mathsf{Lookup}(\perp)$.
  3. If $k \neq \perp$ and $v \neq \perp$ (i.e., v was found in OF), choose a random bin $i \xleftarrow{\$} [B]$ and call $\mathsf{OBin}_i.\mathsf{Lookup}(\perp)$.
  4. If $k \neq \perp$ and $v = \perp$ (i.e., v was not found in OF), let $i := \mathsf{PRF}_{\mathsf{sk}}(k)$ and call $v \leftarrow \mathsf{OBin}_i.\mathsf{Lookup}(k)$.
- **Output:** The value v.

Procedure BigHT.Extract():

- **Input:** The secret state $(\mathsf{OBin}_1, \ldots, \mathsf{OBin}_B, \mathsf{OF}, \mathsf{sk})$.
- **The algorithm:**
  1. Let $T = \mathsf{OBin}_1.\mathsf{Extract}() \| \mathsf{OBin}_2.\mathsf{Extract}() \| \ldots \| \mathsf{OBin}_B.\mathsf{Extract}() \| \mathsf{OF.Extract}()$.
  2. Perform oblivious tight compaction on T, moving all the real balls to the front. Truncate the resulting array at length n. Let $\mathbf{X}$ be the outcome of this step.
  3. Call $\mathbf{X}' \leftarrow \mathsf{IntersperseRD}_n(\mathbf{X})$ (Algorithm 6.6).
- **Output**: $\mathbf{X}'$.

We prove that our construction obliviously implements Functionality 4.7 for every sequence of instructions with non-recurrent lookups between two Build operations and as long as the input array to Build is randomly and secretly shuffled.

Theorem 7.2. *Assume a $\delta^{\mathcal{A}}_{\mathsf{PRF}}$-secure PRF. Then, Construction 7.1 $(1 - n^2 \cdot e^{-\Omega(\log \lambda \cdot \log\log \lambda)} - \delta^{\mathcal{A}}_{\mathsf{PRF}})$-obliviously implements Functionality 4.7 for all $n \geq \log^{11} \lambda$, assuming that the input array (of size n) for Build is randomly shuffled. Moreover,*

- Build *and* $\mathsf{Extract}$ *each take* $O\left(n \cdot \mathsf{poly}\log\log\lambda + n \cdot \frac{\log n}{\log^2 \lambda}\right)$ *time; and*
- Lookup *takes* $O(\mathsf{poly}\log\log\lambda)$ *time in addition to linearly scanning a stash of size* $O(\log\lambda)$.

In particular, if $\log^{11}\lambda \le n \le \mathsf{poly}(\lambda)$*, then hash table is* $(1 - e^{-\Omega(\log\lambda \cdot \log\log\lambda)} - \delta^{\mathcal{A}}_{\mathsf{PRF}})$*-obliviously and consumes* $O(n \cdot \mathsf{poly}\log\log\lambda)$ *time for the* Build *and* $\mathsf{Extract}$ *phases; and* Lookup *consumes* $O(\mathsf{poly}\log\log\lambda)$ *time in addition to linearly scanning a stash of size* $O(\log\lambda)$.

Proof. The proof of security is given in Appendix C.3. We give the efficiency analysis here. In Construction 7.1, there are $n/\log^9\lambda$ major bins and each is of size $O(\log^9\lambda)$. The subroutine $\mathsf{SampleBinLoad}_{B,\delta}(n')$ runs in time $O(B \cdot \log^5(1/\delta)) \le O\left(\lceil n/\log^9\lambda\rceil \cdot \log^6\lambda\right) = O\left(n/\log^3\lambda\right)$ by Theorem 4.20 (recall that $\delta = e^{-\log\lambda\cdot\log\log\lambda}$) and since $n \ge \log^{11}\lambda$. We employed the hash table $\mathsf{naïveHT}$ (Theorem 4.9) for each major bin, and thus their initialization takes time

$$\frac{n}{\mu} \cdot O\left(\mu \cdot \mathsf{poly}\log\mu\right) \le O(n \cdot \mathsf{poly}\log\log\lambda).$$

The overflow pile consists of $\epsilon n \ge \log^9\lambda \ge \log^8(1/\delta)$ elements as $n \ge \log^{11}\lambda$, and it is implemented via an oblivious Cuckoo hashing scheme (Theorem 4.14) so its initialization takes time $O(\epsilon n \cdot \log(\epsilon n)) \le O\left(n \cdot \frac{\log n}{\log^2\lambda}\right)$, where the stash size is $O\left(\frac{\log(1/\delta)}{\log \epsilon n}\right) \le O(\log\lambda)$. Each Lookup incurs $O(\mathsf{poly}\log\log\lambda)$ time from the major bins and $O(\log\lambda)$ time from the linear scan of OF_{S}, the stash of the overflow pile (searching in OF_{T} incurs $O(1)$ time). The overhead of $\mathsf{Extract}$ depends on the overhead of $\mathsf{Extract}$ for each major bin and $\mathsf{Extract}$ from the overflow pile. The former is again bounded by $O(n \cdot \mathsf{poly}\log\log\lambda)$ and the latter is bounded by $O(\epsilon n \cdot \log(\epsilon n)) \le O\left(n \cdot \frac{\log n}{\log^2\lambda}\right)$. □

Finally, observe that it is not difficult to adjust the constants in our construction and analysis to show the following more general corollary:

Corollary 7.3. *Assume a* $\delta^{\mathcal{A}}_{\mathsf{PRF}}$*-secure PRF. Then, for any constant* $c \ge 2$*, there exists an algorithm that* $(1 - n^2 \cdot e^{-\Omega(\log\lambda\cdot\log\log\lambda)} - \delta^{\mathcal{A}}_{\mathsf{PRF}})$*-obliviously implements Functionality 4.7 for all* $n \ge \log^{9+c}\lambda$*, assuming that the input array (of size* n*) for* Build *is randomly shuffled. Moreover,*

- Build *and* $\mathsf{Extract}$ *each take* $O\left(n \cdot \mathsf{poly}\log\log\lambda + n \cdot \frac{\log n}{\log^c\lambda}\right)$ *time; and*
- Lookup *takes* $O(\mathsf{poly}\log\log\lambda)$ *time in addition to linearly scanning a stash of size* $O(\log\lambda)$.

Proof. We can let $\epsilon = \frac{1}{\log^c\lambda}$ in Construction 7.1. The analysis follows in a similar fashion. □

Remark 7.4. *As we mentioned, Construction 7.1 is only the first step towards the final oblivious hash table that we use in the final ORAM construction. We make significant optimizations in Section 8. We show how to improve upon the* Build *and* $\mathsf{Extract}$ *procedures from* $O(n \cdot \mathsf{poly}\log\log\lambda)$ *to* $O(n)$ *by replacing the* $\mathsf{naïveHT}$ *hash table with an optimized version (called* $\mathsf{SmallHT}$*) that is more efficient for small lists. Additionally, while it may now seem that the* $O(\log\lambda)$*-stash overhead of* Lookup *is problematic, we will "merge" the stashes for different hash tables in our final ORAM construction and store them again in an oblivious hash table.*

8 SmallHT: Oblivious Hashing for Small Bins

In Section 7, we constructed an oblivious hashing scheme for randomly shuffled inputs where Build and Extract consumes $n \cdot \mathsf{poly}\log\log\lambda$ time and Lookup consumes $\mathsf{poly}\log\log\lambda$. The extra $\mathsf{poly}\log\log\lambda$ factors arise from the oblivious hashing scheme (denoted naïveHT) which we use for each major bin of size $\approx \log^9\lambda$. To get rid of the extra $\mathsf{poly}\log\log\lambda$ factors, in this section, we will construct a new oblivious hashing scheme for $\mathsf{poly}\log\lambda$-sized arrays which are randomly shuffled. In our new construction, Build and Extract takes linear time and Lookup takes constant time (ignoring the stash which we will treat separately later).

As mentioned in Section 2.1, the key idea is to rely on *packed* operations such that the metadata phase of Build (i.e., the cuckoo assignment problem) takes only linear time — this is possible because the problem size $n = \mathsf{poly}\log\lambda$ is small. The more tricky step is how to route the actual balls into their destined location in the hash-table. We cannot rely on standard oblivious sorting to perform this routing since this would consume a logarithmic extra overhead. Instead, we devise a method to directly place the balls into the destined location in the hash-table in the clear — this is safe as long as the input array has been padded with dummies to the output length, and randomly shuffled; in this way only a random permutation is revealed. A technicality arises in realizing this idea: after figuring out the assigned destinations for real elements, we need to expand this assignment to include dummy elements too, and the dummy elements must be assigned at random to the locations unoccupied by the reals. At a high level, this is accomplished through a combination of packed oblivious random permutation and packed oblivious sorting over metadata.

We first describe two helpful procedures (mentioned in Section 2.1.2) in Sections 8.1 and 8.2. Then, in Section 8.3, we give the full description of the Build, Lookup, and Extract procedures (Construction 8.5). Throughout this section, we assume for simplicity that $n = \log^9\lambda$ (while in reality $n \in \log^9\lambda \pm \log^7\lambda$).

8.1 Step 1 – Add Dummies and Shuffle

We are given a randomly shuffled array $\mathbf{I}$ of length n that contains real and dummy elements. In Algorithm 8.1, we pad the input array with dummies to match the size of the hash-table to be built. Each dummy will receive a unique index label, and we rely on packed oblivious random permutation to permute the labeled dummies. Finally, we rely on Intersperse on the real balls to make sure that all elements, including reals and dummies, are randomly shuffled.

More formally, the output of Algorithm 8.1 is an array of size $n_{\mathsf{cuckoo}} = \mathsf{c}_{\mathsf{cuckoo}} \cdot n + \log\lambda$, where $\mathsf{c}_{\mathsf{cuckoo}}$ is the constant required for Cuckoo hashing, which contains all the real elements from $\mathbf{I}$ and the rest are dummies. Furthermore, each dummy receives a distinct random index from $\{1, \ldots, n_{\mathsf{cuckoo}} - n_R\}$, where n_R is the number of real elements in $\mathbf{I}$. Assuming that the real elements in $\mathbf{I}$ are a-priori uniformly shuffled, then the output array is randomly shuffled.

Algorithm 8.1: Shuffle the Real and Dummy Elements

- **Input:** An input array $\mathbf{I}$ of length n consisting of real and dummy elements.
- **Input Assumption:** The real elements among $\mathbf{I}$ are randomly shuffled.
- **The algorithm:**
  1. Count the number of real elements in $\mathbf{I}$. Let n_R be the output.
  2. Write down a metadata array $\mathbf{MD}$ of length n_{cuckoo}, where the first n_R elements contain only a symbol `real`, and the remaining $n_{\mathsf{cuckoo}} - n_R$ elements are of the form $(\perp, 1), (\perp, 2), \ldots, (\perp, n_{\mathsf{cuckoo}} - n_R)$, i.e., each element is a $\perp$ symbol tagged with a dummy index.

3. Run packed oblivious random permutation (Theorem 4.5) on $\mathbf{MD}$, packing $O\left(\frac{w}{\log n}\right)$ elements into a single memory word. Run oblivious tight compaction (Theorem 5.1) on the resulting array, moving all the dummy elements to the end.
4. Run tight compaction (Theorem 5.1) on the input $\mathbf{I}$ to move all the real elements to the front.
5. Obliviously write down an array $\mathbf{I}'$ of length n_{cuckoo}, where the first n_R elements are the first n_R elements of $\mathbf{I}$ and the last $n_{\mathsf{cuckoo}} - n_R$ elements are the last $n_{\mathsf{cuckoo}} - n_R$ elements of $\mathbf{MD}$, decompressed to the original length as every entry in the input $\mathbf{I}$.
6. Run $\mathsf{Intersperse}$ on $\mathbf{I}'$ (Algorithm 6.6) letting $n_1 := n_R$ and $n_2 := n_{\mathsf{cuckoo}} - n_R$. Let $\mathbf{X}$ denote the outcome array.

- **Output:** The array $\mathbf{X}$.

Claim 8.2. *Algorithm 8.1 fails with probability at most $e^{-\Omega(\sqrt{n})}$ and completes in $O(n + \frac{n}{w} \cdot \log^3 n)$ time. Specifically, for $n = \log^9 \lambda$ and $w \geq \log^3 \log \lambda$, the algorithm completes in $O(n)$ time and fails with probability $e^{-\Omega(\log^{9/2} \lambda)}$.*

Proof. All steps except the oblivious random permutation in Step 3 incur $O(n)$ time and are perfectly correct by construction. Each element of $\mathbf{MD}$ can be expressed with $O(\log n)$ bits, so the packed oblivious random permutation (Theorem 4.5) incurs $O\left((n \cdot \log^3 n)/w\right)$ time and has failure probability at most $e^{-\Omega(\sqrt{n})}$. □

8.2 Step 2 – Evaluate Assignment with Metadata Only

We obliviously emulate the Cuckoo hashing procedure, but doing it directly on the input array is too expensive (as it incurs oblivious sorting inside) so we do it directly on metadata (which is short since there are few elements), and use the packed version of oblivious sort (Theorem 4.2). At the end of this step, every element in the input array should learn which bin (either in the main table or the stash) it is destined for. Recall that the Cuckoo hashing consists of a main table of $\mathsf{c}_{\mathsf{cuckoo}} \cdot n$ bins and a stash of $\log \lambda$ bins.

Our input for this step is an array $\mathbf{MD}_{\mathbf{X}}$ of length $n_{\mathsf{cuckoo}} := \mathsf{c}_{\mathsf{cuckoo}} \cdot n + \log \lambda$ which consists of pairs of bin choices $(\mathsf{choice}_1, \mathsf{choice}_2)$, where each choice is an element from $[\mathsf{c}_{\mathsf{cuckoo}} \cdot n] \cup \{\perp\}$. The real elements have choices in $[\mathsf{c}_{\mathsf{cuckoo}} \cdot n]$ while the dummies have $\perp$. This array corresponds to the bin choices of the original elements in $\mathbf{X}$ (using a PRF) which is the original array $\mathbf{I}$ after adding enough dummies and randomly shuffling that array.

To compute the bin assignments we start with obliviously assigning the bin choices of the real elements in $\mathbf{MD}_{\mathbf{X}}$. Next, we obliviously assign the remaining dummy elements to the remaining available locations. We do so by a sequence of oblivious sort algorithms. See Algorithm 8.3.

Algorithm 8.3: Evaluate Cuckoo Hash Assignment on Metadata

- **Input:** An array $\mathbf{MD}_{\mathbf{X}}$ of length $n_{\mathsf{cuckoo}} = \mathsf{c}_{\mathsf{cuckoo}} \cdot n + \log \lambda$, where each element is either dummy or a pair $(\mathsf{choice}_{i,1}, \mathsf{choice}_{i,2})$, where $\mathsf{choice}_{i,b} \in [\mathsf{c}_{\mathsf{cuckoo}} \cdot n]$ for every $b \in \{1, 2\}$, and the number of real pairs is at most n.
- **Remark:** All oblivious sorting in the algorithm below will be instantiated using packed oblivious sorting (including those called by $\overline{\mathsf{cuckooAssign}}$ and oblivious bin placement).
- **The algorithm:**

1. Run the indiscriminate oblivious Cuckoo assignment algorithm $\overline{\mathsf{cuckooAssign}}$ (see Section 4.5) with parameter $\delta = e^{-\log\lambda\log\log\lambda}$, and let $\mathbf{Assign_X}$ be the result. For every i for which $\mathbf{MD_X}[i] = (\mathsf{choice}_{i,1}, \mathsf{choice}_{i,2})$, we have that $\mathbf{Assign_X}[i] \in \{\mathsf{choice}_{i,1}, \mathsf{choice}_{i,2}\} \cup S_{\mathsf{stash}}$, i.e., either one of the two choices or the stash $S_{\mathsf{stash}} = [n_{\mathsf{cuckoo}}] \setminus [\mathsf{c}_{\mathsf{cuckoo}} \cdot n]$. For every i for which $\mathbf{MD_X}[i]$ is dummy we have that $\mathbf{Assign_X}[i] = \perp$.
2. Run oblivious bin placement (Section 4.3) on $\mathbf{Assign_X}$, and let $\mathbf{Occupied}$ be the output array (of length n_{cuckoo}). For every index j we have $\mathbf{Occupied}[j] = i$ if $\mathbf{Assign_X}[i] = j$ for some i. Otherwise, $\mathbf{Occupied}[j] = \perp$.
3. Label the i-th element in $\mathbf{Assign_X}$ with a tag $t = i$ for all i. Run oblivious sorting on $\mathbf{Assign_X}$ and let $\widetilde{\mathbf{Assign}}$ be the resulting array, such that all real elements appear in the front, and all dummies appear at the end, and ordered by their respective dummy-index (i.e. given in Algorithm 8.1, Step 2).
4. Label the i-th element in $\mathbf{Occupied}$ with a tag $t = i$ for all i. Run oblivious sorting on $\mathbf{Occupied}$ and let $\widetilde{\mathbf{Occupied}}$ be the resulting array, such that all occupied bins appear in the front and all empty bins appear at the end (where each empty bin contains an index (i.e., a tag t) of an empty bin in $\mathbf{Occupied}$).
5. Scan both arrays $\widetilde{\mathbf{Assign}}$ and $\widetilde{\mathbf{Occupied}}$ in parallel, updating the destined bin of each dummy element in $\widetilde{\mathbf{Assign}}$ with the respective tag in $\widetilde{\mathbf{Occupied}}$ (and each real element pretends to be updated).
6. Run oblivious sorting on the array $\widetilde{\mathbf{Assign}}$ (back to the original ordering in the array $\mathbf{Assign_X}$) according to the tag labeled in Step 3. Update the assignments of all dummy elements in $\mathbf{Assign_X}$ according to the output array of this step.

- **Output:** The array $\mathbf{Assign_X}$.

Claim 8.4. *For $n \geq \log^9\lambda$, Algorithm 8.3 fails with probability at most $e^{-\Omega(\log\lambda\cdot\log\log\lambda)}$ and completes in $O\left(n \cdot (1 + \frac{\log^3 n}{w})\right)$ time. Specifically, for $n = \log^9\lambda$ and $w \geq \log^3\log\lambda$, Algorithm 8.3 completes in $O(n)$ time.*

Proof. The input arrays is of size $n_{\mathsf{cuckoo}} = \mathsf{c}_{\mathsf{cuckoo}} \cdot n + \log\lambda$ and the arrays $\mathbf{MD_X}$, $\mathbf{Assign_X}$, $\mathbf{Occupied}$, $\widetilde{\mathbf{Occupied}}$, $\widetilde{\mathbf{Assign}}$ are all of length at most n_{cuckoo} and consist of elements that need $O(\log n_{\mathsf{cuckoo}})$ bits to describe. Thus, the cost of packed oblivious sort (Theorem 4.2) is $O((n_{\mathsf{cuckoo}}/w) \cdot \log^3 n_{\mathsf{cuckoo}}) \leq O((n \cdot \log^3 n)/w)$. The linear scans take time $O(n_{\mathsf{cuckoo}}) = O(n)$. The cost of the $\overline{\mathsf{cuckooAssign}}$ (see Corollary 4.12) from Step 1 has failure probability $e^{-\Omega(\log\lambda\cdot\log\log\lambda)}$ and it takes time $O((n_{\mathsf{cuckoo}}/w) \cdot \log^3 n_{\mathsf{cuckoo}}) \leq O((n \cdot \log^3 n)/w)$. □

8.3 SmallHT Construction

The full description of the construction is given next. It invokes Algorithms 8.1 and 8.3.

Construction 8.5: SmallHT – **Hash table for Small Bins**

Procedure SmallHT.Build(I):

- **Input:** An input array $\mathbf{I}$ of length n consisting of real and dummy elements. Each real element is of the form (k, v) where both the key k and the value v are D-bit strings where $D := O(1) \cdot w$.

- **Input Assumption:** The real elements among $\mathbf{I}$ are randomly shuffled.
- **The algorithm:**
  1. Run Algorithm 8.1 (prepare real and dummy elements) on input $\mathbf{I}$, and receive back an array $\mathbf{X}$.
  2. Choose a PRF key sk where PRF maps $\{0,1\}^D \to [\mathsf{c_{cuckoo}} \cdot n]$.
  3. Create a new metadata array $\mathbf{MD_X}$ of length n. Iterate over the the array $\mathbf{X}$ and for each real element $\mathbf{X}[i] = (k_i, v_i)$ compute two values $(\mathsf{choice}_{i,1}, \mathsf{choice}_{i,2}) \leftarrow \mathsf{PRF}_{\mathsf{sk}}(k_i)$, and write $(\mathsf{choice}_{i,1}, \mathsf{choice}_{i,2})$ in the i-th location of $\mathbf{MD_X}$. If $\mathbf{X}[i]$ is dummy, write $(\perp, \perp)$ in the i-th location of $\mathbf{MD_X}$.
  4. Run Algorithm 8.3 on $\mathbf{MD_X}$ to compute the assignment for every element in $\mathbf{X}$. The output of this algorithm, denoted $\mathbf{Assign_X}$, is an array of length n, where in the i-th position we have the destination location of element $\mathbf{X}[i]$.
  5. Route the elements of $\mathbf{X}$, in the clear, according to $\mathbf{Assign_X}$, into an array $\mathbf{Y}$ of size $\mathsf{c_{cuckoo}} \cdot n$ and into a stash S.
- **Output:** The algorithm stores in the memory a secret state consists of the array $\mathbf{Y}$, the stash S and the secret key sk.

Procedure SmallHT.Lookup(k):

- **Input:** A key k that might be dummy $\perp$. It receives a secret state that consists of an array $\mathbf{Y}$, a stash S, and a key sk.
- **The algorithm:**
  1. If $k \neq \perp$:
     (a) Evaluate $(\mathsf{choice}_1, \mathsf{choice}_2) \leftarrow \mathsf{PRF}_{\mathsf{sk}}(k)$.
     (b) Visit $\mathbf{Y}_{\mathsf{choice}_1}, \mathbf{Y}_{\mathsf{choice}_2}$ and the stash S to look for the key k. If found, remove the element by overwriting $\perp$. Let v^* be the corresponding value (if not found, set $v^* := \perp$).
  2. Otherwise:
     (a) Choose random $(\mathsf{choice}_1, \mathsf{choice}_2)$ independently at random from $[\mathsf{c_{cuckoo}} \cdot n]$.
     (b) Visit $\mathbf{Y}_{\mathsf{choice}_1}, \mathbf{Y}_{\mathsf{choice}_2}$ and the stash S and look for the key k. Set $v^* := \perp$.
- **Output:** Return v^*.

Procedure SmallHT.Extract().

- **Input:** The algorithm has no input; It receives the secret state that consists of an array $\mathbf{Y}$, a stash S, and a key sk.
- **The algorithm:**
  1. Perform oblivious tight compaction (Theorem 5.1) on $\mathbf{Y} \| \mathsf{S}$, moving all the real elements to the front. Truncate the resulting array at length n. Let $\mathbf{X}$ be the outcome of this step.
  2. Call $\mathbf{X}' \leftarrow \mathsf{IntersperseRD}_n(\mathbf{X})$ (Algorithm 6.6).
- **Output:** The array $\mathbf{X}'$.

We prove that our construction obliviously implements Functionality 4.7 for every sequence of instructions with non-recurrent lookups between two Build operations, assuming that the input array for Build is randomly shuffled.

Theorem 8.6. *Assume a $\delta^{\mathcal{A}}_{\mathsf{PRF}}$-secure PRF. Suppose that $n = \log^9 \lambda$ and $w \geq \log^3 \log \lambda$. Then, Construction 8.5 $(1 - n \cdot e^{-\Omega(\log \lambda \cdot \log \log \lambda)} - \delta^{\mathcal{A}}_{\mathsf{PRF}})$-obliviously implements Functionality 4.7 assuming that the input for* Build *(of size n) is randomly shuffled. Moreover,* Build *and* Extract *incur $O(n)$ time,* Lookup *has constant time in addition to linearly scanning a stash of size $O(\log \lambda)$.*

Proof. The proof of security is given in Appendix C.4. We proceed with the efficiency analysis. The Build operation executes Algorithm 8.1 that consumes $O(n)$ time (by Claim 8.2), then performs additional $O(n)$ time, then executes Algorithm 8.3 that consumes $O(n)$ time (by Claim 8.4), and finally performs additional $O(n)$ time. Thus, the total time is $O(n)$. Lookup, by construction, incurs $O(1)$ time in addition to linearly scanning the stash S which is of size $O(\log \lambda)$. The time of Extract is $O(n)$ by construction. □

8.4 CombHT: Combining BigHT with SmallHT

We use SmallHT in place of naïveHT for each of the major bins in the BigHT construction from Section 7. Since the load in the major bin in the hash table BigHT construction is indeed $n = \log^9 \lambda$, this modification is valid. Note that we still assume that the number of elements in the input to CombHT, is at least $\log^{11} \lambda$ (as in Theorem 7.2).

However, we make one additional modification that will be useful for us later in the construction of the ORAM scheme (Section 9). Recall that each instance of SmallHT has a stash S of size $O(\log \lambda)$ and so Lookup will require, not only searching an element in the (super-constant size) stash OF_S of the overflow pile from BigHT, but also linearly scanning the super-constant size stash of the corresponding major bin. To this end, we merge the different stashes of the major bins and store the merged list in an oblivious Cuckoo hash (Section 4.5). (A similar idea has also been applied in several prior works [15, 32, 34, 39].) This results with a new hash table scheme we call CombHT.

Construction 8.7: CombHT: **combining** BigHT **with** SmallHT

Procedure CombHT.Build(I): Run Steps 1–7 of Procedure BigHT.Build in Construction 7.1, where in Step 7 let $\mathsf{OF} = (\mathsf{OF}_\mathsf{T}, \mathsf{OF}_\mathsf{S})$ denote the outcome structure of the overflow pile. Then, perform:

8. *Prepare data structure for efficient lookup.* For $i = 1, \ldots, B$, call $\mathsf{SmallHT.Build}(\mathsf{Bin}_i)$ on each major bin to construct an oblivious hash table, and let $\{(\mathsf{OBin}_i, \mathsf{S}_i)\}_{i \in [B]}$ denote the outcome bins and the stash.
9. Concatenate the stashes $\mathsf{S}_1, \ldots, \mathsf{S}_B$ (each of size $O(\log \lambda)$) from all small hash tables together. Pad the concatenated stash (of size $O(n/\log^7 \lambda)$) to the size $O(n/\log^2 \lambda)$. Call the Build algorithm of an oblivious Cuckoo hashing scheme on the combined set (Section 4.5), and let $\mathsf{CombS} = (\mathsf{CombS}_\mathsf{T}, \mathsf{CombS}_\mathsf{S})$ denote the output data structure, where $\mathsf{CombS}_\mathsf{T}$ is the main table and $\mathsf{CombS}_\mathsf{S}$ is the stash.

Output: Output $(\mathsf{OBin}_1, \ldots, \mathsf{OBin}_B, \mathsf{OF}, \mathsf{CombS}, \mathsf{sk})$.

Procedure Lookup(k_i): The procedure is the same as in Construction 7.1, except that whenever visiting some bin OBin_j for searching for a key k_i, instead of visiting its stash to look for k_i, we visit CombS.

Procedure Extract(). The procedure Extract is the same as in Construction 7.1, except that $T = \mathsf{OBin}_1.\mathsf{Extract}() \| \ldots \| \mathsf{OBin}_B.\mathsf{Extract}() \| \mathsf{OF.Extract}() \| \mathsf{CombS.Extract}()$.

Theorem 8.8. *Assume a $\delta^{\mathcal{A}}_{\mathsf{PRF}}$-secure PRF. Suppose that the input $\mathbf{I}$ of the* Build *algorithm has length $n \geq \log^{11} \lambda$. Then, Construction 8.7 is $(1-n^2 \cdot e^{-\Omega(\log \lambda \log\log \lambda)} - \delta^{\mathcal{A}}_{\mathsf{PRF}})$-obliviously implements Functionality 4.7 assuming that the input for* CombHT.Build *is randomly shuffled. Moreover,*

- Build *and* Extract *each take $O\left(n + n \cdot \frac{\log n}{\log^2 \lambda}\right)$ time; and*
- Lookup *takes $O(1)$ time in addition to linearly scanning a stash of size $O(\log \lambda)$.*

In particular, if $\log^{11} \lambda \leq n \leq \mathsf{poly}(\lambda)$, the hash table is $(1 - e^{-\Omega(\log \lambda \cdot \log\log \lambda)}) - \delta^{\mathcal{A}}_{\mathsf{PRF}})$-oblivious and consumes $O(n)$ time for the Build *and* Extract *phases; and* Lookup *consumes $O(1)$ time in addition to linearly scanning a stash of size $O(\log \lambda)$.*

Proof. The proof of security is given in Appendix C.5. We proceed with the efficiency analysis. Since each stash S_i is of size $O(\log \lambda)$ and there are $n/\log^9 \lambda$ major bins, the merged and padded stash CombS has size size $O(n/\log^2 \lambda)$. The size of the overflow pile OF is $O(n/\log^2 \lambda)$. Thus, we can store each of them using an oblivious Cuckoo hashing and this requires $O(n/\log^2 \lambda)$ space for the main tables (resulting with OF_T and $\mathsf{CombS}_\mathsf{T}$) plus an additional stash of size $O(\log \lambda)$ (resulting with OF_S and $\mathsf{CombS}_\mathsf{S}$).

Thus, by Theorem 8.6 and 7.2, CombHT.Build($\mathbf{I}$) and CombHT.Extract performs in $O\left(n + n \cdot \frac{\log n}{\log^2 \lambda}\right)$ time. Regarding CombHT.Lookup, it needs to perform a linear scan in two stashes (OF_S and $\mathsf{CombS}_\mathsf{S}$) of size $O(\log \lambda)$ plus constant time to search the main Cuckoo hash tables (OF_T and $\mathsf{CombS}_\mathsf{T}$). □

If we use the earlier Corollory 7.3 to instantiate the BigHT, we can generalize the above theorem to the following corollary:

Corollary 8.9. *Assume a $\delta^{\mathcal{A}}_{\mathsf{PRF}}$-secure PRF and $c \geq 2$. Then, there exists an algorithm that $(1 - n^2 \cdot e^{-\Omega(\log \lambda \log\log \lambda)} - \delta^{\mathcal{A}}_{\mathsf{PRF}})$-obliviously implements Functionality 4.7 assuming that the input array is of length at least $n \geq \log^{9+c} \lambda$ and moreover the input is randomly shuffled. Furthermore, the algorithm achieves the following performance:*

- Build *and* Extract *each take $O\left(n + n \cdot \frac{\log n}{\log^c \lambda}\right)$ time; and*
- Lookup *takes $O(1)$ time in addition to linearly scanning a stash of size $O(\log \lambda)$.*

Remark 8.10. *In our ORAM construction we will have $O(\log N)$ levels, where each (non-empty) level has a merged stash and also a stash from the overflow pile OF_S, both of size $O(\log \lambda)$. We will employ the "merged stash" trick once again, merging the stashes of every level in the ORAM into a single one, resulting with a total stash size $O(\log N \cdot \log \lambda)$. We will store this merged stash in an oblivious dictionary (see Section 4.6), and accessing this merged stash would cost $O\left(\log^4(\log N + \log \lambda)\right)$ total time.*

9 Oblivious RAM

In this section, we utilize CombHT in the hierarchical framework of Goldreich and Ostrovsky [30] to construct our ORAM scheme. We denote by λ the security parameter. For simplicity, we assume that N, the size of the logical memory, is a power of 2. Additionally, we assume that w, the word size is $\Theta(\log N)$.

ORAM Initialization. Our structure consists of one dictionary D (see Section 4.6), and $O(\log N)$ levels numbered $\ell+1, \ldots, L$ respectively, where $\ell = \lceil 11 \log\log \lambda \rceil$, and $L = \lceil \log N \rceil$ is the maximal level.

- The dictionary D is an oblivious dictionary storing $2^{\ell+1}$ elements.
- Each level $i \in \{\ell+1, \ldots, L\}$ consists of an instance, called T_i, of the oblivious hash table CombHT from Section 8.4 that has capacity 2^i.

Additionally, each level is associated with an additional bit full_i, where 1 stands for *full* and 0 stands for *available*. Available means that this level is currently empty and does not contain any blocks, and thus one can rebuild into this level. Full means that this level currently contains blocks, and therefore an attempt to rebuild into this level will effectively cause a cascading merge. In addition, there is a global counter ctr that is initialized to 0.

Construction 9.1: Oblivious RAM $\mathsf{Access}(\mathsf{op}, \mathsf{addr}, \mathsf{data})$**.**

- **Input:** $\mathsf{op} \in \{\mathsf{read}, \mathsf{write}\}$, $\mathsf{addr} \in [N]$ and $\mathsf{data} \in \{0,1\}^w$.
- **Secret state:** The dictionary D, levels $T_{\ell+1}, \ldots, T_L$, the bits $\mathsf{full}_{\ell+1}, \ldots, \mathsf{full}_L$ and counter ctr.
- **The algorithm:**
  1. Initialize $\mathsf{found} := \mathsf{false}$, $\mathsf{data}^* := \perp$.
  2. Perform $\mathsf{fetched} := D.\mathsf{Lookup}(\mathsf{addr})$. If $\mathsf{fetched} \neq \perp$, then set $\mathsf{found} := \mathsf{true}$.
  3. For each $i \in \{\ell+1, \ldots, L\}$ in increasing order, do:
     (a) If $\mathsf{found} = \mathsf{false}$:
        i. Run $\mathsf{fetched} := T_i.\mathsf{Lookup}(\mathsf{addr})$ with the following modifications:
           – Do not visit the stash of OF, namely OF_S, in Construction 8.7.
           – Do not visit the stash of CombS.
           (below, these stashes $(\mathsf{OF}_\mathsf{S}, \mathsf{CombS}_\mathsf{S})$ are merged into previous levels.)
        ii. If $\mathsf{fetched} \neq \perp$, let $\mathsf{found} := \mathsf{true}$ and $\mathsf{data}^* := \mathsf{fetched}$.
     (b) Else, $T_i.\mathsf{Lookup}(\perp)$.
  4. If $\mathsf{found} = \mathsf{false}$, i.e., this is the first time addr is being accessed, set $\mathsf{data}^* = 0$.
  5. Let $(k, v) := \{(\mathsf{addr}, \mathsf{data}^*)\}$ if this is a read operation; else let $(k, v) := \{(\mathsf{addr}, \mathsf{data})\}$. Insert $(k, (\ell, \perp, v))$ into oblivious dictionary D using $D.\mathsf{Insert}(k, (\ell, \perp, v))$.
  6. Increment ctr by 1. If $\mathsf{ctr} \equiv 0 \mod 2^\ell$, perform the following.
     (a) Let j be the smallest level index such that $\mathsf{full}_j = 0$ (i.e., available). If all levels are marked full, then $j := L$. In other words, j is the target level to be rebuilt.
     (b) Let $\mathbf{U} := D.\mathsf{Extract}() \| T_{\ell+1}.\mathsf{Extract}() \| \ldots \| T_{j-1}.\mathsf{Extract}()$ and set $j^* := j-1$. If all levels are marked full, then additionally let $\mathbf{U} := \mathbf{U} \| T_L.\mathsf{Extract}()$ and set $j^* := L$. (Here, $\mathsf{Extract}()$ of CombHT does not extract the element from the stashes.)
     (c) Run $\mathsf{Intersperse}^{(j^*-\ell)}_{2^{\ell+1}, 2^{\ell+1}, 2^{\ell+2}, \ldots, 2^{j^*}}(\mathbf{U})$ (Algorithm 6.4). Denote the output by $\tilde{\mathbf{U}}$. If $j = L$, then additionally do the following to shrink $\tilde{\mathbf{U}}$ to size $N = 2^L$:
        i. Run the tight compaction on $\tilde{\mathbf{U}}$ moving all real elements to the front. Truncate $\tilde{\mathbf{U}}$ to length N.
        ii. Run $\tilde{\mathbf{U}} \leftarrow \mathsf{IntersperseRD}_N(\tilde{\mathbf{U}})$ (Algorithm 6.6).
     (d) Rebuild the jth hash table with the 2^j elements from $\tilde{\mathbf{U}}$ via $T_j := \mathsf{CombHT.Build}(\tilde{\mathbf{U}})$ (Construction 8.7) and let $\mathsf{OF}_\mathsf{S}, \mathsf{CombS}_\mathsf{S}$ be the associated stashes (of size $O(\log \lambda)$ each). Mark $\mathsf{full}_j := 1$.

i. For each element (k, v) in the stash OF_{S}, run $D.\mathsf{Insert}(k, v)$.

ii. For each element (k, v) in the stash $\mathsf{CombS}_{\mathsf{S}}$, run $D.\mathsf{Insert}(k, v)$.

(e) For $i \in \{\ell + 1, \ldots, j - 1\}$, reset T_i to be empty structure and set $\mathsf{full}_i := 0$.

- **Output:** Return data^*.

We prove that our construction obliviously implements the ORAM functionality (Functionality 3.4) and analyze it amortized overhead.

Theorem 9.2. *Let $N \in \mathbb{N}$ be the capacity of ORAM and $\lambda \in \mathbb{N}$ be a security parameter. Assume a $\delta^{\mathcal{A}}_{\mathsf{PRF}}$-secure PRF. For any number of queries $T = T(N, \lambda) \geq N$, Construction 9.1 $(1 - T \cdot N^2 \cdot e^{-\Omega(\log \lambda \cdot \log\log \lambda)} - \delta^{\mathcal{A}}_{\mathsf{PRF}})$-obliviously implements the ORAM functionality. Moreover, the construction has $O\left(\log N \cdot \left(1 + \frac{\log N}{\log^2 \lambda}\right) + \log^9 \log \lambda\right)$ amortized time overhead.*

Proof. The proof of security is given in Appendix C.6 and we give the efficiency analysis next.

We may consider two cases:

- if $w < \log^3 \log \lambda$, which implies that $N < 2^{O(\log^3 \log \lambda)}$, we can use a perfect ORAM on N which yields an ORAM scheme with $O(\log^9 \log \lambda)$ overhead (see Theorem 4.8).
- therefore, henceforth it suffices to consider the case when $w \geq \log^3 \log \lambda$.

In each of the T Access operations, Steps 1–5 perform a single Lookup and Insert operation on the oblivious dictionary, and one Lookup on each $T_\ell, \ldots, T_L$. These operations require $O(\log^4 \log \lambda) + O(\log N)$ time. In Step 6, for every 2^ℓ requests of Access, one $\mathsf{Extract}$ and at most $O(2^\ell)$ Insert operations are performed on the oblivious dictionary D, and at most one $\mathsf{CombHT.Build}$ on $T_{\ell+1}$. These require $O\left(2^\ell \cdot \log^3(2^{\ell+1}) + 2^\ell \cdot \log^4(2^{\ell+1}) + 2^{\ell+1}\right) = O(2^\ell \cdot \log^4 \log \lambda)$ time. In addition, for each $j \in \{\ell + 1, \ldots, L\}$, for every 2^j requests of Access, at most one $\mathsf{Extract}$ is performed on T_j, one Build on T_{j+1}, one $\mathsf{Intersperse}^{(j-\ell)}_{2^{\ell+1}, 2^{\ell+1}, 2^{\ell+2}, \ldots, 2^j}$, one $\mathsf{InterspreseRD}_N$, and one tight compaction, all of which require linear time and thus the total time is $O(2^j + 2^j \cdot \frac{j}{\log^2 \lambda})$ by Build and $\mathsf{Extract}$ of CombHT (Theorem 8.8). Hence, over T requests, the amortized time is

$$\frac{1}{T}\left[\frac{T}{2^\ell} \cdot O\left(2^\ell \cdot \log^4 \log \lambda\right) + \sum_{j=\ell+1}^{L} \frac{T}{2^j} \cdot O\left(2^j \cdot \left(1 + \frac{j}{\log^2 \lambda}\right)\right)\right] = O\left(\log^4 \log \lambda + \log N \cdot \left(1 + \frac{\log N}{\log^2 \lambda}\right)\right).$$

□

As a corollary, we can now state a more general version:

Corollary 9.3 (Precise statement of Theorem 1.1)**.** *Let $N \in \mathbb{N}$ be the capacity of ORAM and $\lambda \in \mathbb{N}$ be a security parameter. Assume a $\delta^{\mathcal{A}}_{\mathsf{PRF}}$-secure PRF. For any number of queries $T = T(N, \lambda) \geq N$ and any constant $c > 1$, there is a $(1 - T \cdot N^2 \cdot \delta - \delta^{\mathcal{A}}_{\mathsf{PRF}})$-oblivious construction of an ORAM. Moreover, the construction has $O\left(\log N \cdot \left(1 + \frac{\log N}{\log^c(1/\delta)}\right) + \mathsf{poly}(\log\log(1/\delta))\right)$ amortized time overhead.*

Proof. There are two differences to the statement of Theorem 9.2. First, we replaced the term $e^{-\Omega(\log \lambda \cdot \log\log \lambda)}$ with δ. This means that $\Omega((1/\delta)^{1/\log\log(1/\delta)}) \leq \lambda \leq O(\log(1/\delta))$, and so we can replace the $O(\log N / \log^2 \lambda)$ term from Theorem 9.2 with $O(\log N \cdot \log^2 \log(1/\delta) / \log^2(1/\delta))$ and the $\mathsf{poly}(\log\log \lambda)$ term with $\mathsf{poly}(\log\log(1/\delta))$. Second, we generalize the exponent of the $\log^2 \lambda$ to any constant $c > 1$ (and this absorbes the $\log^c \log(1/\delta)$ factor). This is okay by relying on Corollary 8.9 to instantiate our ORAM's CombHT (which will make the ORAM's smallest level larger but still upper bounded by $\mathsf{poly}\log(\lambda)$). □

Remark 9.4 (Using More CPU Registers)**.** *Our construction can be slightly modified to obtain optimal amortized time overhead (up to constant factors) for* any number of CPU registers*, as given by the lower bound of Larsen and Nielsen [40]. Specifically, if the number of CPU registers is m, then we can achieve a scheme with $O(\log(N/m))$ amortized time overhead.*

If $m \in N^{1-\epsilon}$ for $\epsilon > 0$, then the lower bound still says that $\Omega(\log N)$ amortized time overhead is required so we can use Construction 9.1 without any change (and only utilize a constant number of CPU registers). For larger values of m (e.g., $m = O(N/\log N)$), we slightly modify Construction 9.1 as follows. Instead of storing levels $\ell = \lceil 11 \log\log\lambda \rceil$ through $L = \lceil \log N \rceil$ in the memory, we utilize the extra space in the CPU to store levels ℓ through $\ell_m \triangleq \lfloor \log m \rfloor$ while the rest of the levels (i.e., $\ell_m + 1$ through L) are stored in the memory, as in the above construction. The number of levels that we store in the memory is $O(\log N - \log m)) = O(\log(N/m))$ which is the significant factor in the overhead analysis (as the amortized time overhead per level is $O(1)$).

Acknowledgments

We are grateful to Hubert Chan, Kai-Min Chung, Yue Guo, and Rafael Pass for helpful discussions. This work is supported in part by a Simons Foundation junior fellow award, an AFOSR grant FA9550-15-1-0262, NSF grant CNS-1601879, an Office of Naval Research Young Investigator Program Award, a Packard Fellowship, a Sloan Fellowship, Google Faculty Research Awards, a VMware Research Award, and a Baidu Research Award.

References

[1] Miklós Ajtai, János Komlós, and Endre Szemerédi. An $O(n \log n)$ sorting network. In *ACM STOC*, pages 1–9, 1983.

[2] Arne Andersson, Torben Hagerup, Stefan Nilsson, and Rajeev Raman. Sorting in linear time? In *ACM STOC*, pages 427–436, 1995.

[3] Sanjeev Arora and Boaz Barak. *Computational Complexity - A Modern Approach*. Cambridge University Press, 2009.

[4] Sanjeev Arora, Frank Thomson Leighton, and Bruce M. Maggs. On-line algorithms for path selection in a nonblocking network (extended abstract). In *ACM STOC*, 1990.

[5] Kenneth E. Batcher. Sorting networks and their applications. In *American Federation of Information Processing Societies: AFIPS Conference Proceedings*, volume 32 of *AFIPS Conference Proceedings*, pages 307–314. Thomson Book Company, Washington D.C., 1968.

[6] Vincent Bindschaedler, Muhammad Naveed, Xiaorui Pan, XiaoFeng Wang, and Yan Huang. Practicing oblivious access on cloud storage: the gap, the fallacy, and the new way forward. In *ACM CCS*, pages 837–849, 2015.

[7] Elette Boyle and Moni Naor. Is there an oblivious RAM lower bound? In *ACM ITCS*, pages 357–368, 2016.

[8] Karl Bringmann, Fabian Kuhn, Konstantinos Panagiotou, Ueli Peter, and Henning Thomas. Internal DLA: Efficient Simulation of a Physical Growth Model. In *ICALP*, pages 247–258, 2014. Full version: http://people.mpi-inf.mpg.de/~kbringma/paper/2014ICALP.pdf (accessed: April 3rd, 2019).

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

[10] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *IEEE FOCS*, pages 136–145, 2001.

[11] Hubert Chan, Kai-Min Chung, and Elaine Shi. On the depth of oblivious parallel oram. In *ASIACRYPT*, 2017.

[12] T.-H. Hubert Chan, Yue Guo, Wei-Kai Lin, and Elaine Shi. Oblivious hashing revisited, and applications to asymptotically efficient ORAM and OPRAM. In *ASIACRYPT*, pages 660–690, 2017.

[13] T.-H. Hubert Chan, Yue Guo, Wei-Kai Lin, and Elaine Shi. Cache-oblivious and data-oblivious sorting and applications. In *SODA*, pages 2201–2220, 2018.

[14] T.-H. Hubert Chan, Kartik Nayak, and Elaine Shi. Perfectly secure oblivious parallel RAM. In *TCC*, pages 636–668, 2018.

[15] T.-H. Hubert Chan and Elaine Shi. Circuit OPRAM: unifying statistically and computationally secure orams and oprams. In *TCC*, pages 72–107, 2017.

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

[17] Jean claude Paul and Wilhelm Simon. Decision trees and random access machines. 1980.

[18] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*, pages 428–436. MIT Press, third edition, 2009.

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

[20] Martn Farach-Colton and Meng-Tsung Tsai. Exact Sublinear Binomial Sampling. *Algorithmica*, 73(4):637–651, December 2015.

[21] Alireza Farhadi, MohammadTaghi Hajiaghayi, Kasper Green Larsen, and Elaine Shi. Lower bounds for external memory integer sorting via network coding. In *ACM STOC*, 2019.

[22] P Feldman, J Friedman, and N Pippenger. Non-blocking networks. In *ACM STOC*, pages 247–254, 1986.

[23] R.A. Fisher and F. Yates. *Statistical Tables for Biological, Agricultural and Medical Research.* Oliver and Boyd, 1975.

[24] Christopher W Fletcher, Marten van Dijk, and Srinivas Devadas. A secure processor architecture for encrypted computation on untrusted programs. In *Proceedings of the seventh ACM workshop on Scalable trusted computing*, pages 3–8. ACM, 2012.

[25] Christopher W. Fletcher, Ling Ren, Albert Kwon, Marten van Dijk, and Srinivas Devadas. Freecursive ORAM: [nearly] free recursion and integrity verification for position-based oblivious RAM. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS*, pages 103–116. ACM, 2015.

[26] Ofer Gabber and Zvi Galil. Explicit constructions of linear-sized superconcentrators. *J. Comput. Syst. Sci.*, 22(3):407–420, 1981.

[27] Craig Gentry, Shai Halevi, Charanjit Jutla, and Mariana Raykova. Private database access with he-over-oram architecture. In *International Conference on Applied Cryptography and Network Security*, pages 172–191. Springer, 2015.

[28] Oded Goldreich. Towards a theory of software protection and simulation by oblivious rams. In *ACM STOC*, pages 182–194, 1987.

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

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

[31] Michael T. Goodrich. Zig-zag Sort: A Simple Deterministic Data-oblivious Sorting Algorithm Running in O(N Log N) Time. In *ACM STOC*, pages 684–693, 2014.

[32] Michael T. Goodrich and Michael Mitzenmacher. Privacy-preserving access of outsourced data via oblivious RAM simulation. In *ICALP*, pages 576–587, 2011.

[33] Michael T. Goodrich, Michael Mitzenmacher, Olga Ohrimenko, and Roberto Tamassia. Oblivious ram simulation with efficient worst-case access overhead. In *Proceedings of the 3rd ACM Workshop on Cloud Computing Security Workshop*, CCSW 11, page 95100, 2011.

[34] Michael T. Goodrich, Michael Mitzenmacher, Olga Ohrimenko, and Roberto Tamassia. Privacy-preserving group data access via stateless oblivious RAM simulation. In *SODA*, pages 157–167, 2012.

[35] Torben Hagerup and Hong Shen. Improved nonconservative sequential and parallel integer sorting. *Inf. Process. Lett.*, 36(2):57–63, 1990.

[36] Johan Håstad, Russell Impagliazzo, Leonid A. Levin, and Michael Luby. A pseudorandom generator from any one-way function. *SIAM Journal on Computing*, 28(4):1364–1396, 1999.

[37] Shuji Jimbo and Akira Maruoka. Expanders obtained from affine transformations. *Combinatorica*, 7(4):343–355, 1987.

[38] Adam Kirsch, Michael Mitzenmacher, and Udi Wieder. More robust hashing: Cuckoo hashing with a stash. *SIAM J. Comput.*, 39(4):1543–1561, 2009.

[39] Eyal Kushilevitz, Steve Lu, and Rafail Ostrovsky. On the (in)security of hash-based oblivious RAM and a new balancing scheme. In *SODA*, pages 143–156, 2012.

[40] Kasper Green Larsen and Jesper Buus Nielsen. Yes, there is an oblivious RAM lower bound! In *CRYPTO*, pages 523–542, 2018.

[41] Frank Thomson Leighton, Yuan Ma, and Torsten Suel. On probabilistic networks for selection, merging, and sorting. *Theory Comput. Syst.*, 30(6):559–582, 1997.

[42] Zongpeng Li and Baochun Li. Network coding : The case of multiple unicast sessions. 2004.

[43] Wei-Kai Lin, Elaine Shi, and Tiancheng Xie. Can we overcome the $n \log n$ barrier for oblivious sorting? In *SODA*, 2019.

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

[45] Steve Lu and Rafail Ostrovsky. Distributed oblivious RAM for secure two-party computation. In *TCC*, pages 377–396, 2013.

[46] Martin Maas, Eric Love, Emil Stefanov, Mohit Tiwari, Elaine Shi, Krste Asanovic, John Kubiatowicz, and Dawn Song. PHANTOM: practical oblivious computation in a secure processor. In *ACM CCS*, pages 311–324, 2013.

[47] Grigorii Aleksandrovich Margulis. Explicit constructions of concentrators. *Problemy Peredachi Informatsii*, 9(4):71–80, 1973.

[48] John C. Mitchell and Joe Zimmerman. Data-oblivious data structures. In *31st International Symposium on Theoretical Aspects of Computer Science STACS*, pages 554–565, 2014.

[49] Moni Naor. Bit commitment using pseudo-randomness. In *CRYPTO*, pages 128–136, 1989.

[50] Rafail Ostrovsky and Victor Shoup. Private information storage. In *ACM STOC*, pages 294–303, 1997.

[51] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *J. Algorithms*, 51(2):122–144, May 2004.

[52] Sarvar Patel, Giuseppe Persiano, Mariana Raykova, and Kevin Yeo. Panorama: Oblivious RAM with logarithmic overhead. In *IEEE FOCS*, 2018.

[53] Mark S. Pinsker. On the complexity of a concentrator. In *7th International Teletraffic Conference*, 1973.

[54] Nicholas Pippenger. Superconcentrators. *SIAM J. Comput.*, 6(2):298–304, 1977.

[55] Nicholas Pippenger. Self-routing superconcentrators. *J. Comput. Syst. Sci.*, 52(1):53–60, 1996.

[56] 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 *The 40th Annual International Symposium on Computer Architecture, ISCA*, pages 571–582, 2013.

[57] Elaine Shi, T.-H. Hubert Chan, Emil Stefanov, and Mingfei Li. Oblivious RAM with $O((\log N)^3)$ worst-case cost. In *ASIACRYPT*, pages 197–214, 2011.

[58] Emil Stefanov and Elaine Shi. Oblivistore: High performance oblivious cloud storage. In *IEEE S&P*, pages 253–267, 2013.

[59] Emil Stefanov, Elaine Shi, and Dawn Xiaodong Song. Towards practical oblivious RAM. In *19th Annual Network and Distributed System Security Symposium, NDSS*, 2012.

[60] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher W. Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: an extremely simple oblivious RAM protocol. In *ACM CCS*, pages 299–310, 2013.

[61] Leslie G. Valiant. Graph-theoretic properties in computational complexity. *J. Comput. Syst. Sci.*, 13(3):278–285, December 1976.

[62] Xiao Wang, T.-H. Hubert Chan, and Elaine Shi. Circuit ORAM: on tightness of the goldreich-ostrovsky lower bound. In *ACM CCS*, pages 850–861, 2015.

[63] Xiao Shaun Wang, Yan Huang, T.-H. Hubert Chan, Abhi Shelat, and Elaine Shi. SCORAM: oblivious RAM for secure computation. In *ACM CCS*, pages 191–202, 2014.

[64] Wikipedia. Bitonic sorter. https://en.wikipedia.org/w/index.php?title=Bitonic_sorter. Accessed: Sep 22, 2019.

[65] Wikipedia. Sorting network. https://en.wikipedia.org/wiki/Sorting_network. Accessed: Feb 14, 2020.

[66] Peter Williams, Radu Sion, and Alin Tomescu. Privatefs: A parallel oblivious file system. In *ACM CCS*, 2012.

[67] Samee Zahur, Xiao Shaun Wang, Mariana Raykova, Adria Gascón, Jack Doerner, David Evans, and Jonathan Katz. Revisiting square-root ORAM: efficient random access in multi-party computation. In *IEEE S&P*, pages 218–234, 2016.

A Comparison with Prior Works

In the introduction, we have presented the overhead of our construction as a function of N and assuming that $\lambda \leq N \leq T \leq \mathsf{poly}(\lambda)$ for any fixed polynomial $\mathsf{poly}(\cdot)$, where N is the size of the memory, and T is a bound on the number of accesses. In order to have a more meaningful comparison with previous works, we restate our result as well as some prior results using two parameters, N and δ, as above, and no longer assume that $\lambda \leq N \leq T \leq \mathsf{poly}(\lambda)$. Our construction achieves:

Theorem A.1. *Assume the existence of a pseudo-random function (PRF) family. Let N denote the maximum number of blocks stored by the ORAM.*

For any $\delta > 0$, and any constant $c \geq 1$, there exists an ORAM scheme with

$$O\left(\log N \cdot \left(1 + \frac{\log N}{\log^c(1/\delta)}\right) + \mathsf{poly}\log\log\left(\frac{1}{\delta}\right)\right)$$

amortized overhead, and for every probabilistic polynomial-time adversary $\mathcal{A}$ adaptively issuing at most T requests, the ORAM's failure probability is at most $T^3 \cdot \delta + \delta^{\mathcal{A}}_{\mathrm{prf}}$ where $\delta^{\mathcal{A}}_{\mathrm{prf}}$ denotes the probability that $\mathcal{A}$ breaks PRF security.

Previous works. We state other results with the same parameters N, δ, for the best of our understanding. In all results stated below we assume $O(1)$ CPU private registers and use δ to denote the ORAM's *statistical* failure probability; however, keep in mind that all computationally secure results below have an additional *additive* failure probability related to the PRF's security.

Goldreich and Ostrovsky [28, 30] showed a computationally secure ORAM construction with $O\left(\log^2 N \cdot \log(1/\delta)\right)$ overhead. Their work proposed the elegant hierarchical paradigm for constructing ORAMs and subsequent works have improved the hierarchical paradigm leading to asymptotically better results. The work of [12] (see also [32, 39]) showed how to improve Goldreich and

Ostrovsky's construction to $O\left(\log^2 N/\log\log N + \log(1/\delta)\cdot\log\log(1/\delta)\right)$ overhead. The work of Patel et al. [52] achieved $O\left(\log N\cdot\left(\log\log(1/\delta)+\frac{\log N}{\log(1/\delta)}\right)+\log(1/\delta)\cdot\log\log(1/\delta)\right)$ overhead (to the best of our understanding).

Besides the hierarchical paradigm, Shi et al. [57] propose the tree-based paradigm for constructing ORAMs. Subsequent works [15,16,60,62] have improved tree-based constructions, culminating in the works of Circuit ORAM [62] and Circuit OPRAM [15]. Circuit ORAM [62] achieves $O\big(\log N\cdot(\log N+\log(1/\delta))\big)$ overhead and statistical security. Subsequently, the Circuit OPRAM work [15] showed (as a by-product of their main result on OPRAM) that we can construct a *statistically* secure ORAM with $O\left(\log^2 N+\log(1/\delta)\right)$ overhead (by merging the stashes of all recursion depths in Circuit ORAM) and a *computationally* secure ORAM with $O\left(\log^2 N/\log\log N+\log(1/\delta)\right)$ overhead (by additionally adopting a metadata compression trick originally proposed by Fletcher et al. [25]).

B Details on Oblivious Cuckoo Assignment

Recall that the input of Cuckoo assignment is the array of the two choices, $\mathbf{I}=((u_1,v_1),\ldots(u_n,v_n))$, and the output is an array $\mathbf{A}=\{a_1,\ldots a_n\}$, where $a_i\in\{u_i,v_i,\texttt{stash}\}$ denotes that the i-th ball k_i is assigned to either bin u_i, or bin v_i, or the secondary array of stash. We say that a Cuckoo assignment $\mathbf{A}$ is *correct* iff it holds that (i) each bin is assigned to at most one ball, and (ii) the number of balls in the stash is minimized.

To compute $\mathbf{A}$, the array of choices $\mathbf{I}$ is viewed as a bipartite multi-graph $G=(U\cup V,E)$, where $U=\{u_i\}_{i\in[n]}$, $V=\{v_i\}_{i\in[n]}$, E is the multi-set $\{(u_i,v_i)\}_{i\in[n]}$, and the ranges of u_i and v_i are disjoint. Given G, the Cuckoo assignment algorithm performs an oblivious breadth-first search (BFS) such that traverses a tree for each connected component in G. In addition, the BFS performs the following for each edge $e\in E$: e is marked as either a *tree edge* or a *cycle edge*, e is tagged with the root $r\in U\cup V$ of the connected component of e, and e is additionally marked as pointing toward either *root* or *leaf* if e is a tree edge. Note that all three properties can be obtained in the standard tree traversal. Given such marking, the idea to compute $\mathbf{A}$ is to assign each tree edge $e=(u_i,v_i)$ *toward the leaf side*, and there are three cases for any connected component:

(1) If there is no cycle edge in the connected component, perform the following. If $e=(u_i,v_i)$ points toward a leaf, then assign $a_i=v_i$; otherwise, assign $a_i=u_i$.
(2) If there is exactly one cycle edge in the connected component, traverse from the cycle edge up to the root using another BFS, reverse the pointing of every edge on the path from the cycle edge to the root, and then apply the assignment of (1).
(3) If there are two or more cycle edges in the connected component, throw extra cycle edges to the stash by assigning $a_i=\texttt{stash}$, and then apply the assignment of (2).

The above operations take constant passes of sorting and BFS, and hence it remains to implement an oblivious BFS efficiently.

Recall that in a standard BFS, we start with a root node r and expand to nodes at depth 1. Then, iteratively we expands all nodes at depth i to nodes of depth $i+1$ until all nodes are expanded. Any cycle edges is detected when two or more nodes expand to the same node (because any cycle in a bipartite graph must consist of a even number of edges). We say the nodes at depth i is the *i-th front* and the expanding is the *i-th iteration*. To do it obliviously, the oblivious BFS performs the maximum number of iterations, and, in the i-th iteration, it touches all nodes, yet only the i-th front is actually expanded. Each iteration is accomplished by sorting and grouping

adjacent edges and then updating the marking within each group.[15] Note that the oblivious BFS does not need to know any connected components in advance. It simply expands all nodes in the beginning, and then, a front "includes" nodes in another front when the two meet and the first front has a root node that precedes the other root. Such BFS is not efficient as the maximum number of iterations is n, and each iteration takes several sorting on $O(n)$ elements.

To achieve efficiency, the intuition is that in the random bipartite graph G, with overwhelming probability, (i) the largest connected component in G is small, and (ii) there are many small connected components such that the BFS finishes in a few iterations. The intuition is informally stated by the following two tail bounds, where $\gamma < 1$ and $\beta < 1$ are constants such that depends only on the Cuckoo constant, $\mathsf{c}_{\mathsf{cuckoo}}$.

1. For every integer s, the size of the largest connected component of G is greater than s with probability $O(\gamma^{-s})$.
2. Conditioning on the largest component is at most s, the following holds. For any integer t, let C_t be the total number of edges of all components such that the size is at least t. Let $c = 1/7$. If t satisfies that $n\beta^t \geq \Theta(n^{1-c})$, then $C_k = O(n\beta^k)$ holds with probability $2^{-\Omega\left(\frac{n^{1-2c}}{s^4}\right)}$.

Using such tail bounds, the BFS is pre-programmed in the following way. The second tail bound says that after the t-th iteration of the BFS, we can safely eliminate $(1-\beta)n$ edges that is (with high probability) in components of size at most t until there are $\Theta(n^{1-c})$ edges remaining. Then, using the first tail bound, it suffices to run the BFS for s additional iterations on the remaining edges to figure out the remaining assignment (with overwhelming probability). To achieve failure probability δ, a standard choice is $s = \log \frac{1}{\delta}$ [12, 32, 39].

Therefore, the access pattern of such oblivious BFS is pre-determined by constants γ, β, δ, which does not depend on the input $\mathbf{I}$. The tail bounds incurs failure in correctness for a δ fraction among all $\mathbf{I}$, and then it is fixed by checking and applying perfectly correct but non-oblivious algorithm, which incurs loss in obliviousness. This concludes the construction of cuckooAssign at a very high level, and we have the following lemma.

Lemma B.1. *Let n be the input size. Let the input of two choices be sampled uniformly at random. Then, the cuckoo assignment runs in time $O(T(n)+\log \frac{1}{\delta} \cdot T(n^{6/7}))$ except with probability $\delta + \log n \cdot 2^{-\Omega(\frac{n^{5/7}}{\log^4 1/\delta})}$, where $T(m)$ is the time to sort m integers each of $O(\log m)$ bits.*

Assuming $n = \Omega(\log^8 \frac{1}{\delta})$ and plugging in $T(m) = O(m \log m)$, we have the standard statement of $O(n \log n)$ time and δ failure probability [12, 32, 39].

C Deferred Proofs

C.1 Proof of Theorem 5.2

To prove Theorem 5.2 we need several standard definitions and results. Given $G = (V, E)$ and a set of vertices $U \subset V$, we let $\Gamma(U)$ be the set of all vertices in V which are adjacent to a vertex in U (namely, $\Gamma(U) = \{v \in V \mid \exists u \in U, (u, v) \in E\}$).

[15] If there is a tie in the sorting of edges, we resolve it by the ordering of edges in $\mathbf{I}$. This resolution was arbitrary in Chan et al. [12], which is insufficient in our case. Here, we want it to be decided based on the original ordering as it implies that the assignment $\mathbf{A}$ is determined given (only) the input $\mathbf{I}$. We called this the *indiscrimination* property in Remark 4.13.

Definition C.1 (The parameter $\lambda(G)$ [3, Definition 21.2]). *Given a d-regular graph G on n vertices, we let $A = A(G)$ be the matrix such that for every two vertices u and v of G, it holds that $A_{u,v}$ is equal to the number of edges between u and v divided by d. (In other words, A is the adjacency matrix of G multiplied by $1/d$.) The parameter $\lambda(A)$, denoted also as $\lambda(G)$, is:*

$$\lambda(A) = \max_{\mathbf{v}\in\mathbf{1}^\perp, \|\mathbf{v}\|_2=1} \|A\mathbf{v}\|_2,$$

where $\mathbf{1}^\perp = \{\mathbf{v} \mid \sum \mathbf{v}_i = 0\}$.

Lemma C.2 (Expander mixing lemma [3, Definition 21.11]). *Let $G = (V, E)$ be a d-regular n-vertex graph. Then, for all sets $S, T \subseteq V$, it holds that*

$$\left| e(S,T) - \frac{d}{n} \cdot |S| \cdot |T| \right| \leq \lambda(G) \cdot d \cdot \sqrt{|S| \cdot |T|},$$

Proof of Theorem 5.2. Recall the bipartite graph of Margulis [47]. Fix a positive $M \in \mathbb{N}$. The left and right vertex sets $L = R = [M] \times [M]$. A left node (x, y) is connected to $(x, y), (x, x+y), (x, x+y+1), (x+y, y), (x+y+1, y)$ (all arithmetic is modulo M). We let G_N be the resulting graph that has N vertices on each side.

It is known (Margulis [47], Gabber and Galil [26], and Jimbo and Maruoka [37]) that for every n which is a square (i.e., of the form $n = i^2$ for some $i \in \mathbb{N}$), G_n is 5-regular and $\lambda(G_n) \in (0, 1)$ is constant. Also, the neighbors of each vertex can be computed via $O(1)$ elementary operations so the entire edge set of G_n can be computed in linear time in n. To boost $\lambda(G_n)$ to satisfy $\lambda(G_n) < \epsilon$, we consider the k-th power of G_n. This yields a 5^k-regular graph G_n^k on n vertices whose λ parameter is $\lambda(G_n)^k$. By choosing k to be a sufficiently large constant such that $\lambda(G_n)^k < \epsilon$, we obtain the resulting (5^k)-regular graph. The entire edge set of this graph can also be computed in $O(1)$ time since moving from the $(i-1)$-th to the i-th power requires $O(1)$ operations for each of the 5 edges that replace an edge – so the total time per vertex is $\sum_{i=1}^{k} O(5^i) = O(5^k) = O(1)$.

This completes the required construction for sizes $2^2, 2^4, 2^6, \ldots$ (i.e., for even powers of 2). We can fill in the odd powers by padding. Given the graph G_n^k, we can obtain the required graph for $2n$ vertices on each side by considering 4 copies of G_n^k and connecting them, that is, performing direct product of G_n^k with the complete bipartite graph on two sets of two vertices. The resulting graph G_{2n}^k has $2n$ vertices on each side, it is $(2 \cdot 5^k)$-regular and $\lambda(G_{2n}^k) = \lambda(G_n) < \epsilon$. The theorem now follows by applying the Expander Mixing Lemma (Lemma C.2). □

C.2 Deferred Proofs from Section 6

Proof of Claim 6.3. We build a simulator that receives only $n := n_0 + n_1$ and simulates the access pattern of Algorithm 6.1. The simulating of the generation of the array Aux is straightforward, and consists of modifying two counters (that can be stored at the client side) and just a sequential write of the array Aux. The rest of the algorithm is deterministic and the access pattern is completely determined by the size n. Thus, it is straightforward to simulate the algorithm deterministically.

We next prove that the output distribution of the algorithm is identical to that of the ideal functionality. In the ideal execution, the functionality simply outputs an array $\mathbf{B}$, where $\mathbf{B}[i] = (\mathbf{I}_0 \| \mathbf{I}_1)[\pi(i)]$ and π is a uniformly random permutation on n elements. In the real execution, we assume that the two arrays were first randomly permuted, and let π_0 and π_1 be the two permutations.[16] Let $\mathbf{I}'$ be an array define as $\mathbf{I}' := \pi_0(\mathbf{I}_0) \| \pi_1(\mathbf{I}_1)$. The algorithm then runs the Distribution

[16] Recall that according to our definition, we translate an "input assumption" to a protocol in the hybrid model in which the protocol first invoke a functionality that guarantee that the input assumption holds. In our case, the functionality receives the input array $\mathbf{I}_0 \| \mathbf{I}_1$ and the parameters n_0, n_1, chooses two random permutations π_0, π_1 and permute the two arrays $\mathbf{I}_0, \mathbf{I}_1$.

on $\mathbf{I}'$ and Aux, where Aux is a uniformly random binary array of size n that has n_0 0's and n_1 1's, and ends up with the output array $\mathbf{B}$ such that for all positions i, the label of the element $\mathbf{B}[i]$ is $\mathsf{Aux}[i]$. Note that $\mathsf{Distribution}$ is not a stable, so this defines some arbitrary mapping $\rho\colon [n] \to [n]$. Hence, the algorithm outputs an array $\mathbf{B}$ such that $\mathbf{B}[i] = \rho^{-1}(\mathbf{I}'[i])$. We show that if we sample Aux, π_0, and π_1, as above, the resulting permutation is a uniform one.

To this end, we show that (1) Aux is distributed according to the distribution above, (2) the total number of different choices for $(\mathsf{Aux}, \pi_0, \pi_1)$ is $n!$ (exactly as for a uniform permutation), and (3) any two choices of $(\mathsf{Aux}, \pi_0, \pi_1) \neq (\mathsf{Aux}', \pi_0', \pi_1')$ result with a different permutation. This completes our proof.

For (1) we show that the implementation of the sampling of the array in Step 1 in Algorithm 6.1 is equivalent to uniformly sampling an array of size $n_0 + n_1$ among all arrays of size $n_0 + n_1$ with n_0 0's and n_1 1's. Fix any array $X \in \{0,1\}^n$ that consists of n_0 0's followed by n_1 1's. It is enough to show that

$$\Pr\left[\forall i \in [n]\colon \mathsf{Aux}[i] = X[i]\right] = \frac{1}{\binom{n}{n_0}}.$$

This equality holds since the probability to get the bit $b = X[i]$ in $\mathsf{Aux}[i]$ only depends on i and on the number of b's that happened before iteration i. Concretely,

$$\begin{aligned}\Pr\left[\forall i \in [n]\colon \mathsf{Aux}[i] = X[i]\right] &= \left(\frac{n_0!}{n \cdot \ldots \cdot (n-n_0)}\right) \cdot \left(\frac{n_1!}{(n-n_0-1) \cdot \ldots \cdot 1}\right) \\ &= \frac{n_0! \cdot n_1!}{n!} = \frac{1}{\binom{n}{n_0}}.\end{aligned}$$

For (2), the number of possible choices of $(\mathsf{Aux}, \pi_0, \pi_1)$ is

$$\binom{n}{n_0} \cdot n_0! \cdot n_1! = \frac{(n_0+n_1)!}{n_0! \cdot n_1!} \cdot n_0! \cdot n_1! = n!\ .$$

For (3), consider two different triples $(\mathsf{Aux}, \pi_0, \pi_1)$ and $(\mathsf{Aux}', \pi_0', \pi_1')$ that result with two permutations ψ and ψ', respectively. If $\mathsf{Aux}(i) \neq \mathsf{Aux}'(i)$ for some $i \in [n]$ and without loss of generality $\mathsf{Aux}(i) = 0$, then $\psi(i) \in \{1, \ldots, n_0\}$ while $\psi'(i) \in \{n_0+1, \ldots, n\}$. Otherwise, if $\mathsf{Aux}(i) = \mathsf{Aux}'(i)$ for every $i \in [n]$, then there exist $b \in \{0,1\}$ and $j \in [n_b]$ such that $\pi_b(j) \neq \pi_b'(j)$. Since the tight compaction circuit C_n is fixed given Aux, the jth input in $\mathbf{I}_b$ is mapped in both cases to the same location of the bit b in Aux. Denote the index of this location by j'. Thus, $\psi(j') = \pi_b(i)$ while $\psi'(j') = \pi_b'(i)$ which means that $\psi \neq \psi'$, as needed.

The implementation has $O(n)$ time since there are three main steps and each can be implemented in $O(n)$ time. Step 1 has time $O(n)$ since there are n coin flips and each can be done with $O(1)$ time (by just reading a word from the random tape). Steps 2 is only marking n elements, and Step 3 can be implemented in $O(n)$ time by Theorem 5.21. □

Proof of Claim 6.5. The simulator that receives $n_1, \ldots, n_k$ runs the simulator of $\mathsf{Intersperse}$ for $k-1$ times with the right lengths, as in the description of the algorithm. The indistinguishability follows immediately from the indistinguishability of $\mathsf{Intersperse}$. For functionality, note that whenever $\mathsf{Intersperse}$ is applied, it holds that both of its inputs are randomly shuffled which means that the input assumption of $\mathsf{Intersperse}$ holds. Thus, the final array $\mathbf{B}$ is a uniform permutation of $\mathbf{I}_1 \| \ldots \| \mathbf{I}_k$.

Since the time of $\mathsf{Intersperse}_n$ is linear in n, the time required in the i-th iteration of $\mathsf{Intersperse}^{(k)}_{n_1,\ldots,n_k}$ is $O\left(\sum_{j=1}^{i} n_j\right)$. Namely, we pay $O(n_1)$ in $k-1$ iterations, $O(n_2)$ in $k-2$ iterations, and so on. Overall, the time is bounded by $O\left(\sum_{i=1}^{k}(k-i+1) \cdot n_i\right)$, as required. □

Proof of Claim 6.7. We build a simulator that receives only the size of $\mathbf{I}$ and simulates the access pattern of Algorithm 6.6. The simulation of the first and second steps is immediate since they are completely deterministic. The simulating of the execution of $\mathsf{Intersperse}_n$ is implied by Claim 6.3.

The proof that the output distribution of algorithm is identical to that of the ideal functionality follows immediately from Claim 6.3. Indeed, after compaction and counting the number of real elements, we execute $\mathsf{Intersperse}_n$ with two arrays $\mathbf{I}'_R$ and $\mathbf{I}'_D$ of total size n, where $\mathbf{I}'_R$ consists of all the n_R real elements and $\mathbf{I}'_D$ consists of all the dummy elements. The array $\mathbf{I}'_R$ is uniformly shuffled to begin with by the input assumption, and the array $\mathbf{I}'_D$ consists of identical elements, so we can think of it as if they are randomly permuted. So, the input assumption of $\mathsf{Intersperse}_n$ (see Algorithm 6.1) holds and thus the output is guaranteed to be randomly shuffled (by Claim 6.3).

The implementation runs in $O(n)$ time since the first two steps take $O(n)$ time (by Theorem 5.1) and $\mathsf{Intersperse}_n$ itself runs in $O(n)$ time (by Claim 6.3). □

C.3 Proof of Security of BigHT (Theorem 7.2)

We view our construction in a hybrid model, in which we have ideal implementations of the underlying building blocks: an oblivious hash table for each bin (implemented via naïveHT, as in Section 4.4), an oblivious Cuckoo hashing scheme (Section 4.5), an oblivious tight compaction algorithm (Section 5), and an algorithm for sampling bin loads (Section 4.7). Since the underlying Cuckoo hash scheme (on $\epsilon n = n/\log\lambda$ elements with stash of size $O(\log\lambda)$) is ideal we have to take into account the probability that it fails: at most $O(\delta) + \delta^{\mathcal{A}}_{\text{prf}} = e^{-\Omega(\log\lambda\cdot\log\log\lambda)} + \delta^{\mathcal{A}}_{\text{prf}}$, where $\delta = e^{-\log\lambda\cdot\log\log\lambda}$. The failure probability of sampling the bin loads is $nB\cdot\delta \le n^2\cdot e^{-\log\lambda\cdot\log\log\lambda}$. These two terms are added to the error probability of the scheme. Note that our tight compaction and naïveHT are perfectly oblivious.

We describe a simulator Sim that simulates the access patterns of the Build, Lookup, and Extract operations of BigHT:

- **Simulating Build.** Upon receiving an instruction to simulate Build with security parameter 1^λ and a list of size n, the simulator runs the real algorithm Build on input 1^λ and a list that consists of n dummy elements. It outputs the access pattern of this algorithm. Let $(\mathsf{OBin}_1,\ldots,\mathsf{OBin}_B,\mathsf{OF},\mathsf{sk})$ be the output state. The simulator stores this state.
- **Simulating Lookup.** When the adversary submits a Lookup command with a key k, the simulator simulates an execution of the algorithm Lookup on input $\perp$ (i.e., a dummy element) with the state $(\mathsf{OBin}_1,\ldots,\mathsf{OBin}_B,\mathsf{OF},\mathsf{sk})$ (which was generated while simulating the the Build operation).
- **Simulating Extract.** When the adversary submits an Extract command, the simulator executes the real algorithm with its stored internal state $(\mathsf{OBin}_1,\ldots,\mathsf{OBin}_B,\mathsf{OF},\mathsf{sk})$.

We prove that no adversary can distinguish between the real and ideal executions. Recall that in the ideal execution, with each command that the adversary outputs, it receives back the output of the functionality and the access pattern of the simulator, where the latter is simulating the access pattern of the execution of the command on dummy elements. On the other hand, in the real execution, the adversary sees the access pattern and the output of the algorithm that implements the functionality. The proof is via a sequence of hybrid experiments.

Experiment $\mathsf{Hyb}_0(\lambda)$. This is the real execution. With each command that the adversary submits to the experiment, the real algorithm is being executed, and the adversary receives the output of the execution together with the access pattern as determined by the execution of the algorithm.

Experiment $\mathsf{Hyb}_1(\lambda)$**.** This experiment is the same as Hyb_0, except that instead of choosing a PRF key sk, we use a truly random function $\mathcal{O}$. That is, instead of calling to $\mathsf{PRF}_{\mathsf{sk}}(\cdot)$ in Step 3 of Build and Step 4 of the function Lookup, we call $\mathcal{O}(\mathsf{sk}\|\cdot)$.

The following claim states that due to the $\delta^{\mathcal{A}}_{\mathsf{PRF}}$-security of the PRF, experiments Hyb_0 and Hyb_1 are computationally indistinguishable. The proof of this claim is standard.

Claim C.3. *For any probabilistic polynomial-time adversary $\mathcal{A}$, it holds that*

$$|\Pr[\mathsf{Hyb}_0(\lambda) = 1] - \Pr[\mathsf{Hyb}_1(\lambda) = 1]| \leq \delta^{\mathcal{A}}_{\mathsf{PRF}}(\lambda).$$

Experiment $\mathsf{Hyb}_2(\lambda)$**.** This experiment is the same as $\mathsf{Hyb}_1(\lambda)$, except that with each command that the adversary submits to the experiment, both the real algorithm is being executed as well as the functionality. The adversary receives the access pattern of the execution of the algorithm, yet the output comes from the functionality.

In the following claim, we show that the initial secret permutation and the random oracle, guarantee that experiments Hyb_1 and Hyb_2 are identical.

Claim C.4. $\Pr[\mathsf{Hyb}_1(\lambda) = 1] = \Pr[\mathsf{Hyb}_2(\lambda) = 1]$.

Proof. Recall that we assume that the lookup queries of the adversary are non-recurring. Our goal is to show that the output distribution of the extract procedure is a uniform permutation of the unvisited items even given the access patter of the previous Build and Lookup operations. By doing so, we can replace the Extract procedure with the ideal $\mathcal{F}^n_{\mathsf{HT}}$.Extract functionality which is exactly the difference between $\mathsf{Hyb}_1(\lambda)$ and $\mathsf{Hyb}_2(\lambda)$.

Consider a sequence of operations that the adversary makes. Let us denote by $\mathbf{I}$ the set of elements with which it invokes Build and by $k^*_1, \ldots, k^*_m$ the set of keys with which it invokes Lookup. Finally, it invokes Extract. We first argue that the output of $\mathcal{F}^n_{\mathsf{HT}}$.Extract consists of the same elements as that of Extract. Indeed, both $\mathcal{F}^n_{\mathsf{HT}}$.Lookup and Lookup mark every visited item so when we execute Extract, the same set of elements will be in the output.

We need to argue that the distribution of the permutation of unvisited items in the *input* of Extract is uniformly random. This is enough since Extract performs IntersperseRD which shuffles the reals and dummies to obtain a uniformly random permutation overall (given that the reals were randomly shuffled to begin with). Fix an access pattern observed during the execution of Build and Lookup. We show, by programming the random oracle and the initial permutation appropriately (while not changing the access pattern), that the permutation of the unvisited elements is uniformly distributed.

Consider tuples of the form $(\pi_{\mathsf{in}}, \mathcal{O}, R, \mathsf{T}, \pi_{\mathsf{out}})$, where (1) π_{in} is the permutation performed on $\mathbf{I}$ by the input assumption (prior to Build), (2) $\mathcal{O}$ is the random oracle, (3) R is the internal randomness of all intermediate functionalities and of the balls into bins choices of the dummy elements; (4) T is the access pattern of the entire sequence of commands $(\mathsf{Build}(\mathbf{I}), \mathsf{Lookup}(k^*_1), \ldots, \mathsf{Lookup}(k^*_m))$, and (5) π_{out} is the permutation on $\mathbf{I}' = \{(k, v) \in \mathbf{I} \mid k \notin \{k^*_1, \ldots, k^*_m\}\}$ which is the input to Extract. The algorithm defines a deterministic mapping $\psi_R(\pi_{\mathsf{in}}, \mathcal{O}) \to (\mathsf{T}, \pi_{\mathsf{out}})$.

To gain intuition, consider arbitrary R, π_{in}, and $\mathcal{O}$ such that $\psi_R(\pi_{\mathsf{in}}, \mathcal{O}) \to (\mathsf{T}, \pi_{\mathsf{out}})$ and two distinct existing keys k_i and k_j that are not queried during the Lookup stage (i.e., $k_i, k_j \notin \{k^*_1, \ldots, k^*_m\}$). We argue that from the point of view of the adversary, having seen the access pattern and all query results, he cannot distinguish whether $\pi_{\mathsf{out}}(i) < \pi_{\mathsf{out}}(j)$ or $\pi_{\mathsf{out}}(i) > \pi_{\mathsf{out}}(j)$. The argument will naturally generalize to arbitrary unqueried keys and an arbitrary ordering.

To this end, we show that there is π'_{in} and $\mathcal{O}'$ such that $\psi_R(\pi'_{\mathsf{in}}, \mathcal{O}') \to (\mathsf{T}, \pi'_{\mathsf{out}})$, where $\pi'_{\mathsf{out}}(\ell) = \pi_{\mathsf{out}}(\ell)$ for every $\ell \notin \{i, j\}$, and $\pi'_{\mathsf{out}}(i) = \pi_{\mathsf{out}}(j)$ and $\pi'_{\mathsf{out}}(j) = \pi_{\mathsf{out}}(i)$. That is, the access pattern is

exactly the same and the output permutation switches the mappings of k_i and k_j. The permutation π'_{in} is the same as π_{in} except that $\pi'_{\mathsf{in}}(i) = \pi_{\mathsf{in}}(j)$ and $\pi'_{\mathsf{in}}(j) = \pi_{\mathsf{in}}(i)$, and $\mathcal{O}'$ is the same as $\mathcal{O}$ except that $\mathcal{O}'(k_i) = \mathcal{O}(k_j)$ and $\mathcal{O}'(k_j) = \mathcal{O}(k_i)$. This definition of π'_{in} together with $\mathcal{O}'$ ensure, by our construction, that the observed access pattern remains exactly the same. The mapping is also reversible so by symmetry all permutations have the same number of configurations of π_{in} and $\mathcal{O}$.

For the general case, one can switch from any π_{out} to any (legal) π'_{out} by changing only π_{in} and $\mathcal{O}$ at locations that correspond to unvisited items. We define

$$\pi'_{\mathsf{in}}(i) = \pi_{\mathsf{in}}({\pi_{\mathsf{out}}}^{-1}(\pi'_{\mathsf{out}}(i))) \quad \text{and} \quad \mathcal{O}'(k_i) = \mathcal{O}(k_{\pi_{\mathsf{in}}({\pi_{\mathsf{out}}}^{-1}(\pi'_{\mathsf{out}}(i)))}).$$

This choice of π'_{in} and $\mathcal{O}'$ do not change the observed access pattern and result with the output permutation π'_{out}, as required. By symmetry, the resulting mapping between different $(\pi'_{\mathsf{in}}, \mathcal{O}')$ and π'_{out} is regular (i.e., each output permutation has the same number of ways to reach to) which completes the proof. □

Experiment $\mathsf{Hyb}_3(\lambda)$. This experiment is the same as $\mathsf{Hyb}_2(\lambda)$, except that we modify the definition of $\mathsf{Extract}$ to output a list of dummy elements. This is implemented by modifying each $\mathsf{Obin}_i.\mathsf{Extract}()$ to return a list of dummy elements (for each $i \in [B]$), as well as $\mathsf{OF}.\mathsf{Extract}()$. We also stop marking elements that were searched for during Lookup.

Recall that in this hybrid experiment the output of $\mathsf{Extract}$ is given to the adversary by the functionality, and not by the algorithm. Thus, the change we made does not affect the view of the adversary which means that experiments Hyb_2 and Hyb_3 are identical.

Claim C.5. $\Pr[\mathsf{Hyb}_2(\lambda) = 1] = \Pr[\mathsf{Hyb}_3(\lambda) = 1]$.

Experiment $\mathsf{Hyb}_4(\lambda)$. This experiment is identical to experiment $\mathsf{Hyb}_3(\lambda)$, except that when the adversary submits the command $\mathsf{Lookup}(k)$ with key k, we run $\mathsf{Lookup}(\perp)$ instead of $\mathsf{Lookup}(k)$.

Recall that the output of the procedure is determined by the functionality and not the algorithm. In the following claim we show that the access pattern observed by the adversary in this experiment is statistically close to the one observed in $\mathsf{Hyb}_3(\lambda)$.

Claim C.6. *For any (unbounded) adversary $\mathcal{A}$, there is a negligible function $\mathsf{negl}(\cdot)$ such that*

$$|\Pr[\mathsf{Hyb}_3(\lambda) = 1] - \Pr[\mathsf{Hyb}_4(\lambda) = 1]| \leq n \cdot e^{-\Omega(\log^5 \lambda)}.$$

Proof. Consider a sequence of operations that the adversary makes. Let us denote by $\mathbf{I} = \{k_1, k_2, \ldots, k_n : k_1 < k_2 < \ldots < k_n\}$ the set of elements with which it invokes Build, by π the secret input permutation such that the i-th element $\mathbf{I}[i] = k_{\pi(i)}$, and by $Q = \{k^*_1, \ldots, k^*_m\}$ the set of keys with which it invokes Lookup. We first claim that it suffices to consider only the joint distribution of the access pattern of Step 3 in $\mathsf{Build}(\mathbf{I})$, followed by the access pattern of $\mathsf{Lookup}(k^*_i)$ for all $k^*_i \in \mathbf{I}$. In particular, in both hybrids, the outputs are determined by the functionality, and the access pattern of $\mathsf{Extract}()$ is identically distributed. Moreover, the access pattern in Steps 4 through 6 in Build is deterministic and is a function of the access pattern of Step 3. In addition, in both executions $\mathsf{Lookup}(k)$ for keys such that $k \notin \mathbf{I}$, as well as $\mathsf{Lookup}(\perp)$ cause a linear scan of the the overflow pile followed by an independent visit of a random bin (even when conditioning on the access pattern of Build) – we can ignore such queries. Finally, even though the adversary is adaptive, we essentially prove in the following that the entire view of the adversary is close in both experiment, and therefore the ordering of how the view is obtained cannot help distinguishing.

Let $X \leftarrow \mathsf{BallsIntoBins}(n, B)$ denote a sample of the access pattern obtained by throwing n balls into B bins. It is convenient to view X as a bipartite graph $X = (V_n \cup V_B, E_X)$, where V_n are the

n vertices representing the balls, V_B are B vertices representing the bins, and E_X are representing the access pattern. Note that the output degree of each balls is 1, whereas the degree of each bin is its load, and the expectation of the latter is n/B. For two graphs that share the same bins V_B, $X = (V_{n_1} \cup V_B, E_X)$ and $Y = (V_{n_2} \cup V_B, E_Y)$, we define the union of the two graphs, denoted $X \cup Y$, by $X \cup Y = (V_{n_1} \cup V_{n_2} \cup V_B, E_X \cup E_Y)$. Consider the following two distributions.

Distribution $\mathsf{AccessPtrn}_3(\lambda)$: In $\mathsf{Hyb}_3(\lambda)$, the joint distribution of the access pattern of Step 3 in $\mathsf{Build}(\mathbf{I})$, followed by the access pattern of $\mathsf{Lookup}(k_i)$ for all $k_i \in \mathbf{I}$, can be described by the following process:

1. Sample $X \leftarrow \mathsf{BallsIntoBins}(n, B)$. Let $(n_1, \ldots, n_B)$ be the loads obtained in the process and $\mu = \frac{n}{B}$ be the expectation of n_i for all $i \in [B]$.
2. Sample independent bin loads $(L_1, \ldots, L_B) \leftarrow \mathcal{F}_{n',B}^{\mathsf{throw\text{-}balls}}$, where $n' = n \cdot (1 - \epsilon)$. Let $\mu' = \frac{n'}{B}$ be the expectation of L_i for all $i \in [B]$.
3. Overflow: If for some $i \in [B]$ we have that $|n_i - \mu| > 0.5\epsilon\mu$ or $|L_i - \mu'| > 0.5\epsilon\mu$, then abort the process.
4. Consider the graph $X = (V_n \cup V_B, E_X)$, and for every bin $i \in [B]$, remove from E_X exactly $n_i - L_i$ edges arbitrarily (these correspond to the elements that are stored in the overflow pile). Let $X' = (V_n \cup V_B, E'_X)$ be the resulting graph. Note that X' has n' edges, each bin $i \in [B]$ has exactly L_i edges, and $n - n'$ vertices in V_n have no output edges.
5. Recall that π is the input permutation on $\mathbf{I}$. Let $\tilde{E}'_X = \{(\pi(i), v_i) : (i, v_i) \in E'_X\}$ be the set of permuted edges, $\tilde{V}_{n'} \subset V_n$ be the set of nodes that have an edge in $\tilde{E}'_X$, and $\tilde{X}' = (\tilde{V}_{n'} \cup V_B, \tilde{E}'_X)$. Note that there are n' vertices in $\tilde{V}_{n'}$.
6. For the ϵn remaining vertices in V_n but not in $\tilde{V}_{n'}$ that have no output edges (i.e., the balls in the overflow pile), sample new and independent output edges, where each edge is obtained by choosing independent bin $i \leftarrow [B]$. Let Z' be the resulting graph (corresponds to the access pattern of $\mathsf{Lookup}(k_i)$ for all k_i that appear in OF and not in the major bins). Let $Y = \tilde{X}' \cup Z'$. (The graph Y contains edges that correspond to the "real" elements placed in the major bins which were obtained from the graph $\tilde{X}'$, together with fresh "noisy" edges corresponding to the elements stored in the overflow pile).
7. Output (X, Y).

Distribution $\mathsf{AccessPtrn}_4(\lambda)$: In $\mathsf{Hyb}_4(\lambda)$, the joint distribution of the access pattern of Step 3 in $\mathsf{Build}(\mathbf{I})$, followed by the access pattern of $\mathsf{Lookup}(\perp)$ for all $k_i \in \mathbf{I}$, is described by the following (simpler) process:

1. Sample $X \leftarrow \mathsf{BallsIntoBins}(n, B)$. Let $(n_1, \ldots, n_B)$ be the loads obtained in the process and $\mu = \frac{n}{B}$ be the expectation of n_i for all $i \in [B]$.
2. Sample independent bin loads $(L_1, \ldots, L_B) \leftarrow \mathcal{F}_{n',B}^{\mathsf{throw\text{-}balls}}$, where $n' = n \cdot (1 - \epsilon)$. Let $\mu' = \frac{n'}{B}$ be the expectation of L_i for all $i \in [B]$.
3. Overflow: If for some $i \in [B]$ we have that $|n_i - \mu| > 0.5\epsilon\mu$ or $|L_i - \mu'| > 0.5\epsilon\mu$, then abort the process.
4. Sample an independent $Y \leftarrow \mathsf{BallsIntoBins}(n, B)$. (Corresponding to the access pattern of $\mathsf{Lookup}(\perp)$ for every command $\mathsf{Lookup}(k)$.)
5. Output (X, Y).

By the definition of our distributions and hybrid experiments, we need to show

$$|\Pr[\mathsf{Hyb}_3(\lambda) = 1] - \Pr[\mathsf{Hyb}_4(\lambda) = 1]| \leq \mathsf{SD}(\mathsf{AccessPtrn}_3(\lambda), \mathsf{AccessPtrn}_4(\lambda)) \leq n \cdot e^{-\Omega(\log^5 \lambda)}.$$

Towards this goal, first, by a Chernoff bound per bin and union bound over the bins, then by $\mu = \log^9 \lambda$ and $\epsilon = \frac{1}{\log^2 \lambda}$, it holds that

$$\Pr_{\mathsf{AccessPtrn}_3}[\mathsf{Overflow}] = \Pr_{\mathsf{AccessPtrn}_4}[\mathsf{Overflow}] \leq B \cdot 2\exp(-\mu(0.5\epsilon)^2/2) \leq n \cdot e^{-\Omega(\log^5 \lambda)}.$$

We condition on $\mathsf{Overflow}$ not occurring and show that both distributions output two independent graphs, i.e., two independent samples of $\mathsf{BallsIntoBins}(n, B)$, and thus they are equivalent.

This holds in $\mathsf{AccessPtrn}_4$ directly by definition. As for $\mathsf{AccessPtrn}_3$, consider the joint distribution of $(X, \tilde{X}')$ conditioning on $\mathsf{Overflow}$ not happening. For any graph $G = (V_{n'} \cup V_B, E_G)$ that corresponds to a sample of $\mathsf{BallsIntoBins}(n', B)$, we have that $\tilde{X}' = G$ if and only if (i) the loads of $\tilde{X}'$ equals to the loads of G and (ii) $\tilde{E}'_X = E_G$, where the loads of G are defined as the degrees of nodes $v \in V_B$. Observe that, by definition, the loads of $\tilde{X}'$ are exactly the loads of Z: $(L_1, \ldots, L_n)$, and hence the loads of $\tilde{X}'$ are independent of X. Also, since $\mathsf{Overflow}$ does not happen, X, and the event of (i), the probability of $\tilde{E}'_X = E_G$ is exactly the probability of the n' vertices in $\tilde{X}'$ matching those in G, which is $\frac{1}{n'!}$ by the uniform input permutation π. It follows that

$$\begin{aligned}\Pr[X' = G \mid X \wedge \neg\mathsf{Overflow}] &= \Pr[\text{loads of } G = (L_1, \ldots, L_n) \mid \neg\mathsf{Overflow}] \cdot \frac{1}{n'!} \\ &= \Pr[Z = G \mid \neg\mathsf{Overflow}]\end{aligned}$$

for all G, which implies that X' is independent of X. Moreover, in Step 6, we sample a new graph $Z' = \mathsf{BallsIntoBins}(n - n', B)$, and output Y as $\tilde{X}'$ augmented by Z'. In other words, we sample Y as follows: we sample two independent graphs $Z \leftarrow \mathsf{BallsIntoBins}(n', B)$ and $\tilde{Z}' \leftarrow \mathsf{BallsIntoBins}(n - n', B)$, and output the joint graph $Z \cup Z'$. This has exactly the same distribution as an independent instance of $\mathsf{BallsIntoBins}(n, B)$. We therefore conclude that

$$\mathsf{SD}\left(\mathsf{AccessPtrn}_3(\lambda) \mid \neg\mathsf{Overflow}, \mathsf{AccessPtrn}_4(\lambda) \mid \neg\mathsf{Overflow}\right) = 0.$$

Thus, following a fact on statistical distance,[17]

$$\begin{aligned}&\mathsf{SD}\left(\mathsf{AccessPtrn}_3(\lambda), \mathsf{AccessPtrn}_4(\lambda)\right) \\ &\leq \mathsf{SD}\left(\mathsf{AccessPtrn}_3(\lambda) \mid \neg\mathsf{Overflow}, \mathsf{AccessPtrn}_4(\lambda) \mid \neg\mathsf{Overflow}\right) + \Pr\left[\mathsf{Overflow}\right] \leq n \cdot e^{-\Omega(\log^5 \lambda)}.\end{aligned}$$

Namely, the access patterns are statistically close. The above analysis assumes that all n elements in the input $\mathbf{I}$ are real and the m Lookups visit all real keys in $\mathbf{I}$. If the number of real elements is less than n (or even less than n'), then the construction and the analysis go through similarly; the only difference is that the Lookups reveal a smaller number of edges in $\tilde{X}'$, and thus the distributions are still statistically close. The same argument follows if the m Lookups visit only a subset of real keys in $\mathbf{I}$. Also note that fixing any set $Q = \{k_1^*, \ldots, k_m^*\}$ of Lookup, every ordering of Q reveals the same access pattern $\tilde{X}'$ as $\tilde{X}'$ is determined only by $\mathbf{I}, \pi, X, Z$, and thus the view is identical for every ordering. This completes the proof of Claim C.6. □

Experiment Hyb_5. This experiment is the same as Hyb_4, except that we run Build in input $\mathbf{I}$ that consists of only dummy values.

Recall that in this hybrid experiment the output of $\mathsf{Extract}$ and Lookup is given to the adversary by the functionality, and not by the algorithm. Moreover, the access pattern of Build, due to the

[17] The fact is that for every two random variables X and Y over a finite domain, and any event E such that $\Pr_X[E] = \Pr_Y[E]$, it holds that $\mathsf{SD}(X, Y) \leq \mathsf{SD}(X \mid E, Y \mid E) + \Pr_X[\neg E]$. This fact can be verified by a direct expansion.

random function, each $\mathcal{O}(\mathsf{sk}||k_i)$ value is distributed uniformly at random, and therefore the random choices made to the real elements are similar to those made to dummy elements. We conclude that the view of the adversary in $\mathsf{Hyb}_4(\lambda)$ and $\mathsf{Hyb}_5(\lambda)$ is identical.

Claim C.7. $\Pr[\mathsf{Hyb}_4(\lambda) = 1] = \Pr[\mathsf{Hyb}_5(\lambda) = 1]$.

Experiment Hyb_6. This experiment is the same as Hyb_5, except that we replace the random oracle $\mathcal{O}(\mathsf{sk}\|\cdot)$ with a PRF key sk.

Observe that this experiment is identical to the ideal execution. Indeed, in the ideal execution the simulator runs the real Build operation on input that consists only of dummy elements and has an embedded PRF key. However, this PRF key is never used since we input only dummy elements, and thus the two experiments are identical.

Claim C.8. $\Pr[\mathsf{Hyb}_5(\lambda) = 1] = \Pr[\mathsf{Hyb}_6(\lambda) = 1]$.

By combining Claims C.3–C.8, we have that BigHT is $(1 - n^2 \cdot e^{-\Omega(\log\lambda \cdot \log\log\lambda)} - \delta^{\mathcal{A}}_{\mathsf{PRF}})$-oblivious, which concludes the proof of Theorem 7.2.

C.4 Proof of Security of SmallHT (Theorem 8.6)

We view our construction in a hybrid model, in which we have ideal implementations of the underlying building blocks: an oblivious random permutation (see Section 4.2 and Claim 8.2) and an oblivious Cuckoo assignment (see Section 4.5 and Claim 8.4). Since the two building blocks are ideal, we take into account the failure probability of the Cuckoo assignment, $e^{-\Omega(\log\lambda \cdot \log\log\lambda)}$, and the failure probability of the random permutation, $e^{-\Omega(\sqrt{n})} \leq e^{-\Omega(\log\lambda \log\log\lambda)}$ since $n = \log^9 \lambda$.

We present a simulator Sim that simulates $\mathsf{Build}, \mathsf{Lookup}$ and $\mathsf{Extract}$ procedures of $\mathsf{SmallHT}$.

- **Simulating Build.** Upon receiving an instruction to simulate Build with security parameter 1^λ and a list of size n, the simulator Sim runs the real $\mathsf{SmallHT.Build}$ algorithm on input 1^λ and a list that consists of n dummy elements. It outputs the access pattern of this algorithm. Let $(\mathbf{Y}, \mathsf{S}, \mathsf{sk})$ be the output state, where $\mathbf{Y}$ is an array of size $\mathsf{c}_{\mathsf{cuckoo}} \cdot n$, S is a stash of size $O(\log\lambda)$, and sk is a secret key used to generate pseudorandom values. The simulator stores this state.
- **Simulating Lookup.** When the adversary submits a Lookup command with a key k, the simulator Sim simulates an execution of the algorithm $\mathsf{SmallHT.Lookup}$ on input $\perp$ (i.e., a dummy element) with the state $(\mathbf{Y}, \mathsf{S}, \mathsf{sk})$ (which was generated while simulating the the Build operation).
- **Simulating Extract.** When the adversary submits an $\mathsf{Extract}$ command, the simulator Sim executes the real $\mathsf{SmallHT.Extract}$ algorithm with its stored internal state $(\mathbf{Y}, \mathsf{S}, \mathsf{sk})$.

We proceed to show that no adversary can distinguish between the real and ideal executions. Recall that in the ideal execution, with each command that the adversary outputs, it receives back the output of the functionality and the access pattern of the simulator, where the latter is simulating the access pattern of the execution of the command on dummy elements. On the other hand, in the real execution, the adversary sees the access pattern and the output of the algorithm that implements the functionality. The proof is via a sequence of hybrid experiments.

Experiment $\mathsf{Hyb}_0(\lambda)$. This is the real execution. With each command that the adversary submits to the experiment, the real algorithm is being executed, and the adversary receives the output of the execution together with the access pattern as determined by the execution of the algorithm.

Experiment $\mathsf{Hyb}_1(\lambda)$**.** This experiment is the same as Hyb_0, except that instead of choosing a PRF key sk, we use a truly random function $\mathcal{O}$. That is, instead of calling to $\mathsf{PRF}_{\mathsf{sk}}(\cdot)$ in Step 3 of Build and Step 4 of the function Lookup, we call $\mathcal{O}(\mathsf{sk}\|\cdot)$.

The following claim states that due to the security of the PRF, experiments Hyb_0 and Hyb_1 are computationally indistinguishable. The proof of this claim is standard.

Claim C.9. *For any probabilistic polynomial-time adversary $\mathcal{A}$, it holds that*

$$|\Pr[\mathsf{Hyb}_0(\lambda) = 1] - \Pr[\mathsf{Hyb}_1(\lambda) = 1]| \leq \delta^{\mathcal{A}}_{\mathsf{PRF}}(\lambda).$$

Experiment $\mathsf{Hyb}_2(\lambda)$**.** This experiment is the same as $\mathsf{Hyb}_1(\lambda)$, except that with each command that the adversary submits to the experiment, both the real algorithm is being executed as well as the functionality. The adversary receives the access pattern of the execution of the algorithm, yet the output comes from the functionality.

In the following claim, we show that the initial secret permutation and the random oracle, guarantee that experiments Hyb_1 and Hyb_2 are identical.

Claim C.10. $\Pr[\mathsf{Hyb}_1(\lambda) = 1] = \Pr[\mathsf{Hyb}_2(\lambda) = 1]$.

Proof. Recall that we assume that the lookup queries of the adversary are non-recurring. Our goal is to show that the output distribution of the extract procedure is a uniform permutation of the unvisited items even given the access patter of the previous Build and Lookup operations. By doing so, we can replace the Extract procedure with the ideal $\mathcal{F}^n_{\mathsf{HT}}$.Extract functionality which is exactly the difference between $\mathsf{Hyb}_1(\lambda)$ and $\mathsf{Hyb}_2(\lambda)$.

Consider a sequence of operations that the adversary makes. Let us denote by $\mathbf{I}$ the set of elements with which it invokes Build and by $k^*_1, \ldots, k^*_m$ the set of keys with which it invokes Lookup. Finally, it invokes Extract. We first argue that the output of $\mathcal{F}^n_{\mathsf{HT}}$.Extract consists of the same elements as that of Extract. Indeed, both $\mathcal{F}^n_{\mathsf{HT}}$.Lookup and SmallHT.Lookup remove every visited item so when we execute Extract, the same set of elements will be in the output.

We need to argue that the distribution of the permutation of unvisitied items in the output of Extract is uniformly random. This is enough since Extract performs IntersperseRD which shuffles the reals and dummies to obtain a uniformly random permutation overall (given that the reals were randomly shuffled to begin with). Fix an access pattern observed during the execution of Build and Lookup. We show, by programming the random oracle and the initial permutation appropriately (while not changing the access pattern), that the permutation is uniformly distributed.

Consider tuples of the form $(\pi_{\mathsf{in}}, \mathcal{O}, R, \mathsf{T}, \pi_{\mathsf{out}})$, where (1) π_{in} is the permutation performed on $\mathbf{I}$ by the input assumption (prior to Build), (2) $\mathcal{O}$ is the random oracle, (3) R is the internal randomness of all intermediate procedures (such as IntersperseRD, Algorithms 8.1 and 8.3, etc); (4) T is the access pattern of the entire sequence of commands $(\mathsf{Build}(\mathbf{I}), \mathsf{Lookup}(k^*_1), \ldots, \mathsf{Lookup}(k^*_m))$, and (5) π_{out} is the permutation on $\mathbf{I}' = \{(k, v) \in \mathbf{I} \mid k \notin \{k^*_1, \ldots, k^*_m\}\}$ which is the input to Extract. The algorithm defines a deterministic mapping $\psi_R(\pi_{\mathsf{in}}, \mathcal{O}) \to (\mathsf{T}, \pi_{\mathsf{out}})$.

To gain intuition, consider arbitrary R, π_{in}, and $\mathcal{O}$ such that $\psi_R(\pi_{\mathsf{in}}, \mathcal{O}) \to (\mathsf{T}, \pi_{\mathsf{out}})$ and two distinct existing keys k_i and k_j that are not queried during the Lookup stage (i.e., $k_i, k_j \notin \{k^*_1, \ldots, k^*_m\}$). We argue that from the point of view of the adversary, having seen the access pattern and all query results, he cannot distinguish whether $\pi_{\mathsf{out}}(i) < \pi_{\mathsf{out}}(j)$ or $\pi_{\mathsf{out}}(i) > \pi_{\mathsf{out}}(j)$. The argument will naturally generalize to arbitrary unqueried keys and an arbitrary ordering.

To this end, we show that there is π'_{in} and $\mathcal{O}'$ such that $\psi_R(\pi'_{\mathsf{in}}, \mathcal{O}') \to (\mathsf{T}, \pi'_{\mathsf{out}})$, where $\pi'_{\mathsf{out}}(\ell) = \pi_{\mathsf{out}}(\ell)$ for every $\ell \notin \{i, j\}$, and $\pi'_{\mathsf{out}}(i) = \pi_{\mathsf{out}}(j)$ and $\pi'_{\mathsf{out}}(j) = \pi_{\mathsf{out}}(i)$. The permutation π'_{in} is the same as π_{in} except that $\pi'_{\mathsf{in}}(i) = \pi_{\mathsf{in}}(j)$ and $\pi'_{\mathsf{in}}(j) = \pi_{\mathsf{in}}(i)$, and $\mathcal{O}'$ is the same as $\mathcal{O}$ except that

$\mathcal{O}'(k_i) = \mathcal{O}(k_j)$ and $\mathcal{O}'(k_j) = \mathcal{O}(k_i)$. The fact that the access pattern after this modification remains the same stems from the *indiscriminate* property of the hash table construction procedure which says that the Cuckoo hash assignments are a fixed function of the two choices of all elements (i.e., $\mathbf{MD_X}$), independently of their real key value (i.e., the procedure does not discriminate elements based on their keys in the input array). Note that the mapping is also reversible so by symmetry all permutations have the same number of configurations of π_{in} and $\mathcal{O}$.

For the general case, one can switch from any π_{out} to any (legal) π'_{out} by changing only π_{in} and $\mathcal{O}$ at locations that correspond to unvisited items. We define

$$\pi'_{\mathsf{in}}(i) = \pi_{\mathsf{in}}(\pi_{\mathsf{out}}{}^{-1}(\pi'_{\mathsf{out}}(i))) \quad \text{and} \quad \mathcal{O}'(k_i) = \mathcal{O}(k_{\pi_{\mathsf{in}}(\pi_{\mathsf{out}}{}^{-1}(\pi'_{\mathsf{out}}(i)))}).$$

Due to the indiscriminate property, this choice of π'_{in} and $\mathcal{O}'$ do not change the observed access pattern and result with the output permutation π'_{out}, as required. By symmetry, the resulting mapping between different $(\pi'_{\mathsf{in}}, \mathcal{O}')$ and π'_{out} is regular (i.e., each output permutation has the same number of ways to reach to) which completes the proof.

□

Experiment $\mathsf{Hyb}_3(\lambda)$. This experiment is the same as $\mathsf{Hyb}_2(\lambda)$, except that we modify the definition of $\mathsf{Extract}$ to output a list of n dummy elements. We also stop marking elements that were searched for during Lookup.

Recall that in this hybrid experiment the output of $\mathsf{Extract}$ is given to the adversary by the functionality, and not by the algorithm. Thus, the change we made does not affect the view of the adversary which means that experiments Hyb_2 and Hyb_3 are identical.

Claim C.11. $\Pr[\mathsf{Hyb}_2(\lambda) = 1] = \Pr[\mathsf{Hyb}_3(\lambda) = 1]$.

Experiment $\mathsf{Hyb}_4(\lambda)$. This experiment is identical to experiment $\mathsf{Hyb}_3(\lambda)$, except that when the adversary submits the command $\mathsf{Lookup}(k)$ with key k, we ignore k and run $\mathsf{Lookup}(\perp)$.

Recall that the output of the procedure is determined by the functionality and not the algorithm. By construction, the access pattern observed by the adversary in this experiment is identical to the one observed from $\mathsf{Hyb}_3(\lambda)$ (recall that we already switched the PRF to a completely random choices).

Claim C.12. $\Pr[\mathsf{Hyb}_3(\lambda) = 1] = \Pr[\mathsf{Hyb}_4(\lambda) = 1]$.

Experiment Hyb_5. This experiment is the same as Hyb_4, except that we run Build in input $\mathbf{I}$ that consists of only dummy values.

Recall that in this hybrid experiment the output of $\mathsf{Extract}$ and Lookup is given to the adversary by the functionality, and not by the algorithm. Moreover, the access pattern of Build, due to the random oracle and the obliviousness of all the underlying building blocks (oblivious Cuckoo hash, oblivious random permutation, oblivious tight compaction, $\mathsf{IntersperseRD}$, oblivious bin assignment, and oblivious sorting), the view of the adversary in $\mathsf{Hyb}_4(\lambda)$ and $\mathsf{Hyb}_5(\lambda)$ is identical.

Claim C.13. $\Pr[\mathsf{Hyb}_4(\lambda) = 1] = \Pr[\mathsf{Hyb}_5(\lambda) = 1]$.

Experiment Hyb_6. This experiment is the same as Hyb_5, except that we replace the random oracle $\mathcal{O}(\mathsf{sk}\|\cdot)$ with a PRF key sk.

Observe that this experiment is identical to the ideal execution. Indeed, in the ideal execution the simulator runs the real Build operation on input that consists only of dummy elements and has

an embedded PRF key. However, this PRF key is never used since we input only dummy elements, and thus the two experiments are identical.

Claim C.14. $\Pr[\mathsf{Hyb}_5(\lambda)=1]=\Pr[\mathsf{Hyb}_6(\lambda)=1]$.

By combining Claims C.9–C.14, SmallHT is $(1-e^{-\Omega(\log\lambda\cdot\log\log\lambda)}-\delta^{\mathcal{A}}_{\mathsf{PRF}})$-oblivious, and we conclude the proof of Theorem 8.6.

C.5 Proof of Security of CombHT (Theorem 8.8)

We present a sequence of hybrid constructions (where we ignore the failure probability of the primitives we use) and show that each one of them obliviously implements Functionality 4.7. Afterwards, we will account for the security loss of each modification and/or primitive we use.

- **Construction I:** This construction is the same as Construction 7.1, except that we replace each naïveHT with SmallHT. Let $\mathsf{S}_1,\ldots,\mathsf{S}_B$ denote the small stash of the bins $\mathsf{OBin}_1,\ldots,\mathsf{OBin}_B$, respectively.
- **Construction II:** This construction is the same as Construction I, except the following (inefficient) modification. Instead of searching for the key k_i in the small stash S_i of one of the bins of SmallHT, we search for k_i in all small stashes $\mathsf{S}_1,\ldots,\mathsf{S}_B$ in order.
- **Construction III:** This construction is the same as Construction II, except that we modify Build as follows. We merge all the small stashes $\mathsf{S}_1,\ldots,\mathsf{S}_B$ into one long list. As in Construction II, when we have to access one of the stashes and look for a key k_i, we perform a linear scan in this list, searching for k_i.
- **Construction IV:** This construction is the same as Construction III, except that we make the search in the merged set of stashes more efficient. In CombHT.Build we construct an oblivious Cuckoo hashing scheme as in Theorem 4.14 on the elements in the combined set of stashes. The resulting structure is called $\mathsf{CombS}=(\mathsf{CombS}_\mathsf{T},\mathsf{CombS}_\mathsf{S})$ and it is composed of a main table and a stash. Observe that this construction is identical to Construction 8.7.

Construction II is the same as Construction I, except that there is a blowup in the access pattern of each Lookup by performing a linear scan of all elements in all stashes. In terms of functionality, by construction the input/output behavior of the two constructions is exactly the same. For obliviousness, one can simulate the linear scan of all the stashes by performing a fake linear scan. More formally, there exists a 1-to-1 mapping between access pattern provided by Construction I and an access pattern provided by Construction II. Thus, there is no security loss from Construction I to Construction II.

Construction III and Construction II are the same, except for a cosmetic modification in the locations where we put the elements from the stashes. Thus, no security loss from Construction II to Construction III.

Construction IV is the same as Construction III, except that we apply an oblivious Cuckoo hash on the merged set of hashes $\cup_{i\in[B]}\mathsf{S}_i$ to improve on Lookup time (compared to a linear scan).

Lastly, we analyze the total security loss. We first implemented all the major bins with SmallHT. Since there are $n/\mu<n$ bins, by Theorem 8.6, this amounts to $n\cdot e^{-\Omega(\log\lambda\log\log\lambda)}+\delta^{\mathcal{A}}_{\mathsf{PRF}}$ total security loss. We lose an additional $e^{-\Omega(\log\lambda\cdot\log\log\lambda)}+\delta^{\mathcal{A}}_{\mathsf{PRF}}$ additive term by implementing the merged stashes using an oblivious cuckoo hash. Also, recall that the original security loss of BigHT was already $n^2\cdot e^{-\Omega(\log\lambda\cdot\log\log\lambda)}+\delta^{\mathcal{A}}_{\mathsf{PRF}}$. In conclusion, the final construction $\left(1-n^2\cdot e^{-\Omega(\log\lambda\cdot\log\log\lambda)}-\delta^{\mathcal{A}}_{\mathsf{PRF}}\right)$-obliviously implements Functionality 4.7.

C.6 Proof of Security of ORAM (Theorem 9.2)

Since the functionality of ORAM is deterministic, we prove correctness and obliviousness separately. Correctness is straightforward, and we focus on obliviousness. We show that the access pattern is simulatable for any sequence of operations $\mathsf{Access}(\mathsf{op}_1, \mathsf{addr}_1, \mathsf{data}_i), \ldots, \mathsf{Access}(\mathsf{op}_n, \mathsf{addr}_n, \mathsf{data}_n)$. That is, for any one of the following constructions, there exists a simulator such that for any sequence of operations the real construction on that sequence of operations has indistinguishable access pattern than a simulator that just receives $\mathsf{Access}(\bot, \bot, \bot), \ldots$.

Construction 1. Our starting point is a construction in the $(\mathcal{F}_{\mathsf{HT}}, \mathcal{F}_{\mathsf{Dict}}, \mathcal{F}_{\mathsf{Shuffle}}, \mathcal{F}_{\mathsf{compaction}})$-hybrid model which is slightly different from Construction 9.1. In this construction, each level $T_{\ell+1}, \ldots, T_L$ is implemented using the ideal functionality $\mathcal{F}_{\mathsf{HT}}$ (of the respective size). The dictionary D is implemented using the ideal functionality $\mathcal{F}_{\mathsf{Dict}}$. Steps 6c and 6(c)ii are implemented using $\mathcal{F}_{\mathsf{Shuffle}}$ of the respective size, and the compaction in Step 6(c)i is implemented using $\mathcal{F}_{\mathsf{compaction}}$. Note that in this construction, Step 6(d)ii is invalid, as the $\mathcal{F}_{\mathsf{HT}}$ functionality is not necessarily implemented using stashes. This construction boils down to the construction of Goldreich Ostrovsky using ideal implementations of $(\mathcal{F}_{\mathsf{HT}}, \mathcal{F}_{\mathsf{Dict}}, \mathcal{F}_{\mathsf{Shuffle}}, \mathcal{F}_{\mathsf{compaction}})$. For completeness, we provide a full description:

The construction: Let $\ell = 11 \log\log\lambda$ and $L = \log N$. The internal state include an handle D to $\mathcal{F}_{\mathsf{Dict}}^{2^\ell}$, handles $T_{\ell+1}, \ldots, T_L$ to $\mathcal{F}_{\mathsf{HT}}{}^{2^{\ell+1},N}, \ldots, \mathcal{F}_{\mathsf{HT}}{}^{2^L,N}$, respectively, a counter ctr and flags $\mathsf{full}_{\ell+1}, \ldots, \mathsf{full}_L$. Upon receiving a command $\mathsf{Access}(\mathsf{op}, \mathsf{addr}, \mathsf{data})$:

1. Initialize $\mathsf{found} := \mathsf{false}$, $\mathsf{data}^* := \bot$.
2. Perform $\mathsf{fetched} := D.\mathsf{Lookup}(\mathsf{addr})$. If $\mathsf{fetched} \neq \bot$ then set $\mathsf{found} := \mathsf{true}$.
3. For each $i \in \{\ell+1, \ldots, L\}$ in increasing order, do:
   (a) If $\mathsf{found} = \mathsf{false}$, run $\mathsf{fetched} := T_i.\mathsf{Lookup}(\mathsf{addr})$. If $\mathsf{fetched} \neq \bot$, let $\mathsf{found} = \mathsf{true}$ and $\mathsf{data}^* := \mathsf{fetched}$.
   (b) Else, $T_i.\mathsf{Lookup}(\bot)$.
4. If $\mathsf{found} = \mathsf{false}$, i.e., this is the first time addr is being accessed, set $\mathsf{data}^* = 0$.
5. Let $(k, v) := \{(\mathsf{addr}, \mathsf{data}^*)\}$ if $\mathsf{op} = \mathsf{read}$ operation; else let $(k, v) := \{(\mathsf{addr}, \mathsf{data})\}$. Insert $(k, (\ell, \bot, v))$ into oblivious dictionary D using $D.\mathsf{Insert}(k, v)$.
6. Increment ctr by 1. If $\mathsf{ctr} \equiv 0 \mod 2^\ell$, perform the following.
   (a) Let j be the smallest level index such that $\mathsf{full}_j = 0$ (i.e., empty). If all levels are marked full, then $j := L$. In other words, j is the target level to be rebuilt.
   (b) Let $\mathbf{U} := D.\mathsf{Extract}() \| T_{\ell+1}.\mathsf{Extract}() \| \ldots \| T_{j-1}.\mathsf{Extract}()$ and set $j^* := j - 1$. If all levels are marked full, then additionally let $\mathbf{U} := \mathbf{U} \| T_L.\mathsf{Extract}()$ and set $j^* := L$.
   (c) Run $\mathcal{F}_{\mathsf{Shuffle}}(\mathbf{U})$. Denote the output by $\tilde{\mathbf{U}}$. If $j = L$, then additionally do the following to shrink $\tilde{\mathbf{U}}$ to size $N = 2^L$:
      i. Run $\mathcal{F}_{\mathsf{compaction}}(\tilde{\mathbf{U}})$ moving all real elements to the front. Truncate $\tilde{\mathbf{U}}$ to length N.
      ii. Run $\tilde{\mathbf{U}} \leftarrow \mathcal{F}_{\mathsf{Shuffle}}^N(\tilde{\mathbf{U}})$
   (d) Rebuild the jth hash table with the 2^j elements from $\tilde{\mathbf{U}}$ by calling $\mathcal{F}_{\mathsf{HT}}.\mathsf{Build}(\tilde{\mathbf{U}})$. Mark $\mathsf{full}_j := 1$.
   (e) For $i \in \{l, \ldots, j-1\}$, reset T_i to empty structure and set $\mathsf{full}_i := 0$.
7. Output data^*.

Claim C.15. *Construction 1 is perfectly oblivious.*

Proof. The simulator Sim runs the algorithm Access on dummy values. In more detail, it maintains an internal secret state that consists of handles to ideal implementations of the dictionary D, the hash tables $T_{\ell+1}, \ldots, T_L$, bits $\mathsf{full}_{\ell+1}, \ldots, \mathsf{full}_L$ and counter ctr exactly as the real construction. Upon receiving a command $\mathsf{Access}(\perp, \perp, \perp)$, the simulator runs Construction 1 on input $(\perp, \perp, \perp)$.

By definition of the algorithm, the access pattern (in particular, which ideal functionalities are being invoked with each Access) is completely determined by the internal state $\mathsf{ctr}, \mathsf{full}_{\ell+1}, \ldots, \mathsf{full}_L$. Moreover, the change of these counters is deterministic and is the same in both real and ideal executions. As a result, the real algorithm and the simulator perform the exact same calls to the internal ideal functionalities with each Access. In particular, it is important to note that Lookup is invoked on all levels regardless of which level the element was found, and the level that is being rebuild is completely determined by the value of ctr. Moreover, the construction preserves the restriction of the functionality $\mathcal{F}_{\mathsf{HT}}$ in which any key is being searched for only once between two calls to Build. □

Give that Construction 1 obliviously implement Functionality 3.4, we proceed with a sequence of constructions and show that each and one of them is simulatable. Note that throughout the way, we also look at constructions that are not necessarily correct. All we care is that the access pattern is simulatable.

- **Construction 2.** This is the same as in Construction 1, where we instantiate $\mathcal{F}_{\mathsf{Shuffle}}$ and $\mathcal{F}_{\mathsf{compaction}}$ with the real implementations. Explicitly, we instantiate $\mathcal{F}_{\mathsf{Shuffle}}$ in Step 6c with $\mathsf{Intersperse}^{(j^*-\ell)}$ (Algorithm 6.4), instantiate $\mathcal{F}_{\mathsf{Shuffle}}$ in Step 6(c)ii with $\mathsf{IntersperseRD}$ (Algorithm 6.6, and instantiate $\mathcal{F}_{\mathsf{compaction}}$ in Step 6(c)i with an algorithm for tight compaction (Theorem 5.1). Note that at this point, the hash tables $T_{\ell+1}, \ldots, T_L$ are still implemented using the ideal functionality $\mathcal{F}_{\mathsf{HT}}$, as well as D that uses $\mathcal{F}_{\mathsf{Dict}}$.
- **Construction 3.** This construction is the same as Construction 2, with the difference that in Step 3 we always lookup for dummy elements in all levels, regardless of the value of found.
- **Construction 4.** In this construction, we follow Construction 3 but instantiate $\mathcal{F}_{\mathsf{HT}}$ with Construction 8.7 (i.e., CombHT from Theorem 8.8). Note that we do not combine the stashes yet. That is, we simply replace Step 6d (as Build), Step 3 (as Lookup) and Step 6b (as $\mathsf{Extract}()$) in Construction 1 with the implementation of Construction 8.7 instead of the ideal functionality $\mathcal{F}_{\mathsf{HT}}$.
- **Construction 5.** In this construction, we follow Construction 4 but change Step 6d as in Construction 9.1: We add all elements in OF_{S} and $\mathsf{CombS}_{\mathsf{S}}$ into D.
- **Construction 6.** In this construction, we follow Construction 5, but make the following change: In Step 3, we modify the Lookup procedure of Construction 8.7, and do not access OF_{S} and $\mathsf{CombS}_{\mathsf{S}}$.
- **Construction 7.** This is the same as Construction 6, where we replace the ideal implementation $\mathcal{F}_{\mathsf{Dict}}$ of the dictionary D with the real perfect oblivious dictionary (Corollary 4.17).
- **Construction 8.** This is the same as Construction 7, but here we perform Lookup in each level to addr or $\perp$ in each level according to the value of found. Note that this is exactly Construction 9.1.

The theorem is obtained using a sequence of simple claims, given that Construction 1 is simulatable. Specifically, Construction 2 is simulatable using the composition theorem. Construction 3 is simulatable using the same simulator as Construction 2, as the simulator always perform dummy lookup on each level. Construction 4 is simulatable using again the composition theorem. In Construction 5, we modify the simulator to do the same, and pretend to move the elements from the

stashes into the dictionary. This is a deterministic and well-defined change to the access pattern from Construction 4, and therefore Construction 5 is simulatable. From Construction 5 to 6 we follow a similar argument, and in Construction 7 we use again the composition theorem. Finally, for Construction 8, we again use the argument that the simulator is oblivious to the value of found.