OptORAMa: Optimal Oblivious RAM*

Gilad Asharov† Ilan Komargodski‡ Wei-Kai Lin§ Kartik Nayak¶

Enoch Peserico‖ Elaine Shi**

December 11, 2018

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.

*This paper incorporates the manuscript of Peserico [33].

†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

Oblivious RAM (ORAM), first introduced and constructed by Goldreich and Ostrovsky [18] is a method to compile programs in such a way that the compiled program preserve the input-output functionality of the original program, but the memory access pattern is independent of the data that is being manipulated by the program. This allows a client to outsource an encrypted database to an untrusted server and to later perform queries on it without leaking information about the content of the database. In this work, we focus on the classical setting of logarithmic-size memory word and a constant number of CPU registers.

We are interested in *efficient* compilers where the complexity of the compiled RAM machine is as close as possible to the complexity of the original RAM machine. We focus on the total computation overhead, namely the multiplicative increase, comparing the ORAM and the original RAM. We call this term *time overhead* or *time blowup*.

In their original work, Goldreich and Ostrovsky [18] presented the *hierarchical ORAM* construction that allows us to compile any program into an oblivious one with amortized poly-logarithmic time overhead, assuming one-way functions. That is, for any sequence of operations on a logical memory of size N, the compiler of Goldreich and Ostrovsky [18] performs them obliviously with $O(\log^3 N)$ amortized time overhead.

By extending and optimizing the building blocks used in the hierarchical ORAM constructions, improved constructions were given in subsequent works [1, 11, 20, 34]. Until very recently, the best result was given by Kushilevitz, Lu, and Ostrovsky [25] who presented an ORAM with $O(\log^2 N/\log\log N)$ amortized time blowup. See Chan et al. [8] for a unified framework for hierarchical ORAM constructions that captures the main differences between all of the above schemes.

A different approach for constructing ORAM schemes, called *tree-based ORAM*, was initiated by Shi et al. [36] who achieved $O(\log^3 N)$ time blowup (but with super constant client memory). This approach was optimized (see e.g., Gentry et al. [15]) until the Path ORAM construction of Stefanov et al. [38] that had $O(\log^2 N)$ time blowup. The Circuit ORAM of Wang, Chan, and Shi [39] has the same time blowup but assuming constant client memory. A feature of this construction is that if large blocks are available (e.g., of size $\Omega(\log^2 N)$), then the overhead becomes $O(\log N)$, the best possible (see below). However, assuming large blocks is somewhat non-standard and we avoid doing so in this work.

It is known that $\Omega(\log N)$ amortized time blowup is required by any ORAM. This was proven in the work of Goldreich and Ostrovsky [18] for statistically secure ORAM scheme in the so called *balls and bins* model [5].[1] More recently, Larsen and Nielsen [26] showed that the same lower bound holds even for computationally secure scheme and with no assumption on the model.[2] Bridging the gap between the logarithmic time overhead lower bound and the poly-logarithmic time overhead upper bound was a major open problem in the past three decades.

For years, the two known techniques for ORAM fell short of bridging this gap and it was unclear whether the lower bound could be improved or there are better techniques for ORAM. In a beautiful recent work, Patel, Persiano, Raykova, and Yeo [32] presented an ORAM scheme, in the hierarchical ORAM framework, with *quasi-logarithmic* overhead, assuming one-way functions. Namely, their scheme has a $O(\log N \cdot \log\log N)$ amortized time blowup and it is computationally secure. Still, there is no construction that matches the known lower bound.

[1] In this model, the n data items are modeled as "balls", CPU registers and server-side data storage locations are modeled as "bins", and the set of allowed data operations consists only of moving balls between bins.

[2] An even more recent work of Jacob, Larsen, and Nielsen [22] shows that concrete data structures, such as stacks, queues, deques, priority queues and search trees, when implemented obliviously require $\Omega(\log N)$ overhead.

1.1 Our Results

We provide the first *optimal* ORAM construction, having only $O(\log N)$ amortized time blowup and matching the lower bounds of Goldreich and Ostrovsky [18] and Larsen and Nielsen [26]. Our construction assumes the existence of one-way functions. Moreover, our ORAM (similarly to the one of [32]) could be instantiated to obtain statistical security in the *balls and bins* model if the client is provided with access to a random oracle, which matches the assumptions in the original lower bound of Goldreich and Ostrovsky [18].

Theorem 1.1. *There exists an ORAM (in the* balls and bins *model) with $O(\log N)$ amortized time blowup, where N is the size of the logical memory. The construction assumes one-way functions, and a constant client's memory size.*

Our construction is inspired by and non-trivially improves on the recent beautiful work of Patel et al. [32].

The lower bound of Larsen and Nielsen [26] also relates to the case where the client's internal memory is not constant size. If the client's memory size is m, then the lower bound implies that $\Omega(\log(N/m))$ amortized overhead is inherent. Our construction extends to this scenario, obtaining the following corollary.

Corollary 1.2. *There exists an ORAM with $O(\log(N/m))$ amortized time blowup, where N is the size of the logical memory and m is the client's memory size. The construction assumes one-way functions.*

Tight compaction. Our construction of Theorem 1.1 relies on many building blocks, some of which are new and some of which are known from the literature. One of the building blocks that we significantly improve on is oblivious tight compaction. In this problem, we are given an array of elements where some elements are marked, and our goal is to permute the elements in the array so that all marked elements end up in the front of the array. We give an asymptotically optimal oblivious algorithm that achieves this task.

Theorem 1.3. *There exists a deterministic oblivious tight compaction algorithm that takes $O(n)$ time to compact any input array of length n.*

The best previously known deterministic oblivious tight compaction algorithm was implied by sorting networks (Ajtai et al. [2] and Goodrich [19]) and thus has $O(n \cdot \log n)$ running time. The best previously known randomized tight compaction algorithm has running time $O(n \cdot \log\log n)$ and negligible probability of error in n (Lin et al. [27] and Mitchell and Zimmerman [29]). Our algorithm does not result with a stable permutation (i.e., the marked and unmarked elements do not preserve their relative internal ordering) and this is inherent for every oblivious compaction algorithm in the balls and bins model (Lin et al. [27]) that runs in time $o(n \log n)$.

2 Technical Overview of Our Construction

We start with an overview of the hierarchical ORAM framework initially proposed by Goldreich and Ostrovsky [16, 18] and improved in subsequent works [8, 20, 25]. We then describe the elegant ideas in the very recent work by Patel et al. [32] in which they further improved the asymptotical performance of hierarchical ORAMs, achieving *quasi*-logarithmic blowup. Finally, we explain the obstacles for getting rid of the extra $O(\log\log N)$ factor that arises in their construction, and describe our set of algorithmic ideas for accomplishing this.

We note that this section contains a very high-level description of the different building blocks and steps in our construction and additional more elaborate explanations are given later in the corresponding sections.

2.1 Overview of Previous Constructions and Challenges

Our starting point is the hierarchical ORAM. 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 $\{T_j\}_{j \leq \ell}$ 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 every 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 [8,20]), building a level with 2^k items obliviously requires $O(2^k \cdot \log(2^k)) = O(2^k \cdot k)$ time blowup. This building algorithm is based on oblivious 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 [2, 19]). Thus, summing over all levels (and ignoring the $\log N$ lookup operations across different levels with each access), t accesses require $\sum_{i=1}^{\log N} \left\lceil \frac{t}{2^i} \right\rceil \cdot O(2^i \cdot i) = O(t \cdot \log^2 N)$ time blowup. On the other hand, lookup is essentially constant time per level (ignoring searching in stashes which introduce additive factor), and the time of lookup is $O(\log N)$. Thus, there is an asymmetry between build time and lookup time, and the main overhead is the build.

The work of Patel et al. [32]. In earlier constructions, oblivious hash tables [8, 16, 18, 20, 25] could be built from *every* input array. However, obliviously building such a hash table requires expensive oblivious sorting, causing the extra logarithmic factor. The key idea of Patel et al. [32] 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[3]; further, 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

[3] 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.

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 all 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 technique, called *multi-array shuffle*, that combines these arrays to a shuffled array within $O(n \cdot \log\log\lambda)$ time, where $n = |T_0| + |T_1| + \ldots + |T_{\ell-1}|$.[4] The algorithm is rather involved, and has a negligible probability of failure. We elaborate on this procedure below.

Our construction. Our construction builds upon and simplifies the construction of Patel et al. We improve the construction of Patel et al. in two different aspects:

1. We show how to implement a multi-array shuffle in $O(n)$ time. Our algorithm has perfect security and perfect correctness.
2. We develop a hash table that supports build in $O(n)$ time assuming that the input array is randomly shuffled.

In the following, we describe the core ideas behind these improvements, where in Section 2.2 we present our multi-array shuffle algorithm. In Section 2.3 we show how to build a hash table in $O(n)$ time. In Section 2.4 we describe our tight compaction algorithm that plays a pivotal role in our construction.

2.2 Multi-Array Shuffle in Linear Time: Intersperse

The multi-array shuffle of Patel et al. takes as input k arrays, $\mathbf{I}_1, \ldots, \mathbf{I}_k$ of size $n_1, \ldots, n_k$, respectively. Each $\mathbf{I}_i$ is assumed to already be randomly shuffled. The output of the procedure is a single array that contains the all the elements from $\mathbf{I}_1, \ldots, \mathbf{I}_k$ in a randomly shuffled order. Ignoring obliviousness for now, we could first initialize an output array of size $n = n_1 + \ldots + n_k$. Then, we assign for $\mathbf{I}_1$ exactly n_1 random locations in the output array, and place the elements from $\mathbf{I}_1$ arbitrarily in these locations. Then, for $\mathbf{I}_2$ we assign exactly n_2 random locations in the remaining $n - n_1$ open cells in the output array, and place the elements from $\mathbf{I}_2$ in these cells. We continue in a similar manner until all input arrays are placed in the output array.[5] The challenge is how to perform this placement obliviously, without revealing the mapping from the input array to the output array. Patel et al. shows a rather involved technique for achieving this placement obliviously in total time $O(n \log\log\lambda)$ with negligible probability of failure.

We simplify this construction. For simplicity of exposition, we consider the case of randomly shuffling only two input arrays $\mathbf{I}_0$ and $\mathbf{I}_1$, and call this procedure "intersperse". We choose at random a bit vector Aux in which exactly $|\mathbf{I}_0|$ locations are 0, and $|\mathbf{I}_1|$ locations are 1. Our goal is to (obliviously!) place in each location i in the output array an element from $\mathbf{I}_{\mathsf{Aux}[i]}$. Towards this end, we observe that this problem is exactly the reverse problem of oblivious tight compaction: given an input array of size n containing keys that are 1-bit, we want to sort the array such that all elements with key 0 will appear before all elements with key 1. We devise an oblivious tight compaction algorithm for arrays of size n in $O(n)$ time. We refer to Section 2.4 for an overview of

[4] 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 an hierarchical ORAM construction).

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

this construction, and to Section 5 for the detailed construction. We show how to use this algorithm "in reverse" to intersperse two arrays. The reader is referred to Section 6 for further details.

2.3 Hash Table Build in Linear Time

In previous works, hash tables were implemented as oblivious Cuckoo hash tables, where build requires $O(n \cdot \log n)$ time and relies on oblivious sorts. The hash table of Patel et al. requires $O(n \cdot \log\log\lambda)$ time and works as long as the input array is a-priori shuffled. 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 $\mathsf{poly}\log\log\lambda$ factors in both the build and lookup phases.

The following warmup construction already deviates from the one by Patel et al. In particular, 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, conceptually, we rely only on a single layer of such a hash table construction; thus, lookup accesses only a single bin. While the warmup construction requires $\mathsf{poly}\log\log\lambda$ time to perform the single access within that bin, this simplification, however, allows us to later show a construction that performs this in-bin lookup in O(1) time.

Warmup construction: Oblivious hash table with $\mathsf{poly}\log\log\boldsymbol{\lambda}$ 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 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 place the n balls into the B bins, and get loads $n_1, \ldots, n_B$ which are revealed to the adversary. Then, we sample new secret loads $L_1, \ldots, L_B$ corresponding to an independent process of throwing n' balls into B bins. By tuning n', we can guarantee that $L_i < n_i$ holds for every $i = 1, \ldots, B$ with overwhelming probability. We extract from each bin arbitrary $n_i - L_i$ elements obliviously and move them to an overflow pile, while padding the sizes of all bins to the same bound. The crux of the security proof is that the secret loads $L_1, \ldots, L_B$ are never revealed and thus the access pattern in the lookup phase is independent to that of the build phase, and therefore is simulatable.

There are many techincal details needed to implement the above approach. First, how are the bin sizes truncated to the secret loads $L_1, \ldots, L_B$. Second, how are the elements in the overflow pile stored and accessed. Finally, how each bin is being implemented.

For the first question, we leverage the linear time tight compaction algorithm (see Section 2.4) to extract the number of elements we want from each bin. For the overflow pile, which turns out to be of size $m = n/\log^2\lambda$, 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 (plus some stash of size

$O(\log \lambda)$, which we ignore for now). To support lookups for bins of size $O(\mathsf{poly}\log \lambda)$, we use 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 [9, 11].

Oblivious hash table with linear build time and constant lookup time. In the warmup construction, ignoring the lookup time in the cuckoo hash table, the only super-linear operation that we have is the use of a perfectly secure ORAM, which we employ for bins of size $O(\mathsf{poly}\log \lambda)$. We now show how to replace this with a data structure with linear time build and constant time lookup.

Oblivious sorts and oblivious random permutations are frequently used in ORAM constructions. We observe that if a memory word can hold B elements, then one can run oblivious sorts and oblivious random permutations in $O(\frac{n}{B} \cdot \log^2 n)$ time. Moreover, since our bin sizes are $O(\mathsf{poly}\log \lambda)$, the metadata to represent an element can be represented in $O(\log\log \lambda)$ bits. We assume that the word size is $\Omega(\log N)$, and therefore each word can hold up to $B = \Omega(\log N / \log\log \lambda)$ elements' metadata (but not the elements themselves). Thus, for $n = O(\mathsf{poly}\log \lambda)$, we can run oblivious sorts and oblivious random permutations on metadata in linear time in n. Namely, we can compute for each element its final destination, but not actually move it.

We show that this weaker primitive is sufficient. We carefully separate between working on metadata and working on the elements themselves, and show how to compute a Cuckoo hash table for the elements while first considering only the metadata and not the actual elements. We refer the reader to Section 8 for the actual construction.

Additional techniques to achieve the final ORAM construction. Our final ORAM construction uses the linear time intersperse procedure and linear time oblivious hash table described above. There are still two concerns that are not addressed. First, 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. Second, 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. [25] 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.

2.4 Oblivious Tight Compaction

Given an input array $\mathbf{I}$ of n elements in which some are marked 0 and the rest are marked 1, we want to move all elements marked 0 to the beginning of the array. We show a deterministic algorithm for this task that runs in time $O(n)$. We first show how to reduce this problem (in linear time) to the relaxed problem of loose compaction: Given an array $\mathbf{I}$ with n elements in which at most n/ℓ elements are real (for some constant $\ell > 2$), return an array of size $n/2$ that contain all the real elements. We then show how to implement loose compaction in linear time.

Reducing tight compaction to loose compaction. Given an input array $\mathbf{I}$ of n elements in which some are marked 0 and the rest are marked 1, we first count the number of total elements marked 0 in the input array, and let c be this number. The first observation is that all 0-elements in the input array that reside in locations $1, \ldots, c$, and all 1-elements in locations $c+1, \ldots, n$ are already placed correctly. Thus, we just need to handle the rest of the elements which we call the *misplaced* ones. The number of misplaced elements marked 0 equals to the number of misplaced elements marked 1, and all we have to do is to swap between each misplaced 0-element with some misplaced 1-element.

Our main idea to perform these swaps obliviously is to perform swaps along the edges of a bipartite expander graph. That is, consider a bipartite expander graph where the left nodes are associated with the elements. The edges of the graph are the access pattern of our algorithm. We will swap two misplaced elements that have a different mark if they have a common neighbor on the right. To make sure this algorithm runs in linear time, the graph has to be d-regular for $d = O(1)$ and that the list of neighbors of every node can be computed using $O(1)$ basic operations. Such explicit expander graphs are known to exist (for example, Margulis [28]).

Using the expansion properties of the graph, we can bound the number of misplaced elements that were not swapped by this process: at most n/ℓ for some $\ell > 2$. Thus, we can invoke loose compaction where we consider the remaining misplaced element as the real ones and the rest as dummies. By this, we reduced the problem from n elements to $n/2$ and we proceed in recursion to swap the misplaced elements on that first half of the array, until all 0-elements and 1-elements are swapped.

Loose compaction. The main idea of the procedure is to first distribute the real balls to many bins, while ensuring that no bin consists of too many real balls. Particularly, we will show, for some $B = O(1)$, a method to distribute all real balls to $2n/B$ bins such that each bin will have at most $B/4$ real balls (and the rest are dummies). After performing this balanced distribution, all bins have small load, and we can merge any 4 bins together to one bin. Thus, the $2n/B$ bins results in $n/(2B)$ bins of size B each, i.e., an array of size $n/2$. It remains to explain how we obtain the balanced distribution of real balls.

Interpret the input array $\mathbf{I}$ as n/B bins of B elements each, simply by considering the i-th bin as the elements $\mathbf{I}[i \cdot B + 1], \ldots, \mathbf{I}[(i + 1) \cdot B]$, for $i = 0, \ldots, n/B - 1$. Additionally, allocate an array $\mathbf{I}'$ of n/B bins, each consists of B dummy elements. Each bin that contains at least $B/4$ real elements we mark as "dense", and all other bins we mark as "sparse". Consider a bipartite expander graph $G = (L, R, E)$, where the left n/B nodes correspond to the bins of $\mathbf{I}$ and the right n/B nodes to the bins of $\mathbf{I}'$, and let $S \subset L$ be the set of dense bins in $\mathbf{I}$. The balanced distribution of elements will be defined using the edges of the bipartite expander and more specifically using a $(B, B/4)$-matching in G for the set S: such a matching is a set of edges $M \subseteq E$, such that for every $u \in S$ there are at least B outgoing edges, and for every $v \in R$ there are at most $B/4$ incoming edges. Using this matching M for the set of dense bins S, we distribute these bins in $\mathbf{I}$ into bins in $\mathbf{I}'$, while guaranteeing that no bin in both $\mathbf{I}, \mathbf{I}'$ holds more than $B/4$ real elements. Note that the access pattern in this procedure is again determined by the edges of the graph G, which is independent to the input array $\mathbf{I}$. It is important that the graph has a linear number of edges and the set of edges can be computed in linear time.

We describe how to obliviously find the aforementioned matching in linear time. A non-oblivious linear time algorithm that achieves exactly this was given by Pippenger [35], but this algorithm is non-oblivious. A naïve approach to make this algorithm oblivious introduces a logarithmic time overhead [9]. While this is too expensive as is, we still use this procedure as a subroutine that we can apply on lists of size roughly $n/\log^2 n$ (since $n/\log^2 n \cdot \log(n/\log^2 n) = O(n)$). We additionally observe that for lists of very short size, around $\log n$, we can also implement Pippenger's algorithm in linear time in the input (in our case, around $\log n$) by "compacting" many nodes into one word and acting on them "in parallel". Another important observation is that the computation of the matching can be done on "metadata" and does not involve the actual moves of the balls. Let us see why these are useful.

Recall that to perform loose compaction, all we need is to compute the above matching but it is not known how to do this in linear time. Consider again the input array $\mathbf{I}$ for loose compaction. We break it up into $n/\log^2 n$ major bins each containing $\log^2 n$ elements. Our first goal is to extract all the major bins that are very dense, and add them to the output array. Towards this end, we treat every major bin as a single element and mark it if it contains many real elements. We perform loose compaction on the major bins (which we can afford, as we can implement Pippenger's algorithm obliviously on input size $n/\log^2 n$ in linear time in n), and we move all the dense major bins to the output array. Our second goal is to extract all real elements from the sparse major bins. We do the same trick again. We break each major bin of $\log^2 n$ elements into $\log n$ sub-bins each containing $\log n$ elements. We perform loose compaction on each such sub-bin independently (which we can since we can implement Pippenger's algorithm obliviously on such tiny, $\log n$, length inputs in linear time in n) adding all the dense sub-bins to the output array. Still, we have to perform compaction on the sparse sub-bins, each of which is of size $\log n$. Here, we can again run Pippenger's algorithm obliviously, and to extract all the real elements in the sparse sub-bins to the output array. Combining all these steps together requires some fine tuning of the parameters and the compaction factor, and the reader is referred to Section 5 for further details and for a formal description of our tight compaction algorithm.

2.5 Paper Organization

In Section 3, we provide the definitions of oblivious simulation. In Section 4, we present other building blocks that we use in our construction (some are new and some are known). In Section 5, we present our optimal oblivious tight compaction and in Section 6, we present a procedure that randomly shuffles two given a-priori randomly shuffled arrays. In Section 7, we present our basic construction of an oblivious hash table and in Section 8 we present another oblivious hash table that has optimal parameters for very small input lists. A combination of these hash tables is then used in Section 9, where we present our final construction of the ORAM scheme.

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.

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 [6, 7, 17]), 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 $\mathsf{data} \in \{0,1\}^w \cup \{\bot\}$. If $\mathsf{op} = \mathsf{read}$, then $\mathsf{data} = \bot$ 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) and $N = \mathsf{poly}(\lambda)$ as in literature, and we may use the direct implication that $w = \Theta(\log \lambda)$; We follow the convention that the CPU performs one *word-level operation* per unit time, i.e., arithmetic operations (addition, subtraction, or multiplication), bitwise operations (AND, OR, NOT, or shift), memory accesses (read or write), or evaluating a pseudorandom function [8, 18, 20, 25, 26, 32].

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 3.1 (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 the following holds:*

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

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

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 [6, 7, 17]). 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.

3.2 Modeling Reactive Functionalities

We further 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 3.1 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 3.2 (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:*

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

Definition 3.2 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).

An example: ORAM. An example of a reactive functionality is an ordinary ORAM, implementing logical memory. Functionality 3.3 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 3.3: $\mathcal{F}_{\mathsf{ORAM}}$

The functionality is reactive, and holds an internal state – N memory blocks, each of size w. Denote the internal state $\mathbf{X}[1, \ldots, N]$.

- $\mathsf{Access}(\mathsf{op}, \mathsf{addr}, \mathsf{data})$: 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 3.2 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 3.2 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 3.2, where here the access pattern might also contain $\mathcal{F}$-commands.

Concurrent composition follows from [7], 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. [2] and present a *new* oblivious sorting algorithm that is more efficient in settings where each memory word can hold multiple elements.
- **Oblivious Tight Compaction** (Section 4.2): We present an oblivious tight compaction algorithm that has $O(1)$ overhead. This procedure allows, given an array of elements where some elements are marked, to permute the elements in the array so that all marked elements end up in the front of the array.
  We extend our tight compaction algorithm to *distribution* with $O(1)$ overhead. This solves the following problem: Given an array of n elements, some of which are marked and a set $A \subseteq [n]$ where $|A|$ is equal to the number of marked elements in n, obliviously permute the elements in the array so that all marked elements end up in the locations given by the assignment A.
- **Oblivious Random Permutations** (Section 4.3): We show how to perform *efficient* oblivious random permutations in settings where each memory word can hold multiple elements.
- **Oblivious Bin Placement** (Section 4.4): We state the known results for oblivious bin placement of Chan et al. [8, 10].
- **Oblivious Hashing** (Section 4.5): 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 and Assignment** (Section 4.6): 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 4.7): 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.

4.1 Oblivious Sorting Algorithms

The elegant work of Ajtai et al. [2] 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. [2])**.** *There is a deterministic oblivious sorting algorithm that sorts n elements in $O(n \cdot \log n)$ time.*

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 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 [4]. 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 will 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.

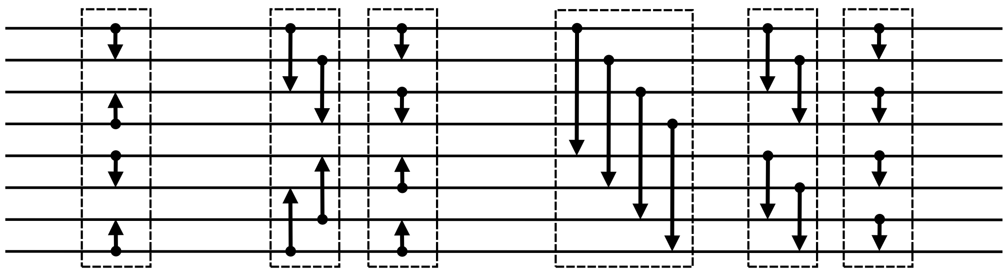

Figure 1: A bitonic sorting network for 8 inputs. Each horizontal line denotes an input from the left end and output to the right end. Each vertical arrow denotes a comparator such that compares two elements and then swaps the greater one to the pointed end. Each dashed box denotes an iteration in the algorithm. The figure is modified from [40].

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 Tight Compaction

Oblivious tight compaction solves the following problem: given an input array containing n elements 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-elements are moved to the front of the array. Several earlier works [27,29] showed how to construct *randomized* oblivious tight compaction algorithms running in time $O(n \cdot \log\log n)$. We give an $O(n)$ time, deterministic oblivious tight compaction algorithm. The proof of this theorem appears in Section 5.

Theorem 4.3 (Restatement of Theorem 1.3)**.** *There exists a deterministic oblivious tight compaction algorithm that takes $O(n)$ time to compact any input array of length n.*

It is known that any oblivious tight compaction algorithm that makes $o(n \cdot \log n)$ element movements cannot preserve *stability*. Here, stability means that all the 0-elements in the output must preserve their relative ordering in the input, and so do all the 1-elements. In particular, Lin et al. [27] showed a $\Omega(n \cdot \log n)$ time lower bound for oblivious tight *stable* compaction suffers in the balls and bins model. Indeed, all known upper bounds that overcome the $O(n \cdot \log n)$ barrier are *not* stable [27, 29].

Oblivious distribution. Let $\mathbf{I}$ be an array containing n balls where each ball is labeled as either 0 or 1, and let A be a subset of $[n]$ bits such that the number of 0-balls in $\mathbf{I}$ equals to $|A|$. The goal of *oblivious distribution* is to permute $\mathbf{I}$ into an output $\mathbf{O}$ such that for all $i \in [n]$, $\mathbf{O}[i]$ is a 0-ball in $\mathbf{I}$ if and only if $i \in A$.

Note that oblivious distribution implies oblivious tight compaction by counting 0-balls and then writing A as an array of 0s at the front. Our construction of deterministic oblivious tight compaction can be adjusted to achieve the stronger notion of oblivious distribution. See detail in Section 5.

Theorem 4.4 (Oblivious distribution)**.** *There exists a deterministic oblivious distribution algorithm that takes $O(n)$ time to permute an input array of length n.*

4.3 Oblivious Random Permutations

Let $A = (a_1, \ldots, a_n)$ be an array of n elements. We say that an algorithm ORP implements a permutation if it outputs an array $A' = (a_{\pi(1)}, a_{\pi(2)}, \ldots, a_{\pi(n)})$ for some permutation $\pi \colon [n] \mapsto [n]$. We say that ORP implements a *statistically oblivious random permutation* if the distribution over the permutation π is the uniform one and the access pattern of the algorithm is statistically close for all A and π.

Simple oblivious random permutation. A straightforward way to implement ORP, is to sample *random keys* $r_i \in \{0,1\}^\ell$ for each a_i, perform oblivious sort (see Section 4.1) by random keys r_i on the augmented array $((r_i, a_i))_{i\in[n]}$, and then output the a_i's in the sorted array. Taking $\ell = \omega(\log \lambda)$, the collision probability of any pair of r_i's is $\mathsf{negl}(\lambda)$, and thus the output permutation is completely uniform. In the standard RAM model, where word size is $\Theta(\log \lambda)$, we set $\ell = \alpha(\lambda) \cdot \log \lambda$ for any super-constant function $\alpha(\cdot)$ (for example, $\log\log(\cdot)$). Then, the comparator of ℓ bit strings takes $\Theta(\alpha(\lambda))$ time, and using the standard AKS sort, the ORP runs in $O\left(\alpha(\lambda) \cdot n \cdot \log n\right)$ time.

Theorem 4.5. *There exists an algorithm that implements a statistically oblivious random permutation in $O\left(\alpha(\lambda) \cdot n \cdot \log n\right)$ time, where $\alpha(\cdot)$ is any super-constant function (for example, $\log\log(\cdot)$).*

Packed oblivious random permutation. In the simple oblivious random permutation, the size of random keys ℓ is $\omega(\log \lambda)$ which implies that the collision probability is negligible. An alternative way is to choose a smaller ℓ such that the number of collision is *a constant fraction of* n with overwhelming probability, solve the permutation of collided elements recursively, and then run the simple ORP in the base case when the subproblem is small enough. Using this alternative, several random keys of size ℓ is short enough to be packed in to one memory word, and thus we gain the efficiency of packed oblivious sort if the elements are also short in bits. We consider the following variant of the oblivious random permutation problem on a RAM, which is useful when each memory word can hold up to B elements.

Theorem 4.6 (Packed oblivious random permutation)**.** *Let w be the number of bits in a memory word. Assume that $n \geq \log^5 \lambda$ and that elements consist of at most $\ell = O\left(\log n \cdot \log\log \lambda\right)$ bits. There exists a statistically oblivious random permutation algorithm for any array of n elements that requires $O\left(\frac{n}{B} \cdot \log^2 n\right)$ time, where $B = w/\ell$ is the number of elements each memory word can pack.*

Proof. The intuition is to instantiate the straightforward ORP from Theorem 4.5 using the packed oblivious sort from Theorem 4.2. The only subtlety is that when $n = \mathsf{poly}\log \lambda$, random keys of length $O\left(\log n \cdot \log\log \lambda\right)$ bits have a non-negligible collision probability. Nevertheless, we show that a constant fraction of keys *do not* collide and hence we are left to permute the remaining elements, which we do recursively.

Let n be the size of the original problem, m be the problem size in the recursion, and the permutation starts with $\mathsf{ORPSmall}_{n,\lambda}(A, n, n)$.

Algorithm 4.7: $\mathsf{ORPSmall}_{n,\lambda}(A, m, k)$ **— oblivious random permutation of small array**

- **Input:** An array $A = (a_1, \ldots, a_m)$ of size m and an integer $k \leq m$.
- **Public parameters:** n, λ.

- **The algorithm:**
  1. If $m \le \frac{n}{\log^3 \lambda}$, perform the straightforward oblivious random permutation and output the result. Otherwise, do the following.
  2. For $i = 1, \ldots, k$, sample random key $r_i \in \{0,1\}^\ell$ uniformly at random, where $\ell = \log n \cdot \log\log\lambda$. For $i = k+1, \ldots, m$, let $r_i = \infty$.
  3. Run packed oblivious sort on $((r_i, a_i))_{i \in [m]}$ such that elements are sorted by the r_i's. Let S be the sorted array.
  4. Mark *colliding* elements in S: any consecutive elements that have the same random key r_i are marked. Let k' be the number of marked elements.
  5. Run tight compaction (Theorem 5.1) that copies all elements from S to S' such that all k' marked elements are moved to the front of S'; record every movement of an element in an auxiliary array.
  6. Randomly permute the marked elements in S' by recursing on the first $m/2$ elements of S', i.e., $\mathsf{ORPSmall}_{n,\lambda}(S', m/2, k')$.
  7. Reverse the procedures of the tight compaction to put marked elements back (using the recorded movements in the tight compaction from Step 5).
- **Output:** The array S.

We first show that $\mathsf{ORPSmall}_{n,\lambda}$ requires $O\left(\frac{n}{B} \cdot \log^2 n\right)$ time. Each iteration takes $O\left(\frac{n}{B} \cdot \log^2 n\right)$ time, the problem size shrinks by a factor of 2, and the base case takes $O(n)$ time since $m \le \frac{n}{\log^3 \lambda}$ is a small fraction of n.

To prove the security, observe that the output is a perfectly uniform permutation conditioned on the event that the number of marked elements is at most $m/2$ at Step 6. Let X_i be a random variable indicating that the i-th element is marked as colliding at Step 4. Then, it suffices to prove that for $m > n/\log^3 \lambda$, it holds that $\sum_{i=1}^m X_i \le m/2$ with all but $\mathsf{negl}(\lambda)$ probability. For every set $S \subseteq [m]$, the X_i's satisfy the negative dependence such that $\Pr\left[\bigwedge_{i \in S} X_i = 1\right] \le \mu^{|S|}$, where $\mu = n^{1 - \log\log\lambda}$ is an upper bound on the marginal probability $\Pr[X_i = 1]$. Hence, applying the generalized Chernoff's inequality [12,31], $\Pr\left[\sum_{i=1}^m X_i > (1+\epsilon)m\mu\right] \le e^{-2m(\epsilon\mu)^2}$, taking $\epsilon = \frac{1}{2\mu} - 1$, we have

$$\Pr\left[\sum_{i=1}^m X_i > \frac{m}{2}\right] \le e^{-2m(\frac{1}{2} - \mu)^2} = e^{-\Omega(m)}.$$

Since $m > \frac{n}{\log^3 \lambda} \ge \log^2 \lambda$, the above probability is negligible in λ. □

As a special case, if $n = \Theta(\log^c \lambda)$, where $c \ge 5$ is a constant, the size of a word is $\Theta(\log \lambda)$ bits (as the standard RAM model, Section 3.1) and $B = O\left(\frac{\log \lambda}{\log^2 \log \lambda}\right)$, then the $\mathsf{ORPSmall}_{n,\lambda}$ requires $O(n)$ total time.

4.4 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 [8, 10]. In fact, these works [8, 10] 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.

4.5 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 will 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 4.8: $\mathcal{F}_{\mathsf{HT}}^{n,N}$ – Hash Table Functionality for Non-Recurrent Lookups

- $\mathcal{F}_{\mathsf{HT}}^{n,N}.\mathsf{Build}(\mathbf{I})$:
  - **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 $(k_i, v_i) \in \{0,1\}^{\log N} \times \{0,1\}^*$.
  - **The procedure:**
    1. Initialize the state state to $(\mathbf{I}, \mathbf{P})$, where $\mathbf{P} = \emptyset$.
  - **Output:** The Build operation has no output.

- $\mathcal{F}_{\mathsf{HT}}^{n,N}.\mathsf{Lookup}(k)$:
  - **Input:** The procedure receives as input a key k (that might be $\perp$, i.e., dummy).
  - **The procedure:**
    1. Parse the internal state as $\mathsf{state} = (\mathbf{I}, \mathbf{P})$.
    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^*.

- $\mathcal{F}_{\mathsf{HT}}^{n,N}.\mathsf{Extract}()$:
  - **Input:** The procedure has no input.

– **The procedure:**
  1. Parse the internal state $\mathsf{state} = (\mathbf{I}, \mathbf{P})$.
  2. Define an array $\mathbf{I}' = (a'_i, \ldots, a'_n)$ 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 naïveHT. A naïve, perfectly secure oblivious hashing scheme can be obtained directly from a perfectly secure ORAM construction [9, 11]. Indeed, we can use a perfectly secure ORAM scheme 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 scheme by Chan et al. [9] 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 5) 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. In particular, almost all known computationally secure [16, 18, 20, 25] or statistically secure [36,38,39] ORAM schemes have a (statistical) failure probability that is negligible in the problem's size and thus are 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 [36].

4.6 Oblivious Cuckoo Hashing and Assignment

A Cuckoo hashing scheme [30] is a hashing method with very efficient lookup. The input consists of n balls, each tagged with a key, and we assign them into two tables, each consisting of $\mathsf{c}_{\mathsf{cuckoo}} \cdot n$ bins, where $\mathsf{c}_{\mathsf{cuckoo}} > 1$ is an appropriate fixed constant. Each bin can hold at most one ball. The assignment of balls into bins is done by letting each ball choose one independent random bin in each table (e.g., by evaluating a PRF on the ball's associated key), and attempting to place itself into one of these two bins. It is known that with all but inverse polynomial probability in the number of balls (over the bin choices), each ball will succeed in placing itself in one of the two associated bins.

To make this failure probability negligible, a well known solution is to introduce a secondary array S of size s, called a *stash*, for "stashing" problematic elements that cannot be otherwise stored [24]. After running the initialization procedure of the Cuckoo hashing scheme, except with $O(n^{-s})$ probability, every ball is either in one of the two bins of its choice or in the stash. Thus, to look up a ball identified by a given key, unless the ball is in the stash S, lookup can be accomplished simply by examining the ball's two bin choices.

Absent any privacy concerns, it is known that building a Cuckoo hash table over n balls takes $O(n)$ time with high probability. However, it is also known that the standard procedure for building a Cuckoo hash table leaks information through the algorithm's access patterns [8,20,37]. We would like to *obliviously* build the Cuckoo hash table. Goodrich and Mitzenmacher [20] (see also the recent work of Chan et al. [8][6]) showed that a Cuckoo hash table containing $n \geq \log^8 \lambda$ balls can be obliviously built in $O(n \cdot \log n)$ total time.

Oblivious Cuckoo assignment. To obtain an oblivious Cuckoo hashing scheme it is sufficient to solve the Cuckoo assignment problem obliviously. 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 choices of the n balls, where $u_i, v_i \in [\mathsf{c_{cuckoo}} \cdot n]$ for $i \in [n]$. In the *Cuckoo assignment* problem, given as input $\mathbf{I}$, the goal is to output an array $\mathbf{A} = \{a_1, \ldots a_n\}$, where $a_i \in \{u_i, v_i, \mathtt{stash}\}$ denotes that the i-th ball k_i is assigned either to bin u_i or bin v_i, or to 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. Given a correct assignment $\mathbf{A}$, the Cuckoo hash table can be built by running oblivious bin placement algorithm (Section 4.4) that places each ball to its assigned bin or the stash.[7]

Theorem 4.10 (Obliviously Cuckoo assignment [8, 20])**.** *Let $\alpha(\cdot)$ be any arbitrarily small, fixed super-constant function. Given a uniformly random input array $\mathbf{I} = ((u_1, v_1), \ldots (u_n, v_n))$ of length $n \geq \log^8 \lambda$. There exists a procedure* $\mathsf{cuckooAssign}$ *such that, except with* $\mathsf{negl}(\lambda)$ *failure probability, is oblivious and computes a correct Cuckoo assignment with an $O(n)$-sized main table and a $O(\log \lambda)$-sized stash.*

Moreover, $\mathsf{cuckooAssign}$ *proceeds in iterations such that each iteration performs only $O(1)$ number of oblivious sorts and additionally a linear scan of an array. Instantiating the oblivious sort with Theorem 4.1,* $\mathsf{cuckooAssign}$ *runs in $O(n \cdot \log n)$ time. In addition, for every input $\mathbf{I}$, the output of* $\mathsf{cuckooAssign}$ *is a deterministic function of $\mathbf{I}$.*

At a high level, the algorithm proceeds in quasi-logarithmic number of iterations. It is parametrized by two constants $\beta < 1$ and c such that the length of the array handled in the current iteration shrinks as follows:

- For each $k \in [c \cdot \log n]$, the k-th iteration is performed on an array of $\Theta(n \cdot \beta^k)$ elements.
- For each $k \in [c \cdot \log n + 1, \alpha(\lambda) \cdot \log \lambda]$, the k-th iteration is performed on an array of $\Theta(n^{0.87})$ elements.

For completeness, we provide additional details about the construction of $\mathsf{cuckooAssign}$ in Appendix A.

Oblivious Cuckoo hashing. To obtain an oblivious hashing $(\mathsf{Build}, \mathsf{Lookup}, \mathsf{Extract})$ as defined in Section 4.5, the construction (which includes $\mathsf{cuckooAssign}$) given by Chan et al. [8] implements

[6]Chan et al. [8] is a re-exposition and slight rectification of the elegant ideas of Goodrich and Mitzenmacher [20]; also note that the Cuckoo hashing appears only in the full version of Chan et al., `http://eprint.iacr.org/2017/924`.

[7]The description here slightly differs from previous works (e.g., [8]). 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 Section 4.6.1 for a discussion on this property that we call *indiscrimination*.

Build, but read-only Lookup and no $\mathsf{Extract}$. We augment the construction as follows. To construct Lookup on a query q, we perform the standard Cuckoo hash Lookup that first scans the stash S, if q is a dummy query or q is found in S, then read two random choices in the Cuckoo table; otherwise, evaluate the PRF (that is used in Build) on q and then read the two choices of q in the table. If the key is found (in S or the table), we also remove the element. To perform $\mathsf{Extract}$, we obliviously shuffle all unvisited elements using oblivious random permutation (Theorem 4.5), which runs in $O\left(\alpha(\lambda) \cdot n \cdot \log n\right)$ time, where $\alpha(\lambda) = O\left(\log\log\lambda\right)$. We denote the resulting scheme $\mathsf{cuckooHT}$ and state its properties next.

Corollary 4.11 ($\mathsf{cuckooHT}$). *Let $\alpha(\lambda) = O\left(\log\log\lambda\right)$ be a fixed super-constant function, n be a number such that $\log^8 \lambda \leq n \leq \mathsf{poly}(\lambda)$. Assume that one-way functions exist. Then, $\mathsf{cuckooHT} = (\mathsf{Build}, \mathsf{Lookup}, \mathsf{Extract})$ is a computationally secure oblivious hashing scheme that supports n elements and has the following properties:*

- Build *takes as input $\mathbf{I}$, outputs a Cuckoo table T and a stash S. It requires $O\left(n \cdot \log n\right)$ time.*
- Lookup *queries the table T a constant number of times $O(1)$ and perfoms a linear scan of the stash S, which requires $O\left(\log\lambda\right)$ time.*
- $\mathsf{Extract}$ *requires $O\left(n \cdot \log^2 n\right)$ time.*

Dummy elements. The above algorithms, including $\mathsf{cuckooHT}$ and $\mathsf{cuckooAssign}$, naturally extend to input lists that consists of dummy elements. That is, some k_i's in the array $\mathbf{K} = (k_1, \ldots, k_n)$ (or correspondingly some (u_i, v_i)'s in the array $\mathbf{I} = ((u_1, v_1), \ldots, (u_n, v_n)))$ could be dummy elements. This follows directly from the construction of $\mathsf{cuckooAssign}$ (see Appendix A for details).

4.6.1 Packing and Indiscriminating

We will use the oblivious Cuckoo hashing $\mathsf{cuckooHT}$ in a black-box way, but we will also use $\mathsf{cuckooAssign}$ to construct another instantiation of Cuckoo hashing. This instantiation is more efficient whenever the input list $\mathbf{I}$ is short (e.g., poly-logarithmic in λ). To this end, we observe that the time of $\mathsf{cuckooAssign}$ is dominated by a sequence of oblivious sorts. Thus, for small input size n, we can use the packed oblivious sort (Section 4.1) to achieve better efficiency. Recall that the word size is $\Theta(\log\lambda)$ bits in the standard RAM model, the two choices of each element is $O\left(\log n\right)$ bits, and the algorithm of Cuckoo assignment works only on the array $\mathbf{I}$ of two choices. Hence, a word can pack $B = O\left(\frac{\log\lambda}{\log n}\right)$ choices, and thus packed oblivious sort runs in $O\left(\frac{n}{\log\lambda} \cdot \log^3 n\right)$ time. For $n = \mathsf{poly}\log\lambda$, such $\mathsf{cuckooAssign}$ runs in $O(n)$ time. However, this does not imply that we have an oblivious Cuckoo hashing runs in $O(n)$ time. Indeed, oblivious bin placement still takes $O\left(n \cdot \log n\right)$ time because packed oblivious sorting is not efficient for large balls.

Handling dummies: $\overline{\mathsf{cuckooAssign}}$. We extend the $\mathsf{cuckooAssign}$ algorithm to handle input arrays that also include dummy elements. For readability, we extend the Cuckoo assignment such that it handles $O(n)$ dummy elements in the input, and, if $\mathbf{A}[i]$ is $\texttt{stash}$, it outputs the offset in the stash. Let $n_{\mathsf{cuckoo}} = 2 \cdot \mathsf{c}_{\mathsf{cuckoo}} \cdot n + \log\lambda$ be the size of $\mathbf{I}$ such that $\mathbf{I}$ consists of n real elements and $n_{\mathsf{cuckoo}} - n$ dummy elements, the set $[2 \cdot \mathsf{c}_{\mathsf{cuckoo}} \cdot n]$ be the indices of the main table in Cuckoo hash, the set $S_{\mathsf{stash}} = [2 \cdot \mathsf{c}_{\mathsf{cuckoo}} \cdot n + 1, n_{\mathsf{cuckoo}}]$ be the indices of the stash. The following algorithm $\overline{\mathsf{cuckooAssign}}$ takes as input $\mathbf{I}$ and outputs the required assignment by $\mathsf{cuckooAssign}$ and oblivious sort.

1. For every $i \in [n_{\mathsf{cuckoo}}]$, label the element $\mathbf{I}[i]$ with a tag $t = i$. Run oblivious sort on $\mathbf{I}$ such that all real elements are in the front and any tie is resolved by putting the smaller tag in the front. Let $\widetilde{\mathbf{I}}$ be the result.
2. Run cuckooAssign (Theorem 4.10) on the first n elements of array $\widetilde{\mathbf{I}}$ and let $\mathbf{A}$ be the result (for all $i \in [n]$, $\mathbf{A}[i]$ is either one choice of $\widetilde{\mathbf{I}}[i]$ or the symbol `stash`).
3. In one linear scan on $\mathbf{A}$, replace the i-th `stash` symbol with the value $2 \cdot \mathsf{c}_{\mathsf{cuckoo}} \cdot n + i$. Append $n_{\mathsf{cuckoo}} - n$ dummy elements, $\perp$, to the end of $\mathbf{A}$.
4. Run oblivious bin placement (Section 4.4) on the appended $\mathbf{A}$ such that for each $i \in [n_{\mathsf{cuckoo}}]$, $\mathbf{A}[i]$ is placed to the t_i-th position of a newly created array **Assign**.
5. Output the array **Assign**.

Indiscrimination. A property of the $\overline{\mathsf{cuckooAssign}}$ algorithm (that we will use in our proof of security) is what we call "indiscriminate hashing" (we hinted to this property in Footnote 7 above).

Definition 4.12 (Indiscriminate hashing)**.** *For any i such that $\mathbf{I}[i]$ corresponds to a real ball, the bin assignment of the i-th ball is a function fully determined by the index i and $\mathbf{I}$.*

This property is useful since it allows us to swap the randomness that corresponds to two elements i and j that two real balls receive (while fixing all other randomness), thereby causing the i-th element and the j-th element to swap their positions in the Cuckoo hash table. This property holds directly by the fact that the output of cuckooAssign is a function of the input $\mathbf{I}$ in the above abstraction.[8]

Proposition 4.13. *Given the array $\mathbf{I}$ of length $n_{\mathsf{cuckoo}} = 2 \cdot \mathsf{c}_{\mathsf{cuckoo}} \cdot n + \log \lambda$ such that $n = \log^8 \lambda$, the algorithm $\overline{\mathsf{cuckooAssign}}$ obliviously computes a valid Cuckoo assignment* **Assign***, except with probability $\mathsf{negl}(\lambda)$, where the probability is taken over the randomness of the input array $\mathbf{I}$. In addition, the algorithm requires $O(n)$ time and satisfies the indiscriminate hashing property.*

Proof. The $O(n)$ time follows by the fact that any element in the procedure is $O(\log n)$ in bits, which implies that packed oblivious sorting (Theorem 4.2), cuckooAssign (Theorem 4.10, instantiated by packed oblivious sort), and oblivious bin placement (Section 4.4, instantiated by packed oblivious sort) all run in $O\left(\frac{n}{\log \lambda} \cdot \log^3 n\right)$ time, which is bounded by $O(n)$ since $n = \log^8 \lambda$. The security follows directly by the security of the underlying building blocks and correctness follows since the only point of failure is cuckooAssign which fails with probability at most $\mathsf{negl}(\lambda)$.

To show the indiscriminate hashing property, recall that the result of cuckooAssign is determined by its input, the first n elements of $\widetilde{\mathbf{I}}$ (Theorem 4.10). Moreover, $\widetilde{\mathbf{I}}$ is determined by $\mathbf{I}$, which implies that **Assign** is also determined by $\mathbf{I}$. It follows that for every i such that $\mathbf{I}[i]$ is real, $\mathbf{Assign}[i]$ is fully determined by i and $\mathbf{I}$. ☐

4.7 Oblivious Dictionary

As opposed to the oblivious hash table from Section 4.5, 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 pre-determined 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

[8] This property is not guaranteed to hold in previous works [8, 20] since the tie resolution of oblivious sorting there depended also on the data of input balls instead of only the two choices. See Appendix A for details.

to be made and they could be mixed arbitrarily. Another difference between our hashing notion and the dictionary notion is that the $\mathsf{Extract}$ operation outputs all elements, including "visited" elements (while $\mathsf{Extract}$ of oblivious hashing outputs only "unvisited" elements). In summary, an oblivious dictionary realizes Functionality 4.14 described below.

Functionality 4.14: $\mathcal{F}^n_{\mathsf{Dict}}$ – 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 $\bot$, i.e., dummy).
  - **The procedure:**
    1. Initialize $v^* := \bot$.
    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 4.15 (Perfectly secure oblivious dictionary)**.** *For every capacity n, there exists a perfectly secure oblivious dictionary* $(\mathsf{Init}, \mathsf{Insert}, \mathsf{Lookup}, \mathsf{Extract})$ *such that the time of each operation is* $O\left(n \cdot \log^3 n\right), O\left(\log^4 n\right), O\left(\log^4 n\right), O\left(n \cdot \log^3 n\right)$*, respectively.*

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 [13]. To achieve obliviousness, it suffices to compile the algorithms and the data structure using a perfect ORAM [9], which incurs an $O\left(\log^3 n\right)$ time overhead factor. □

5 Oblivious Tight Compaction

In this section we describe a procedure 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. Our main result is an optimal deterministic algorithm to achieve this.

Theorem 5.1 (Restatement of Theorems 1.3 and 4.3)**.** *There exists a deterministic oblivious tight compaction algorithm that takes $O(n)$ time to compact any input array of length n.*

Note that our algorithm is not stable. Stability means that all the 0-balls in the output must preserve their relative ordering in the input, and so do all the 1 balls. Indeed, it is known that any oblivious tight compaction algorithm that makes $o(n \cdot \log n)$ element movements cannot preserve stability. Lin et al. [27] showed a $\Omega(n \cdot \log n)$ time lower bound for oblivious tight *stable* compaction in the balls and bins model. As a result, all known upper bounds that overcome the $O(n \cdot \log n)$ barrier are not stable [27, 29].

In addition to tight compaction, our approach extends to *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 to 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.

Roadmap. Our algorithm relies (in multiple parts) on the existence of expander graphs and is presented by a sequence of modular steps. In Section 5.1, we give the required preliminaries on expander graphs. Then, in Section 5.2 we reduce the problem of tight compaction to a weaker abstraction called *loose* compaction. In Section 5.3, we present an efficient algorithm for loose compaction. The algorithm of distribution is presented in Section 5.4.

5.1 Preliminaries on Expanders

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*

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

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 theorems are well known (c.f. Pippenger [35]) and we provide a proof below 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 used before).

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, $\Gamma(U) = \{v \in V \mid \exists u \in U, (u,v) \in E\}$).

To prove Theorem 5.2 we need several standard definitions and results.

Definition 5.3 (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 5.4 (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 [28]. 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 [28], Gabber and Galil [14], and Jimbo and Maruoka [23]) 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 5.4). □

5.2 Reducing Tight Compaction to Loose Compaction

In this section, we reduce the problem of tight compaction in linear time to loose compaction in linear time. Recall that in tight compaction, given an input array $\mathbf{I}$ where all balls are marked with either 0 or 1, the goal is to reorder the balls such that all 0 balls will appear in the front of the array, and all 1 balls will appear at the end.

A loose compaction algorithm is parametrized by a sufficiently large constant ℓ (which will be chosen in Section 5.3), 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" exactly equals to 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 to 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 5.5.

Algorithm 5.5: $\mathsf{TightCompaction}(\mathbf{I})$**:**

- **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 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 $\mathsf{SwapMisplaced}(\mathbf{I})$, 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 is equal to 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 5.3.
- 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 is equal to 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 5.7 and we have the following claim.

Claim 5.6. *Let $\mathbf{I}$ be any input where the number of balls marked* red *equals to 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 5.7: 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 $\binom{d}{2}n$ neighbors of distance 2, and the running time is $O(n)$ since $d = O(1)$.

Algorithm 5.8: $\mathsf{LooseSwapMisplaced}_\ell(\mathbf{I})$

- **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 to 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 5.9. *Let $\mathbf{I}$ be an input array in which the number of balls marked* blue *equals to the number of balls marked* red*. Denote as $\mathbf{O}$ the output array. Then, $\mathbf{O}$ is a permutation of all n balls in $\mathbf{I}$ such that the following holds for every $i = 1, \ldots, n$:*

1. *If $\mathbf{I}[i]$ is marked $\perp$, then $\mathbf{O}[i] = \mathbf{I}[i]$.*

2. *If* $\mathbf{I}[i]$ *is marked* red *(resp.* blue*), then* $\mathbf{O}[i]$ *is either*
   (a) $\mathbf{I}[i]$ *and marked* red *(resp.* blue*) or;* *(b)* $\mathbf{I}[j]$ *element marked* blue *(resp.* red*) for some* j*.*

In $\mathbf{O}$*, the number of balls marked* red *equals to 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 to 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 $|R_{\mathsf{red}}| = |R_{\mathsf{blue}}| \le n/(2\ell)$, 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 $\epsilon \le \frac{1}{2\sqrt{\ell}}$, it holds that

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

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

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

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 $|L'| > n/2$. ☐

5.3 Loose Compaction

The algorithm $\mathsf{LooseCompaction}_\ell$ receives as input an array $\mathbf{I}$ consisting of n balls, where at most n/ℓ are real and the rest are dummies. The goal is to return an array of size $n/2$ where all the real balls reside in the returned array. The intuition of using bipartite expander graph and matching is known in [9], but we present it for completeness and then show our novel improvements. The main idea of the procedure is to first distribute the real balls to many bins, while ensuring that no bin consists of too many real balls. Particularly, we will show, for some $B = O(1)$, a method to distribute all real balls to $2n/B$ bins such that each bin will have at most $B/4$ real balls (and the rest are dummies). At this point, as all bins have small load, we can merge 4 bins together easily and compact the array by “folding” it.

Compacting the array after balanced distribution. In more details, given an input array $\mathbf{I}$, we first interpret it as n/B bins of size B each, simply by considering the array $\mathbf{I}[(i-1)\cdot B+1,\ldots,iB]$ as the i-th bin, for $i = 1,\ldots,n/B$. If a bin contains more than $B/4$ real balls, then we say that the bin is "dense", and otherwise we call the bin "sparse". Our goal is to distribute all dense bins in $\mathbf{I}$. Towards this end, we allocate another set of n/B bins of size B each, denoted as an array $\mathbf{I}'$ of dummy balls. As we will show below, there is a way to compute which target bins in $\mathbf{I}'$ we should distribute each one of the dense bins in $\mathbf{I}$, such that, the distribution would be balanced. In particular, the balanced distribution guarantees that no bin in $\mathbf{I}'$ receives more than $B/4$ real balls.

After this balanced distribution, we can compact the arrays $\mathbf{I},\mathbf{I}'$ into an array of size $n/2$ by "folding": Interpret $\mathbf{I} = (\mathbf{I}_0,\mathbf{I}_1), \mathbf{I}' = (\mathbf{I}'_0,\mathbf{I}'_1)$ where $|\mathbf{I}_0| = |\mathbf{I}_1| = |\mathbf{I}'_0| = |\mathbf{I}'_1|$ and each array consists of $n/(2B)$ bins of size B; Then, for every $i = 1,\ldots,n/(2B)$, we merge all real balls in $(\mathbf{I}_{0,i},\mathbf{I}_{1,i},\mathbf{I}'_{0,i},\mathbf{I}'_{1,i})$ into a bin of size B. As no bin consists of more than $B/4$ real balls, there is enough "room" in $\mathbf{I}_0$ to perform this folding. We then just output the concatenation of all these $n/(2B)$ bins, i.e., we return an array of size $n/2$.

Balanced distribution of the dense bins. The distribution of dense bins in $\mathbf{I}$ into $\mathbf{I}'$ relies again on expander graphs. Fixing a proper constant ϵ, we consider a d_ϵ-regular graph $G_{\epsilon,n/B} = (L,R,E)$ with $|L| = |R| = n/B$, where L corresponds to $\mathbf{I}$, R corresponds to $\mathbf{I}'$ and we let $B = d_\epsilon/2$. Let $S \subset L$ be the set of dense bins in L. We then look for a $(B,B/4)$-*matching for* S: We look for a set of edges $M \subseteq E$ such that, on one hand, from every bin in S there are at least B out edges, and on the other hand, for every bin in R there are at most $B/4$ incoming edges. Given such a matching M, every dense bin in $\mathbf{I}$ can be distributed to $\mathbf{I}'$ while guaranteeing that no bin in $\mathbf{I}'$ will have load greater than $B/4$, while the access pattern corresponds to the edges of the graph, which is public, linear, and known to the adversary.

Computing the matching. So far we showed how one can perform loose compaction once the $(B,B/4)$-matching is given. However, we still did not discuss whether such a matching exists, and if so, how to find it – obliviously. To this end, we now turn to show how to find the matching obliviously (which implies the existence directly).

We first describe a non-oblivious algorithm for finding the matching by Pippenger [35]. For the following, we let $m = |L| = |R|$.[9] The algorithm proceeds in rounds, where initially all *dense* vertices in L are "unsatisfied", and in each round:

1. **Each unsatisfied *dense* vertex $u \in L$:** Send a request to each one of the neighbors of u.
2. **Each vertex $v \in R$:** If v receives more than $B/4$ requests in this round, it replies with "negative" to all the requests it received in this round. Otherwise, it replies "positive" to all requests it received.
3. **Each unsatisfied *dense* vertex $u \in L$:** If u received more than B positive replies then take these edges to the matching and change the status to "satisfied".

The output is the edges in the matching. In each round, there are $O(m)$ transmitted messages, where each message is in fact just a single bit. Using properties of the expander graphs, we will show that in each round the number of unsatisfied vertices decreases by a factor of 2. Thus, the

[9]Note that we are working here with a parameter m and not n, as m is the number of vertices in the graph G – e.g., the number of bins and not the number of balls.

algorithm proceeds in $O(\log m)$ rounds, and the total runtime of the algorithm is $O(m)$.[10] However, the algorithm is non-oblivious.

Making it oblivious: Slow matching. One way to make this algorithm oblivious is by sending from each vertex in L to all vertices in R in each round, that is, once a vertex becomes satisfied it still sends fictitious messages in the proceeding rounds. In particular, in each round the algorithm hides whether a vertex $v \in L$ is in the set of satisfied vertices ($v \notin L'$) or is still unsatisfied ($v \in L'$), and in fact, we run each iteration on the entire graph. However, this results in algorithm that takes overall $O(m \cdot \log m)$ time. The above algorithm is previously known in [9], and we present our new contribution in the following.

Oblivious fast matching for small m. A new important observation is that when m is "small", i.e., $m \leq \frac{w}{\log w}$ where w is the word size, then all the information required for the algorithm can be packed into $O(1)$ words. In particular, when accessing information related to one node $u \in L$, we also access at the same time all information regarding to all other nodes in L. This enables us to hide which node is being visited, i.e., whether a node is in L' or not, and therefore the algorithm can now just visit the nodes in L' and does not have to make fictitious accesses on the entire graph. As a result, when m is small (i.e., of a logarithmic size) we are able to compute the matching in $O(m)$ word-level operations.

Our word-level arithmetics support simply additions, multiplication, and bitwise operations (such as AND, OR, NOT, shift). As an example, when a word a encodes data on $m = w/\log w$ nodes $v_1, \ldots, v_m$, accessing the data with respect to the ith node involves shifts and extracting the data using AND operations, and costs $O(1)$. Note that we cannot access all vertices simultaneously (this will result in $O(m)$ operations), but the adversary cannot infer which node among the set $\{v_1, \ldots, v_m\}$ we are accessing.

Another important observation is that matching algorithm works only on "metadata" (i.e., it suffices just to consider the number of bins and which bins are dense for computing the matching) and it does not involve moving balls around.

Putting it all together. So far, we showed how one can perform the loose compaction given the matching. However, computing the matching is possible in linear time only for very small instances of the problem (i.e., when m is roughly the size of the word size, w), and for large instances, our best algorithm computes the matching in $O(m \cdot \log m)$ time. As we next show, these two ingredients enable us to achieve loose compaction for every input problem n:

- Given the array $\mathbf{I}$ of size n and word size $w = \Omega(\log n)$, we first break it into blocks of size $p^2 = (w/\log w)^2$, and our goal is to move all "dense" blocks to the beginning of the array. We find the matching obliviously using the "slow" matching algorithm, that takes $O(m \cdot \log m) = O\left(\frac{n}{p^2} \cdot \log \frac{n}{p^2}\right)$ time, which is linear. Then, compaction given the matching (by folding) takes $O(n)$. By running this compaction twice we get an output of size $n/4$, consisting of all dense blocks of size p^2.
- At this point, we want to run compaction on each one of the sparse blocks in $\mathbf{I}$ (where again, blocks are of size p^2) independently, and then take only the result of the compaction of each block for the remaining part of the output array.

[10] The set of unsatisfied vertices (in L) and its neighboring set (in R) are both stored in double-linked lists to visit and remove efficiently.

In order to run the compaction on each block of size p^2, we perform the same trick again. We break each instance into p sub-blocks of a *smaller* size p, and mark each sub-block as dense or sparse. As the number of sub-blocks we have in each instance is $p = w/\log w$, we can find the matching using the fast matching algorithm. Note that as previously, we did not handle the real balls in the sparse sub-blocks.

- Finally, we have to solve compaction of all sparse sub-blocks of the previous step. Each sparse sub-block is of size $p = w/\log w$, and thus can be solved in linear time using the fast matching algorithm.

The final output consists of the following: (i) The output of compaction of the dense block in $\mathbf{I}$ (to total size $n/4$), and (ii) a compaction of each one of the sparse blocks in $\mathbf{I}$ (sums up together to $n/4$). Note that each one of these sparse blocks (of size p^2), by itself, is divided to p sub-blocks (each of size p) and its compaction consists of (i) a compaction of its dense sub-blocks; and (ii) a compaction of each one of its sparse sub-blocks.

Roadmap. We describe the algorithms in the same ordering as the above overview. In Section 5.3.1 we describe the algorithm $\mathsf{CompactionFromMatching}$ – compacting an array given the matching via folding. In Section 5.3.2 we show how to compute the matching, both for the case where m is "big" (and therefore computing the matching is slow $\mathsf{SlowMatch}$) and when m is "small" ($\mathsf{FastMatch}$). In Section 5.3.3, we present the loose compaction algorithm.

5.3.1 Compaction from Matching

Here 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 5.10 ($(B, B/4)$-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 the following Algorithm 5.11, we show how to compact (via folding) 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 $\mathsf{ComputeMatching}_{w,G}(S)$. The implementation of Algorithm $\mathsf{ComputeMatching}$ is given later in Section 5.3.2. 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 5.11: $\mathsf{CompactionFromMatching}_{w,\mu}(A)$

- **Public parameters:** word size w and μ - the size of each ball (in words).
- **Input:** An array A 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 A as m/B bins, where each bin consists of B balls. Let S be the set of indexes i such that $A[i]$ consists of more than $B/4$ real balls (the "dense" bins in A). Let D 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. Invoke $M \leftarrow \mathsf{ComputeMatching}_{w,G_{\epsilon,m/B}}(S)$. M is a $(B, B/4)$-matching for 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 $A[u]$ to bin $D[v]$.
   (b) If $M[i] = 0$, access $A[u]$ and $D[v]$ 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 $(A[i], A[m/(2B) + i])$, $(D[i], D[m/2B + i])$ 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 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 $\frac{m}{128}$, thus $|S| \leq \frac{m}{32B}$, then it implies that M is a $(B, B/4)$-matching by ComputeMatching as we will show later in Claim 5.18. Hence the correctness holds. The following claim is immediate.

Claim 5.12. *Let A be an array of m balls, where each ball is of size μ words, and where at most $m/128$ balls are marked real. Then,* $\mathsf{CompactionFromMatching}_{w,\mu}(A)$ *is an array of $m/2$ balls that contains all real balls in A. The total running time of the algorithm is $O(\mu \cdot m)$ plus the running time of* $\mathsf{ComputeMatching}_{w,G_{\epsilon,m/B}}$.

5.3.2 Computing the Matching

Our next goal is to compute the matching. We have two cases to consider, depending on the size of the input, where each algorithm results in different running time.

Algorithm 5.13: $\mathsf{ComputeMatching}_{w,G_{\epsilon,m/B}}(S)$

- **Public parameters:** Word size w, $\epsilon = \frac{1}{64}$, d_ϵ, 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 $M \leftarrow \mathsf{SlowMatch}_{w,G_{\epsilon,m/B}}(S)$.
  2. If $\frac{m}{B} \leq \frac{w}{\log w}$, then let $M \leftarrow \mathsf{FastMatch}_{w,G_{\epsilon,m/B}}(S)$.
- **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 [9]. The **bolded** instructions in $\mathsf{SlowMatch}_{w,G_{\epsilon,m/B}}(S)$ are the ones where we pay the extra $\log m$ factor in efficiency; these accesses will be avoided in Case II (FastMatch).

Algorithm 5.14: $\mathsf{SlowMatch}_{w,G_{\epsilon,m/B}}(S)$

- **Public parameters:** Word size w, $\epsilon = \frac{1}{64}$, d_ϵ, 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 m/B that is filled by 0.
  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 $\boldsymbol{u \notin L'}$, perform fake accesses)**
     (b) For each vertex $v \in R$, if $v \in R'$, do the following **(perform fake accesses if $\boldsymbol{v \in R \setminus R'}$)**:
        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 $v \in L'$, do the following **(perform fake accesses to $\boldsymbol{u \in L \setminus L'}$).**
        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 $\boldsymbol{M}$).**
     (d) Recompute $R' = \Gamma(L')$ from the updated L'.
- **Output:** The array M.

We prove the following claim, then it implies that $|L'| = 0$ after $\log(m/B)$ iterations, and then SlowMatch outputs a correct $(B, B/4)$-matching for S in time $O(m \cdot \log m)$.

Claim 5.15. *Let $S \subset L$ such that $|S| \leq m/32B$, and let $\epsilon = \frac{1}{64}$. In each iteration of Algorithm 5.14, 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 $|R'_{\mathsf{neg}}| \cdot B/4 < e(L', R'_{\mathsf{neg}}) \leq d_\epsilon \left|L'\right| \left|R'_{\mathsf{neg}}\right| / m' + \epsilon d_\epsilon \sqrt{|L'| \left|R'_{\mathsf{neg}}\right|}$; 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 5.16. *Let $S \subset L$ such that $|S| \leq m/32B$, and let $\epsilon = \frac{1}{64}$. Then,* $\mathsf{SlowMatch}_{w,G_{\epsilon,m/B}}$ *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. The obliviousness follows as the access pattern is a deterministic function of the graph $G_{\epsilon,m/B}$, which depends only on the size m (but not the input S). The correctness is argued as follows. 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 an vertex in R' such 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 5.15, after $\log(m/B)$ iterations, we have $L' = \emptyset$, and hence every $u \in S$ have at least B edges in M. □

Case II: FastMatch $\left(\frac{m}{B} \leq \frac{w}{\log w}\right)$. Here, to improve the time, we rely on the fact that for such instances of the problem, the number of words needed to encode the whole graph is constant. Hence, the obliviousness is obtained "for free" by accessing the whole graph data. The algorithm is fully described as Algorithm 5.17. 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, in each iteration we make $O(|L'| + |R'|)$ time, and since the size of $|L'|$ is reduced by a factor of 2 in each iteration (and since $|R'| \leq d_\epsilon |L'|$), we perform $O(m)$ time in total. 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 5.17: $\mathsf{FastMatch}_{w,G_{\epsilon,m/B}}(S)$

- **Public parameters:** Word size w, $\epsilon = \frac{1}{64}$, d_ϵ, 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 $E_1[u] = v_1, \ldots, E_{d_\epsilon} = v_{d_\epsilon}$. 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 indicators.
  2. Repeat the following for $i = \log(m/B), \ldots, 1$ iterations.
     (a) 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 $\mathsf{ctr}[v_1], \ldots, \mathsf{ctr}[v_{d_\epsilon}]$.
     (b) 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.

(c) For each vertex $u \in L'$, do the following (perform total 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.

(d) Recompute $R' = \Gamma(L')$ from the updated L', and initialize again the counters.

- **Output:** The array M.

Claim 5.18. *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,* $\mathsf{ComputeMatching}_{w,G_{\epsilon,m/B}}(S)$ *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 is followed by Claim 5.16 and the time is followed by the time of $\mathsf{FastMatch}$ and $\mathsf{SlowMatch}$ in the two cases. □

5.3.3 Putting It All Together

By combining Claim 5.18 and Claim 5.12 we get the following corollary.

Corollary 5.19. *The total running time of* $\mathsf{CompactionFromMatching}_{w,\mu}$ *is $O(\mu \cdot m)$ if $m \leq \frac{w}{\log w}$, or $O(\mu \cdot m + m \cdot \log m)$ otherwise.*

The next algorithm finally 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 5.20: $\mathsf{LooseCompaction}_\ell(\mathbf{I})$

- **Public parameters:** problem size n, word size w, maximum acceptable sparsity ℓ $(= 2^{38})$.
- **Input:** An array $\mathbf{I}$ with n balls, in which at most n/ℓ are real and the rest are dummy.
- **The procedure:** Let $p = \left(\frac{w}{\log w}\right)$. 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 $\mathbf{O}_1 = \mathsf{CompactionFromMatching}_{w,\mu}(A)$. $\mathbf{O}_1$ 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 $\mathbf{O}_1' = \mathsf{CompactionFromMatching}_{w,\mu}(\mathbf{O}_1)$. $\mathbf{O}_1'$ 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:** $\mathbf{O}_1' \| \mathbf{O}_{2,1}' \| \ldots \| \mathbf{O}_{2,n/\mu}'$ (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., $p \le n < p^2$:

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 $\mathbf{O} = \mathsf{CompactionFromMatching}_{w,1}(A)$.
2. **Output:** $\mathbf{O}$.

Theorem 5.21. *Let $\ell = 2^{38}$. For any input array $\mathbf{I}$ with $|\mathbf{I}| = n$ and with at most n/ℓ real balls in $\mathbf{I}$, 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 $O(n)$.*

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

- **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 CompactionFromMatching takes as input at most $(\frac{n}{\mu})/\sqrt{\ell}$ and then $(\frac{n}{2\mu})/(\sqrt{\ell}/2)$ dense blocks, and the 128-rule 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-rule 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 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-rule 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-rule holds directly for CompactionFromMatching.

To show the time complexity, observe that, except for CompactionFromMatching, all other procedures run in time $O(n)$. Recall that in Corollary 5.19, running CompactionFromMatching on m items of size μ takes $O(\mu \cdot m)$ time if $m \le \frac{w}{\log w}$, or $O(\mu \cdot m + m \cdot \log m)$ otherwise. Observe that in all three cases, it holds that the $O(\mu \cdot m)$ term from CompactionFromMatching is $O(n)$. Hence, by the depth of the recursive call to LooseCompaction is at most 2, it suffices to show that in every case, every CompactionFromMatching run in $O(n)$ time. We proceed from Case III back to I.

- **Case III:** Note that $n < \frac{w}{\log w}$. It implies the subsequent CompactionFromMatching takes an input size $m = n < \frac{w}{\log w}$ runs in linear time by Corollary 5.19.
- **Case II:** Given that $n < \left(\frac{w}{\log w}\right)^2$, the subsequent CompactionFromMatching takes input size $m = \frac{n}{p} < \frac{w}{\log w}$. Hence, CompactionFromMatching runs in linear time as in Case III.
- **Case I:** For arbitrary n, the subsequent invocation of CompactionFromMatching, in Steps 3 and 4, takes an input size $m = n/p^2$ and then $m/2$. By Corollary 5.19, the procedure runs in time $O(\mu \cdot m + m \cdot \log m)$. (If it holds that $m \le \frac{w}{\log w}$, then it is already linear.) Fortunately, as Case I is the top-level problem in the recursion, we have that $w = \Omega(\log n)$ by the word size in the RAM model, which implies that $m = n/p^2 = O(n/\log n)$ as $p = \frac{w}{\log w}$. Thus, the total time is bounded by $O(m \cdot \log m) = O(n)$.

□

5.4 Oblivious Distribution

In oblivious distribution (mentioned in Section 4.2), 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 equals 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, SwapMisplaced achieves oblivious distribution as elaborated in Algorithm 5.22, where the set A is represented as an array of n indicators such that $i \in A$ iff $A[i] = 0$.

Algorithm 5.22: $\mathsf{Distribution}(\mathbf{I}, A)$

- **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 $\mathsf{SwapMisplaced}(\mathbf{I})$ 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 5.23 (Oblivious distribution, restatement of Theorem 4.4)**.** *There exists a deterministic oblivious distribution algorithm that takes $O(n)$ time on input arrays of size n.*

6 Interspersing Randomly Shuffled Arrays

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 $n = n_0 + n_1$.
- **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 5.4 — mark every element in $\mathbf{I}_0$ as 0-balls, mark every element in $\mathbf{I}_1$ as 1-balls, and then run $\mathsf{Distribution}$ (Algorithm 5.22) 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.*

Proof. 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.[11] 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 $\mathsf{Distribution}$ 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

[11] 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 $\mathbf{I}_0, \mathbf{I}_1$.

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 5.23. □

6.2 Interspersing Multiple Arrays

We generalize the $\mathsf{Intersperse}$ 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 $O\left(\sum_{i=1}^{k-1} (k-i) \cdot n_i\right)$ time.*

Proof. The simulator that receives $n_1, \ldots, n_k$ runs the simulator of Intersperse for $k-1$ times with the right lengths, as in the description of the algorithm. The indistinguishability follows immediately from the indistinguishability of Intersperse. For functionality, note that whenever Intersperse is applied, it holds that both of its inputs are randomly shuffled which means that the input assumption of 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 $O\left(\sum_{i=1}^{k-1}(k-i)\cdot n_i\right)$, as required. □

6.3 Interspersing Reals and Dummies

We describe a related algorithm, called 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 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 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.
- **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.*

Proof. 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). $\square$

7 BigHT: Oblivious Hashing for Non-Recurrent Lookups

Our final ORAM construction will be based on the hierarchical ORAM of Goldreich and Ostrovsky [18]. A core building block in that construction (to implement different "levels") is an oblivious hashing scheme for non-recurrent requests [8, 18].

The construction we describe in this section is *the first step* towards the oblivious hash table we use in our final ORAM construction since this construction suffers from $\mathsf{poly}\log\log$ extra multiplicative factor (which lead to similar overhead in the final ORAM construction). Nevertheless, this hash table already captures and simplifies many of the ideas in the oblivious hash table of Patel et al. [32] and can be used to get an ORAM with similar overhead to that of Patel et al. In the next Section 8, we describe how to optimize the hash table and get rid of the extra $\mathsf{poly}\log\log$ factor.

As defined in Section 4.5, an oblivious hashing scheme has three algorithms, Build, Lookup, and Extract with the following syntax:

- Build takes in an array of length n, where each coordinate is either a dummy or a real element tagged with a numerical key. It is guaranteed that all real elements have distinct keys.[12] The Build algorithm outputs a memory data structure T.
- Lookup takes in a key or $\perp$. The former is referred to as a *real* request and the latter is a *dummy* request. Operating on the in-memory data structure T, the algorithm outputs an element associated with the requested key (consistent with the input provided to the Build algorithm), or outputs $\perp$ if not found or if the request is of the form $\perp$; and
- Extract outputs an array of length n containing a list of elements that have not been looked up, padded with dummies to the maximum length n.

During the lifecycle of the hashing scheme, Build acts as a constructor and is called once upfront to initialize the hash table's data structure. Afterwards, Lookup may be called multiple times and to

[12]Looking forward, in our ORAM construction, the key of an element will be its logical address.

guarantee obliviousness, it is assumed that Lookup does not query the same real key twice (although dummy lookups can be made multiple times). Finally, Extract is a destructor that is called when the hash table may be destructor.

In this section, we implement an oblivious hashing scheme, according to Functionality 4.8, where the input to Build is assumed to be *randomly shuffled*. In other words, as long as the input array **I** provided to the Build algorithm is randomly shuffled (with a hidden permutation) and keys provided to Lookup are non-recurring, the algorithm (1) preserves obliviousness, i.e., the joint distribution of access patterns encountered in the Build, Lookup, and Extract phases are indistinguishable regardless of the inputs to these algorithms, and (2) the output of Extract consists of an array of **I** elements which include the unvisited elements and they are completely shuffled within (with hidden locations and order).

7.1 Intuition and Overview of the Construction

Let **I** denote the input array provided to the Build algorithm and suppose that $|\mathbf{I}| = n$. We think of elements here as balls. Our goal is to hash the n balls of **I** into $\frac{n}{\log^8 \lambda}$ bins, where λ denotes a security parameter.

The assignment of balls into bins is determined by a pseudo-random function (PRF). Recall that each real ball in the input **I** is associated with a distinct key denoted k. We use $\mathsf{PRF}_{\mathsf{sk}}(k)$ to determine the bin, where this ball should land in and where sk is a secret key known only to the CPU and is freshly sampled upfront in the Build algorithm.

Due to standard Chernoff bound, the load of each bin does not exceed $O(\log^8 \lambda)$, except with $\mathsf{negl}(\lambda)$ probability. Since each bin is poly-logarithmically sized, for the time being we will think of each bin as a small, *perfectly secure* oblivious hash table implemented via naïveHT (Section 4.5). For $\mathsf{poly}\log(\lambda)$-sized bins, each Lookup request consumes $\mathsf{poly}\log\log\lambda$ time. In Section 8, we describe how to optimize the data structure within each small poly-logarithmically sized bin to get rid of the $\mathsf{poly}\log\log$ factor.

A flawed strawman. To aid understanding, we first describe a strawman scheme. Intuitively, we assume that the input array **I** provided to Build has been secretly randomly shuffled. Due to this assumption, it would seem safe to directly place balls into their destined bins in the clear. The dummy balls are also placed in the bins randomly. During the Lookup phase, whenever we receive a real request, we evaluate the PRF outcome over the requested key to determine which bin to look up. If we receive a dummy request, we sample a random bin and pretend to perform look up. To simplify this overview, we ignore the details of how to realize the Extract procedure. Instead we focus on understanding the (in)security of this strawman scheme only based on the Build and Lookup phases. Further, for simplicity, we pretend that the PRF acts like a random oracle.

Indeed, since the input array **I** has been secretly and randomly shuffled, the marginal access pattern of the Build phase is simulatable. Indeed, the access pattern is distributed like a random "balls and bins" process, i.e., throwing n balls into $\frac{n}{\log^8 \lambda}$ bins. Similarly, the marginal access pattern observed in the Lookup phase is simulatable too, and follows another independent balls and bins process. However, the joint distribution of access patterns encountered in *both* the Build and Lookup phases leak information.[13] To illustrate the point, consider a special case where all elements of the input array **I** are reals. In this case, the Build phase reveals how many times each bin is hit (henceforth called the bin loads). In the Lookup phase, if we query all the keys that appeared

[13]Formally, this scheme does not satisfy Definition 3.2. While each operation is simulatable by itself, there are sequences of operations that leak non-trivial information.

in the input $\mathbf{I}$ one by one, then the bin loads revealed in the Lookup phase would be identical to the Build phase. Otherwise, if all requests in the Lookup phase are dummy (or alternatively, if all requests are real but not contained in $\mathbf{I}$), the bin loads leaked in the Lookup phase would be an independent sample and would most likely differ from that of the Build phase.

Breaking the correlations. In the strawman scheme above, although the marginal access pattern distribution of the Build and Lookup phase alone each follows a random balls and bins process, correlating the two distributions leaks information. Towards fixing this problem, the idea is to make sure that the "balls and bins" process revealed in the Build and Lookup phases are independent (no matter what inputs were used in both phases). To explain the idea, let us pretend for a moment that the input array $\mathbf{I}$ contains n real balls and no dummies (later we will get rid of this assumption and support dummy items as well). Recall that we assume that the input array is secretly randomly shuffled. Here is the main idea of the construction:

1. Build*, step 1: revealed balls and bins process.* We first throw the n real balls in $\mathbf{I}$ into $B := \frac{n}{\log^8 \lambda}$ bins. Let $\mathsf{Bin}_1, \ldots, \mathsf{Bin}_B$ denote the balls that land in each of the bins, and $|\mathsf{Bin}_1|, |\mathsf{Bin}_2| \ldots, |\mathsf{Bin}_B|$ denote the revealed bin loads. By Chernoff's bound, $\Pr[||\mathsf{Bin}_i| - \mu| \geq \delta\mu] \leq 2e^{-\mu\delta^2/3}$, where $\mu = \log^8 \lambda$ is the expectation of $|\mathsf{Bin}_i|$. Choosing $\delta = \frac{1}{2\log^2 \lambda}$, we have that each $|\mathsf{Bin}_i|$ must be $\log^8 \lambda \pm 0.5 \log^6 \lambda$ with overwhelming probability.

2. Build*, step 2: sample secret independent loads.* We sample independent loads $L_1, L_2, \ldots, L_B$ by throwing $n' = \left(1 - \frac{1}{\log^2 \lambda}\right) \cdot n$ balls into B bins. We keep $L_1, L_2, \ldots, L_B$ secret from the adversary. With overwhelming probability, we have that each L_i must be $\left(\log^8 \lambda - \log^6 \lambda\right) \pm 0.5 \log^6 \lambda$. Therefore, with overwhelming probability, $|\mathsf{Bin}_i| \geq \log^8 \lambda - 0.5 \log^6 \lambda \geq L_i$ for every $i \in [B]$. If this does not hold, then the Build operation would fail, which happens with negligible probability. We call this bad event Overflow.

3. Build*, step 3: form major bins and overflow pile.* We duplicate the bins into another identical structure. In the first structure, for each Bin_i from Step 1, we obliviously truncate it to contain only L_i real balls and we pad the bin with dummies to a capacity of $\log^8 \lambda$ (this keeps the L_i's private). This structure is henceforth called the *major bins*.

   In the second structure, we replace all the real balls in the major bins with dummies. Then, there are exactly $n - n' = \frac{n}{\log^2 \lambda}$ real balls remaining in this structure. We merge all the elements in the structure into a single list and perform oblivious tight compaction (Theorem 5.1) so that the real elements appear before the dummies. Then, we employ a standard oblivious Cuckoo hashing scheme (Corollary 4.11). This structure is henceforth called the *overflow pile.*

4. Lookup*: overflow pile first and then the major bins.* To look up each requested (real) key, we first search for the key in the overflow pile. If the key is already found in the overflow pile, we make a dummy lookup in the major bins; otherwise, we perform a real lookup in the major bins.

Overview of security. We condition on the Overflow event not happening (recall that it happens only with negligible probability). The crux of the proof is showing that after having observed the Build phase bin loads $|\mathsf{Bin}_1|, \ldots, |\mathsf{Bin}_B|$, the random bin choice made by each of the n' balls in the second balls and bins process have a joint distribution that is negligibly apart from uniform at random. Obviously, if the Lookup is for dummy or requesting an item not contained in the union of the major bins, the bin choices revealed are random (even when conditioned on the access patterns revealed in the Build phase and in the Lookup phase so far). The key observation is that a request for a ball existing in the major bins also, in the eyes of the adversary, visits a random bin, *even when conditioned on the access patterns of the* Build *phase*. This stems from the fact that the Lookup phase balls and bins process was prepared *in secret* during the Build phase and was never revealed.

We will turn the above intuition into a formal proof in Section 7.3 (after we present the full details of the construction in Section 7.2) and the proof there will additionally take into account the possibility of dummy items in the input array.

7.2 The Full Construction

In Construction 7.2 we give the implementation of Functionality 4.8. As a subroutine, we use a procedure (see Algorithm 7.1) to sample bin loads of a random balls-into-bins process.

Algorithm 7.1: SampleBinLoad – Sample bin load of "n balls into B bins" process

---

- **Input:** Two numbers $n \in \mathbb{N}$ and $B \in \mathbb{N}$.
- **The Algorithm:**
  1. Let $m := n$. For each $i = 1, 2, \ldots, B$, perform:
     (a) Sample $L_i = \mathsf{Binomial}\left(m, \frac{1}{n-i+1}\right)$, where $\mathsf{Binomial}(m, p)$ denotes the number of heads from m random coins each coming up heads with probability p.
     (b) Set $m := m - L_i$.
- **Output:** $L_1, \ldots, L_B$.

Construction 7.2: 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).
- **Input assumption:** The elements in the array are uniformly shuffled.
- **The algorithm:**
  1. Let $\mu := \log^8 \lambda$ and $B := n/\mu$.
  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 an independent smaller loads.* Execute Algorithm 7.1 to obtain $(L_1, \ldots, L_B) \leftarrow \mathsf{SampleBinLoad}(n', B)$, where $n' = n \cdot \left(1 - \frac{1}{\log^2 \lambda}\right)$. If there exists $i \in [B]$ such that $||\mathsf{Bin}_i| - \mu| > 0.5 \log^6 \lambda$ or $\left| L_i - \frac{n'}{B} \right| > 0.5 \log^6 \lambda$, 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 rest elements of Bin'_i with dummy. (L_i is not revealed during this process, by continuing iterating 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 $\frac{n}{\log^2 \lambda}$.
7. Prepare an oblivious hash table for elements in the overflow pile by calling the Build algorithm of an oblivious Cuckoo hashing scheme (Corollary 4.11). Let $\mathsf{OF} = (\mathsf{OF_T}, \mathsf{OF_S})$ denote the outcome data structure. Henceforth, we use the notation $\mathsf{OF.Lookup}$ to denote lookup operations to this standard 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 $i := \mathsf{PRF_{sk}}(k)$ 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 $\mathbf{X}' \leftarrow \mathsf{IntersperseRD}_n(\mathbf{X})$, i.e., to Algorithm 6.6.
- **Output**: $\mathbf{X}'$.

7.3 Efficiency and Security Analysis

We prove that our construction obliviously implements Functionality 4.8 for every sequence of instructions with non-recurrent lookups between two Build operations. Towards this goal, 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 4.5), an oblivious Cuckoo hashing scheme (Section 4.6) and an oblivious tight compaction algorithm (Section 5).

Theorem 7.3. *Construction 7.2 obliviously implements Functionality 4.8 for all $n \geq \log^{12}\lambda$, assuming that the input array $\mathbf{I}$ for* Build *is randomly shuffled, and assuming one-way functions. Moreover, the construction consumes $O(n \cdot \mathsf{poly}\log\log\lambda)$ time for the* Build *and* Extract *phases.* Lookup *has $O(\mathsf{poly}\log\log\lambda)$ time in addition to linearly scanning a stash of size $O(\log\lambda)$.*

Remark 7.4. *As we mentioned, Construction 7.2 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* Extract *procedures from $O(n\cdot\mathsf{poly}\log\log\lambda)$ to $O(n)$ by replacing the* naïveHT *hash table with an optimized version (we call* SmallHT*) that is more efficient for small lists. Note that while it may now seem that the overhead of* Lookup *is problematic, we will "merge" the stashes across different levels in our final ORAM construction and store them again in an oblivious hash table.*

We start with the efficiency analysis. In our oblivious hashing construction from Construction 7.2, there are $n/\log^8\lambda$ major bins and each is of size $O(\log^8\lambda)$. The size of the overflow pile is $O(n/\log^2\lambda)$. We employed a naive hash table naïveHT (see Section 4.5) for each major bin, and thus their initialization incurs

$$\frac{n}{\log^8(\lambda)} \cdot O\left(\log^8\lambda \cdot \mathsf{poly}\log\log\lambda\right) = O(n \cdot \mathsf{poly}\log\log\lambda)$$

time. The overflow pile is implemented via an oblivious Cuckoo hashing scheme (Corollary 4.11) so its initialization incurs $O(n)$ time. (This is where we use the fact that $n \geq \log^{12}\lambda$ as this implies that the overflow pile is at least of size $\log^8\lambda$.) Each access incurs $O(\mathsf{poly}\log\log\lambda)$ time from the major bins and $O(\log\lambda)$ time from the linear scan of $\mathsf{OF_S}$ the stash of the overflow pile (searching in $\mathsf{OF_T}$ incurs $O(1)$ time). The overhead of Extract is the same as that of Build.

In total, our oblivious hashing scheme consumes $O(n\cdot\mathsf{poly}\log\log\lambda)$ time for the Build phase and Extract phases. Lookup has $O(\mathsf{poly}\log\log\lambda)$ time in addition to linearly scanning a stash of size $O(\log\lambda)$.

We proceed with the proof of security. Towards this end, we present a simulator Sim that simulates the access patterns of the Build, Lookup, and Extract operations:

- **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 security of the PRF, experiments Hyb_0 and Hyb_1 are computationally indistinguishable. The proof of this claim is standard.

Claim 7.5. *For any PPT adversary $\mathcal{A}$, there is a negligible function* $\mathsf{negl}(\cdot)$ *such that*

$$|\Pr[\mathsf{Hyb}_0(\lambda) = 1] - \Pr[\mathsf{Hyb}_1(\lambda) = 1]| \leq \mathsf{negl}(\lambda).$$

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

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 $\mathsf{Extract}$ procedure with the ideal $\mathcal{F}_{\mathsf{HT}}^{n,N}.\mathsf{Extract}$ 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 $\mathsf{Extract}$. We first argue that the output of $\mathcal{F}_{\mathsf{HT}}^{n,N}.\mathsf{Extract}$ consists of the same elements as that of $\mathsf{Extract}$. Indeed, both $\mathcal{F}_{\mathsf{HT}}^{n,N}.\mathsf{Lookup}$ and Lookup mark every visited item so when we execute $\mathsf{Extract}$, 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 $\mathsf{Extract}$ is uniformly random. This is enough since $\mathsf{Extract}$ performs $\mathsf{IntersperseRD}$ 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 $\mathsf{Extract}$. 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 $\mathsf{OF.Extract}()$. 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 7.7. $\Pr[\mathsf{Hyb}_2(\lambda) = 1] = \Pr[\mathsf{Hyb}_3(\lambda) = 1]$.

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 $\mathsf{OBin}_i.\mathsf{Lookup}(\perp)$ instead of $\mathsf{OBin}_i.\mathsf{Lookup}(k)$.

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

$$|\Pr[\mathsf{Hyb}_3(\lambda) = 1] - \Pr[\mathsf{Hyb}_4(\lambda) = 1]| \leq \mathsf{negl}(\lambda).$$

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 is 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, we define the union of two graphs $X = (V_{n_1} \cup V_B, E_X)$ and $Y = (V_{n_2} \cup V_B, E_Y)$, 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 follows:

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 an independent $Z \leftarrow \mathsf{BallsIntoBins}(n', B)$, where $n' = n \cdot \left(1 - \frac{1}{\log^2 \lambda}\right)$. Let $(L_1, \ldots, L_B)$ be the loads obtained in this process and $\mu' = \frac{n'}{B}$ be the expectation of L_i for all $i \in [B]$.
3. <u>$\mathsf{Overflow}$</u>: If for some $i \in [B]$ we have that $|n_i - \mu| > 0.5 \cdot \log^6 \lambda$ or $|L_i - \mu'| > 0.5 \cdot \log^6 \lambda$, 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 $n - n'$ 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 an independent $Z \leftarrow \mathsf{BallsIntoBins}(n', B)$, where $n' = n \cdot \left(1 - \frac{1}{\log^2 \lambda}\right)$. Let $(L_1, \ldots, L_B)$ be the loads obtained in this process and $\mu' = \frac{n'}{B}$ be the expectation of L_i for all $i \in [B]$.
3. Overflow: If for some $i \in [B]$ we have that $|n_i - \mu| > 0.5 \cdot \log^6 \lambda$ or $|L_i - \mu'| > 0.5 \cdot \log^6 \lambda$, 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

$$|\Pr[\mathsf{Hyb}_3(\lambda) = 1] - \Pr[\mathsf{Hyb}_4(\lambda) = 1]| \leq \mathsf{SD}(\mathsf{AccessPtrn}_3(\lambda), \mathsf{AccessPtrn}_4(\lambda)) \leq \mathsf{negl}(\lambda).$$

Towards this goal, first, by a Chernoff bound per bin and union bound over the bins, it holds that

$$\Pr_{\mathsf{AccessPtrn}_3}[\mathsf{Overflow}] = \Pr_{\mathsf{AccessPtrn}_4}[\mathsf{Overflow}] \leq \mathsf{negl}(\lambda).$$

Thus, 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

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

Thus, following a fact on statistical distance,[14]

$$\begin{aligned}&\mathsf{SD}(\mathsf{AccessPtrn}_3(\lambda), \mathsf{AccessPtrn}_4(\lambda)) \\ &\quad \leq \mathsf{SD}(\mathsf{AccessPtrn}_3(\lambda) \mid \neg\mathsf{Overflow}, \mathsf{AccessPtrn}_4(\lambda) \mid \neg\mathsf{Overflow}) + \Pr[\mathsf{Overflow}] \leq \mathsf{negl}(\lambda).\end{aligned}$$

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

[14] 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.

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 7.8. □

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

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

By combining Claims 7.5–7.10 we conclude the proof of Theorem 7.3.

8 SmallHT: Oblivious Hashing for Small Bins

Recall that the warmup oblivious hashing scheme in Section 7 consumes $O(n \cdot \mathsf{poly} \log\log\lambda)$ time for Build and $\mathsf{Extract}$ phase, and $O(\mathsf{poly}\log\log\lambda)$ time for each Lookup request in addition to linearly scanning a stash of size $O(\log\lambda)$. To achieve our optimal ORAM result, we need to get rid of the extra $\mathsf{poly}\log\log\lambda$ factors (and we will handle the linear scanning of the stash separately).

In this section, we will describe a constant-overhead oblivious hashing scheme for bins of poly-logarithmic size. At a high level, the idea is that each major bin will no longer be implemented using a $\mathsf{naïveHT}$ (Section 4.5), but rather as an efficient oblivious hashing scheme, where Build and $\mathsf{Extract}$ consume linear amount of time and each Lookup consumes constant time (not including a linear scan of a small stash that we are going to handle separately). The functionality that we implement here is the same as in the previous section, i.e., Functionality 4.8.

8.1 Intuition and Overview of the Construction

We refer to Section 4.6 for background information on Cuckoo hashing. Here, we recall that, by [8, 20], a Cuckoo hash table containing $n \geq \log^8\lambda$ balls can be obliviously constructed in $O(n \cdot \log n)$ total time. For $n = \mathsf{poly}\log\lambda$, the total time would be $O(n \cdot \log\log\lambda)$ and we would like to get rid of the extra $\log\log\lambda$ factor.

Remark 8.1. *For the time being, the reader need not worry about how to perform lookup in the stash. Later, when we apply our oblivious Cuckoo hashing scheme to the major bins in an oblivious hash table, we will merge the stashes for all major bins into a single one and treat the merged stash specially.*

Our approach. We are only concerned about Cuckoo hashing for lists of polylogarithmic size. In linear time, 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. One important observation is that the index of each item and its two bin choices can be expressed in $O(\log\log\lambda)$ bits. This 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. Specifically, for each element, we store only its index and its two bin choices, but not the actual element which we intend to move into the Cuckoo hash table eventually.

As mentioned earlier in Section 4, if a single memory word can pack B elements, then oblivious sorting and oblivious random permutation can both be conducted in a "packed" fashion, by performing SIMD operations on multiple items at a time, where each SIMD operation on a batch of B elements can be expressed with $O(1)$ number of word-level addition, subtraction, and bitwise boolean operations. A packed oblivious sorting scheme and packed oblivious random permutation consume only $O\left(\frac{n}{B} \cdot \mathsf{poly}\log n\right)$ time. In particular, when $n = \mathsf{poly}\log\lambda$, packed oblivious sorting and oblivious random permutation can be accomplished in $O(n)$ time.

Our algorithm, called $\mathsf{SmallHT}$, is inspired by this observation. Suppose that we receive an input array $\mathbf{I}$, where $|\mathbf{I}| = \mathsf{poly}\log\lambda$. The input $\mathbf{I}$ contains both real and dummy elements, and it is guaranteed that *the real elements in* $\mathbf{I}$ *appear in random order*, where the randomness is concealed from the adversary. We would like to build a Cuckoo hash table containing a main table and a stash, and the total length of the two is $O(n)$ elements. At a high level, our idea is the following:

1. *Add dummies and shuffle.* Create a secretly and randomly shuffled array of length $n_{\mathsf{cuckoo}} = \mathsf{c}_{\mathsf{cuckoo}} \cdot n + \log\lambda$, for some constant $\mathsf{c}_{\mathsf{cuckoo}} > 0$ (that we will define below) containing all the n_R real balls from the input $\mathbf{I}$ and $n_{\mathsf{cuckoo}} - n_R$ dummies, where each dummy receives a distinct random dummy-index from $\{1, 2, \ldots, n_{\mathsf{cuckoo}} - n_R\}$, where n_{cuckoo} is the total space of the Cuckoo hashing including the main table $\mathsf{c}_{\mathsf{cuckoo}} \cdot n$ and the stash $\log\lambda$. Henceforth, a dummy with the dummy-index i is said to be the *i-th dummy.* Let $\mathbf{X}$ denote the outcome of this step.

   In Section 8.2.1, we describe how to achieve this in linear total time using a combination of $\mathsf{IntersperseRD}$, as well as packed oblivious sort and packed oblivious permutation.

2. *Evaluate assignment with metadata only.* Now, we emulate the Cuckoo hashing procedure obliviously operating only on metadata. At the end of this step, every element in the array $\mathbf{X}$ should learn which bin (either in the main table or the stash) it is destined for. Specifically, we require the following:

   - Every real element in $\mathbf{X}$ receives a bin number consistent with Cuckoo hashing scheme; and
   - The i-th dummy receives the bin number of the i-th empty bin in the Cuckoo hash table (including the main table and the stash).

   This step establishes a bijection between the elements in $\mathbf{X}$ and each position in the Cuckoo hash table (including the main table and the stash). In Section 8.2.2, we describe how to achieve this using packed oblivious sorts and packed oblivious random permutations.

3. *Actual routing of the balls into the Cuckoo hash table.* Once the metadata phase completes, each element in the array $\mathbf{X}$ routes itself to the destined position in the Cuckoo hash table and this takes $O(n)$ total time. At this point, the building of the Cuckoo hash table completes.

4. *Lookup.* In the lookup phase, every lookup request will visit two bins in the main table and also search for the requested item in the stash. If the request is dummy, two random bins in the main table are visited; otherwise, we evaluate the PRF on the requested key to determine which two bins are visited. During lookup, if the requested key is found, we mark the corresponding position as dummy and thus removing the item from the hash table. For obliviousness, even when the requested key is not in some location that is visited during lookup, we make a dummy write anyway.

   Although looking up the item in the stash may take super-constant time, we will later treat the stash specially by merging multiple stashes into a single overflow pile; so the reader need not worry about the overhead of accessing the stash right now.

5. *Extract.* Concatenate the main hash table and the stash, run oblivious tight compaction (Theorem 5.1) on the concatenated array. Truncate the outcome and output only the first n elements (note that with each Lookup we replace the retrieved real key with dummy).

Overview of security. Note that we make the routing of all balls into their final destination "in the clear", which might seem insecure. However, since in the input array $\mathbf{I}$, the real elements appear in random order, after the initial IntersperseRD procedure, the resulting array $\mathbf{X}$ is secretly and randomly permuted. Therefore, the routing of the elements in $\mathbf{X}$ into their destined bins in the Cuckoo hash table reveals a uniformly random permutation that is independent of the input one. Further, even conditioned on having observed this random permutation, the two bin choices each element makes remain uniformly random.

The hardest part in the proof is to argue why the Extract procedure outputs an array where real elements are permuted in random order. For this property to be preserved, we need a special "indiscriminate" property from the hash table construction procedure. This property says that the procedure does not discriminate balls based on the order in which they are added, their keys, or their positions in the input array. More specifically, imagine that in the input array $\mathbf{I}$, every real ball is associated with a sequence of random bits, including the random bits representing its two bin choices as well as any additional randomness that the algorithm needs. Denote by ρ_i the random bits received by the i-th element in $\mathbf{I}$.

> Indiscriminate hash table construction: For any i such that $\mathbf{I}[i]$ is a real ball, its bin assignment is a function fully determined by (ρ_i, ρ_{-i}) where ρ_{-i} denotes the set of random bits received by all other real balls besides i — importantly, ρ_{-i} is a set that is insensitive to ordering.

This property allows us to swap the randomness ρ_i and ρ_j of two real balls (while fixing all other randomness), thereby resulting with the i-th element and the j-th element *swapping* their positions in the Cuckoo hash table. Thus, a *coupling* argument will show that in the outcome of Extract, the position of each real ball is equally likely and the order is completely random (where the randomness here stems from the randomness corresponding to unvisited real items in the input).

The question is whether this property holds. It is not hard to verify that it does hold throughout the algorithm except one possible place: the Cuckoo hash table construction algorithm. A-priori it is not clear that the bin assignments are "indiscriminate", namely, they are independent of the elements themselves or from the order in which they appear. Nevertheless, in Section 4.6, we showed how to slightly modify the Cuckoo hashing bin assignment procedure of [8, 20] to achieve this property, and we formally restated this property in Proposition 4.13.

We formalize the above intuition and present a detailed proof in Section 8.3, after we present the full details of the construction in Section 8.2.

8.2 The Full Construction

In this section we give the full description of our hash table, following the high-level overview given in Section 8.1. We start with formally describing the first two steps from the overview (Steps 1 and 2) in Sections 8.2.1 and 8.2.2, respectively. Then, in Section 8.2.3 (Construction 8.7), we give the full description of the Build, Lookup, and Extract procedures.

8.2.1 Step 1 – Add Dummies and Shuffle

We are given an array $\mathbf{I}$ of length n that contains real and dummy elements. The output of this procedure (described in Algorithm 8.2) 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 8.2: 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 assigned to dummy-indexes $(\perp, 1), (\perp, 2), \ldots, (\perp, n_{\mathsf{cuckoo}} - n_R)$.
  3. Run packed oblivious random permutation (Theorem 4.6) on $\mathbf{MD}$, packing $O\left(\frac{\log \lambda}{\log n}\right)$ 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 $\mathsf{IntersperseRD}$ on $\mathbf{I}'$ (Algorithm 6.6) mixing the reals and dummies at random. Let $\mathbf{X}$ denote the outcome array.
- **Output:** The array $\mathbf{X}$.

Claim 8.3. *If $n = \log^{12} \lambda$, then Algorithm 8.2 has $O(n)$ total time.*

Proof. All steps but 3 incur $O(n)$ time by description. Note that each element in $\mathbf{MD}$ can be expressed with $O(\log n) = O(\log \log \lambda)$ bits. Thus, in Step 3, we pack $O\left(\frac{\log \lambda}{\log \log \lambda}\right)$ elements into a single memory word, so the oblivious random permutation incurs $O\left(\frac{n}{\log\lambda/\log\log\lambda}\cdot\log^2 n\right)\leq O(n)$ time. □

8.2.2 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 $\mathbf{MD_X}$ of length n_{cuckoo} 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 $\mathbf{MD_X}$. 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 8.4.

Algorithm 8.4: Evaluate Cuckoo Hash Assignment on Metadata

- **Input:** An array $\mathbf{MD_X}$ 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.
- **The algorithm:**
  1. Run the indiscriminate oblivious Cuckoo assignment, $\overline{\mathsf{cuckooAssign}}$ (Proposition 4.13), 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 $\mathbf{Assign_X}[i]=\mathsf{assignment}_i$ where $\mathsf{assignment}_i\in\{\mathsf{choice}_{i,1},\mathsf{choice}_{i,2}\}\cup S_{\mathsf{stash}}$, 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 4.4) 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 $\mathbf{Assign_X}[i]$ with a tag $t=i$ for all i. Run oblivious sorting on $\mathbf{Assign_X}$ and let $\widetilde{\mathbf{Assign}}$ be the resulted 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 8.2, 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 resulted 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 of 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 8.5. *If $n = \log^8 \lambda$, then Algorithm 8.4 has $O(n)$ total time.*

Proof. The input arrays is of size $\mathsf{c_{cuckoo}} \cdot n + \log \lambda$ and each entry is of size $O(\log n)$. By Proposition 4.13, Step 1 runs in $O(n)$ time. Steps 2 through 6 consist of a constant number of packed oblivious sorting (Theorem 4.2) invocations so they consume altogether $O(n)$ time. □

Remark 8.6 (On the indiscriminate property)**.** *This property of indiscriminate hashing (Definition 4.12) is necessary in our construction of* SmallHT *(Theorem 8.8), but was not needed in previous works that use oblivious Cuckoo hashing [8, 20, 32]. This is since we reveal* $\mathbf{Assign_X}$ *in* SmallHT*, and then we have to prove that the joint distribution of the access pattern and the output remains indistinguishable given this information. Thus, we need it to hold that* $\mathbf{Assign_X}$ *does not depend on any real key value in the original array* $\mathbf{I}$ *(except for the evaluation PRF on such keys). Otherwise, the security proof does not go through, e.g., the output of* Extract *would not be uniform.*

8.2.3 Combining it All Together

The full description of the construction is given next. It invokes Algorithms 8.2 and 8.4.

Construction 8.7: 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.
- **Input Assumption:** The real elements among $\mathbf{I}$ are randomly shuffled.
- **The algorithm:**
  1. Run Algorithm 8.2 (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 $\{0,1\}^{\log N} \to [\mathsf{c_{cuckoo}} \cdot n]$.
  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 $(\mathsf{choice}_{i,1}, \mathsf{choice}_{i,2}) \leftarrow \mathsf{PRF_{sk}}(k_i)$, 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 8.4 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 $\mathsf{SmallHT.Lookup}(k)$:

- **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 $[\mathsf{c}_{\mathsf{cuckoo}} \cdot n]$.
     (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 $\mathsf{SmallHT.Extract}()$.

- **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 $\mathbf{X}'$.

8.3 Efficiency and Security Analysis

We prove that our construction obliviously implements Functionality 4.8 for every sequence of instructions with non-recurrent lookups between two Build operations. As in Section 7, we view our construction in a hybrid model, in which we have ideal implementations of the underlying building blocks.

Theorem 8.8. *Construction 8.7 obliviously implement Functionality 4.8, assuming that the input for* Build *is randomly shuffled, and assuming one-way functions. Moreover, on input lists of size* $n = \log^8 \lambda$, Build *incurs* $O(n)$ *time,* Lookup *has constant time in addition to linearly scanning a stash of size* $O(\log \lambda)$*, and* $\mathsf{Extract}$ *incurs* $O(n)$ *time.*

We start with the efficiency analysis. The Build operation executes Algorithm 8.2 that consumes $O(n)$ time (by Claim 8.3), then performs additional $O(n)$ time, then executes Algorithm 8.4 that consumes $O(n)$ time (by Claim 8.5), and finally performs additional $O(n)$ time. Thus, the total time is $O(n)$. For Lookup, by construction, we perform in constant time in addition to lookup in the stash S which is of size $O(\log \lambda)$. The time of $\mathsf{Extract}$ is $O(n)$ by construction.

For security, 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 $\mathsf{Extract}$.** 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 8.9. *For any PPT adversary $\mathcal{A}$, there is a negligible function $\mathsf{negl}(\cdot)$ such that*

$$\left|\Pr\left[\mathsf{Hyb}_0(\lambda) = 1\right] - \Pr\left[\mathsf{Hyb}_1(\lambda) = 1\right]\right| \leq \mathsf{negl}(\lambda).$$

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 8.10. $\Pr\left[\mathsf{Hyb}_1(\lambda) = 1\right] = \Pr\left[\mathsf{Hyb}_2(\lambda) = 1\right]$.

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 $\mathsf{Extract}$ procedure with the ideal $\mathcal{F}_{\mathsf{HT}}^{n,N}.\mathsf{Extract}$ 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 $\mathsf{Extract}$. We first argue that the output of $\mathcal{F}_{\mathsf{HT}}^{n,N}.\mathsf{Extract}$ consists of the same elements as that of $\mathsf{Extract}$. Indeed, both $\mathcal{F}_{\mathsf{HT}}^{n,N}.\mathsf{Lookup}$ and $\mathsf{SmallHT.Lookup}$ remove every visited item so when we execute $\mathsf{Extract}$, 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 $\mathsf{Extract}$ is uniformly random. This is enough since $\mathsf{Extract}$ performs $\mathsf{IntersperseRD}$ 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 $\mathsf{IntersperseRD}$, Algorithms 8.2 and 8.4, 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 $\mathsf{Extract}$. 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 8.11. $\Pr[\mathsf{Hyb}_2(\lambda) = 1] = \Pr[\mathsf{Hyb}_3(\lambda) = 1]$.

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

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 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, InterspersеRD, oblivious bin assignment, and oblivious sorting), the view of the adversary in $\mathsf{Hyb}_4(\lambda)$ and $\mathsf{Hyb}_5(\lambda)$ is identical.

Claim 8.13. $\Pr[\mathsf{Hyb}_4(\lambda) = 1] = \Pr[\mathsf{Hyb}_5(\lambda) = 1]$.

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

By combining Claims 8.9–8.14 we conclude the proof of Theorem 8.8.

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 $n = \log^8 \lambda$, this modification is valid. Note that we still assume that the number of elements in the input to CombHT, is at least $\log^{12} \lambda$ (as in Theorem 7.3).

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 $\mathsf{OF_S}$ of the overflow pile from BigHT, but also linearly scanning the super-constant size stash of the corresponding major bin. It will be convenient for us to merge the different S stashes of the major bins and store the merged list in an oblivious Cuckoo hash (Section 4.6). (A similar idea has also been applied in several prior works [10, 20, 21, 25].) This results with a new hash table scheme we call CombHT.

Looking ahead, 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 $\mathsf{OF_S}$, 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 $O(\log N \cdot \log\lambda) = O(\log^2 N)$. We will store that merged stash in a dictionary, and accessing this merged stash would cost $O(\log\log N)$ total time.

Construction 8.15: CombHT: combining BigHT with $\mathsf{SmallHT}$

Procedure $\mathsf{CombHT.Build}(\mathbf{I})$: Run Steps 1–7 of Procedure $\mathsf{BigHT.Build}$ in Construction 7.2, where in Step 7 let $\mathsf{OF} = (\mathsf{OF_T}, \mathsf{OF_S})$ 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. Merge all the stashes $\mathsf{S}_1, \ldots, \mathsf{S}_B$ from all small hash tables together. If the combined set of all stashes consists of less than $\log^8\lambda$ elements, then pad it to $\log^8\lambda$ with dummies. Call the Build algorithm of an oblivious Cuckoo hashing scheme on the combined set (Section 4.6), and let $\mathsf{CombS} = (\mathsf{CombS_T}, \mathsf{CombS_S})$ denote the output data structure, where $\mathsf{CombS_T}$ is the main table and $\mathsf{CombS_S}$ is the stash.

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

Procedure $\mathsf{Lookup}(k_i)$: The procedure is the same as in Construction 7.2, 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 $\mathsf{Extract}()$. The procedure $\mathsf{Extract}$ is the same as in Construction 7.2, except that $T = \mathsf{OBin}_1.\mathsf{Extract}() \| \ldots \| \mathsf{OBin}_B.\mathsf{Extract}() \| \mathsf{OF}.\mathsf{Extract}() \| \mathsf{CombS}.\mathsf{Extract}()$.

Theorem 8.16. *Construction 8.15 obliviously implement Functionality 4.8, assuming that the input for* $\mathsf{CombHT.Build}$ *is randomly shuffled, and assuming one-way functions. Assuming that* $|\mathbf{I}| = n$ *and* $n \geq \log^{12}\lambda$*, procedures* $\mathsf{CombHT.Build}(\mathbf{I})$ *and* $\mathsf{CombHT.Extract}$ *perform* $O(n)$ *time, and* $\mathsf{CombHT.Lookup}$ *runs in constant time in addition to searching in two stashes of size* $O(\log\lambda)$.

Proof. We start with the analysis of the overhead of the procedures. Since each stash S_i is of size $O(\log\lambda)$ and there are $n/\log^8\lambda$ major bins, the merged stash CombS has maximum size $O(n/\log^7\lambda)$. The size of the overflow pile OF is $O(n/\log^2\lambda)$. Thus, storing each of them with a oblivious Cuckoo hashing schemes requires $O(n/\log^2\lambda)$ space (resulting with $\mathsf{OF_T}$ and $\mathsf{CombS_T}$) plus an additional stash of size $O(\log\lambda)$ (resulting with $\mathsf{OF_S}$ and $\mathsf{CombS_S}$).

Thus, by construction, $\mathsf{CombHT.Build}(\mathbf{I})$ and $\mathsf{CombHT.Extract}$ performs in $O(|\mathbf{I}|)$ time. Regarding $\mathsf{CombHT.Lookup}$, it needs to perform a linear scan in two stashes ($\mathsf{OF_S}$ and $\mathsf{CombS_S}$) of size $O(\log\lambda)$ plus constant time to search in the main Cuckoo hash tables ($\mathsf{OF_T}$ and $\mathsf{CombS_T}$).

We prove obliviousness via a sequence of constructions, showing that each one of them obliviously implements Functionality 4.8.

- **Construction I:** This construction is the same as Construction 7.2, except that we replace each $\mathsf{naïveHT}$ with $\mathsf{SmallHT}$. 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 $\mathsf{SmallHT}$, 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 $\mathsf{CombHT.Build}$ we construct an oblivious Cuckoo hashing scheme as in [8] on the elements in the combined set of stashes. The resulting structure is called $\mathsf{CombS} = (\mathsf{CombS_T}, \mathsf{CombS_S})$ and it is composed of a main table and a stash. Observe that this construction is identical to Construction 8.15.

As follows from Theorems 7.3 and 8.8, Construction I obliviously implements Functionality 4.8 assuming that the input array for Build is randomly shuffled. 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, Construction II obliviously implements Functionality 4.8.

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, Construction III obliviously implements Functionality 4.8. Finally, 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). In terms of functionality, since an oblivious Cuckoo hash implements the same functionality as a linear scan, Construction IV implements functionality 4.8. The obliviousness of Construction IV stems directly from the obliviousness of the oblivious Cuckoo hash. We conclude that Construction II obliviously implements Functionality 4.8, as well. □

9 Oblivious RAM

In this section we present our final ORAM construction.

9.1 Overview and Intuition

Recall that Sections 7 and 8 together provided a construction for CombHT, an oblivious hash table for non-recurrent lookups. In this section, we utilize CombHT in the hierarchical framework of Goldreich and Ostrovsky [18] to construct our ORAM scheme. We first give a brief overview of the hierarchical ORAM framework. Then, we describe the challenges in adapting CombHT in this framework and describe how we overcome these challenges. Finally, we describe our ORAM scheme in detail. Recall that λ is the security parameter and $N = \mathsf{poly}(\lambda)$ is the capacity of ORAM. For simplicity, we assume N is a power of 2.

Hierarchical ORAM. A hierarchical ORAM consists of $O(\log N)$ levels of geometrically increasing sizes. In particular, a level l is capable of storing 2^l blocks of data and the largest level ($l = \log N$) can store the entire data. Each level in this framework is an oblivious hash table capable of supporting non-recurrent requests. Initially, all the blocks are stored in the largest level and all other levels are empty. In order to access a block with address addr, we sequentially query levels of the hierarchy starting at the smallest level. If addr is found at some level i, then for all subsequent levels a dummy block is queried instead. When a requested block $(\mathsf{addr}, \mathsf{data})$ is found in a specific level, it is marked as deleted in that level and is written back (possibly updated if it was a write request) to the smallest level of the hierarchy. Every 2^i accesses, all the logical blocks in levels smaller than i are merged to rebuild level i. In a hierarchical ORAM, obliviousness is guaranteed as long as a block is not looked up twice at a given level. The hierarchical ORAM guarantees this by (i) ensuring that a queried block is moved to a smaller level, and (ii) once a block is found in a level, only dummy blocks are queried in subsequent levels.

Our final goal is to achieve a hierarchical construction with an (amortized) $O(\log N)$ time overhead per access. In the hierarchical ORAM framework, if an oblivious hash table can perform lookup with $O(1)$ time, then the cost to read from $\log N$ levels is $O(\log N)$. Similarly, if the level can be built in linear time, the amortized cost per access to perform a rebuild operation is $O(\log N)$. Indeed, CombHT does satisfy these requirements of an oblivious hash table except (1) CombHT can be used only for tables of size $\Omega(\log^{12}\lambda)$, and (2) CombHT achieves $O(1)$ time lookup if we ignore the cost to access $O(\log\lambda)$ sized stashes. We now describe how these issues can be resolved to use CombHT in our scheme.

1. *The smallest level.* CombHT provides an $O(1)$ lookup time; however, it can only be used for levels of size $\Omega(\log^{12}\lambda)$. Typically, the smallest level in a hierarchical ORAM is $O(\log\lambda)$ sized, but it also requires $O(\log\lambda)$ time to access. Thus, if we use the latter construction for the first $12\log\log N$ levels (until the expected number of elements in a level is $\Omega(\log^{12}\lambda)$), the access cost would exceed our budget. Moreover, for levels of size $n \leq \log^8\lambda$, it is not clear if there is an oblivious hash table construction that achieves $O(1)$ time lookup and linear time build. Specifically, the known candidate $O(1)$ constructions, oblivious Cuckoo hashing and two-tier hashing, do not seem to be secure in this range [8]. We address this issue by using a *larger* smallest level such that the expected capacity of this level is $\log^{12}\lambda$. We then employ a $\mathsf{naïveHT}$ based oblivious dictionary for the smallest level, but use CombHT for subsequent levels. The smallest level thus incurs a $\mathsf{poly}\log\log\lambda$ blowup (Corollary 4.15), but this is incurred only at the smallest level and hence is an additive factor. We stress that although the capacity constraint of CombHT is satisfied, it still cannot be used at the smallest level. This is because blocks in the smallest level are added one-by-one, and not all at once. Moreover, for $n = \mathsf{poly}\log\lambda$, almost all known computationally and statistically secure schemes are either insecure or inefficient.

2. CombHT *stashes.* Accessing a stash of size $\log\lambda$ at a level requires $O(\log\lambda)$ time, which exceeds our budget. We address this by using the same technique as in Kushilevitz et al. [25]. Essentially, we merge all stashes to create a common stash of size $O(\log^2\lambda)$, which is added to the smallest level at the end of a rebuild operation.

Finally, our scheme differs from a standard hierarchical ORAM in the ordering of elements when a level rebuild is started. Typically, a level rebuild is performed using a sequence of expensive oblivious sorts which introduce fresh randomness for the blocks at that level. In order to achieve an overall $O(\log N)$ blowup, we are inspired by the technique of reusing unused randomness that was introduced by Patel et al. [32]. Thus, while building a level i, we apply $\mathsf{CombHT.Extract}()$ to levels $j < i$; the outputs at each of these levels retain the randomness from the previous build operation. We then use our linear time $\mathsf{Intersperse}$ procedure (Section 6) to create a single randomly-shuffled array of blocks.

9.2 The Construction

ORAM Initialization. Let N be the memory size. Our structure consists of one dictionary D (see Section 4.7), and $O(\log N)$ levels numbered $\ell+1,\ldots,L$ respectively, where $\ell = \lceil 12\log\log\lambda\rceil$, and $L = \lceil \log N \rceil$ is the maximal level.

- The dictionary D is an oblivious dictionary storing $2^{\ell+1}$ elements. Every element in D is of the form $(\mathsf{levelIndex}, \mathsf{whichStash}, \mathsf{data})$, where $\mathsf{levelIndex} \in \{\ell, \ldots, L\}$, $\mathsf{whichStash} \in \{\mathsf{overflow}, \mathsf{stashes}\}$ and $\mathsf{data} \in \{0,1\}^w$.
- 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 $\mathsf{found} := \mathsf{false}$, $\mathsf{data}^* := \bot$, $\mathsf{levelIndex} := \bot$ and $\mathsf{whichStash} := \bot$.
  2. Perform $\mathsf{fetched} := D.\mathsf{Lookup}(\mathsf{addr})$. If $\mathsf{fetched} \neq \bot$:
     (a) Interpret fetched as $(\mathsf{levelIndex}, \mathsf{whichStash}, \mathsf{data}^*)$.
     (b) If $\mathsf{levelIndex} = \ell$, 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:
           – Instead of visiting the stash of OF, namely $\mathsf{OF_S}$, in Construction 8.15, check whether $\mathsf{levelIndex} = i$ and $\mathsf{whichStash} = \mathsf{overflow}$, and in that case use the value data^* as the fetched element from OF. Otherwise, use $\bot$.
           – Instead of visiting the stash of CombS, namely $\mathsf{CombS_S}$, check whether $\mathsf{levelIndex} = i$ and $\mathsf{whichStash} = \mathsf{stashes}$, and in that case use the value data^* as the fetched value. Otherwise use $\bot$.
        ii. If $\mathsf{fetched} \neq \bot$, let $\mathsf{found} := \mathsf{true}$ and $\mathsf{data}^* := \mathsf{fetched}$.
     (b) Else, $T_i.\mathsf{Lookup}(\bot)$.
  4. Let $(k, v) := \{(\mathsf{addr}, \mathsf{data}^*)\}$ if this is a read operation; else let $(k, v) := \{(\mathsf{addr}, \mathsf{data})\}$. Insert $(k, (\ell, \bot, v))$ into oblivious dictionary D using $D.\mathsf{Insert}(k, (\ell, \bot, v))$.
  5. 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 $j^* := L$.
     (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 8.15) and let $\mathsf{OF_S}, \mathsf{CombS_S}$ be the associated stashes (of size $O(\log\lambda)$ each). Mark $\mathsf{full}_j := 1$.
   i. For each element (k, v) in the stash $\mathsf{OF_S}$, run $D.\mathsf{Insert}(k, (j, \mathsf{overflow}, v))$.
   ii. For each element (k, v) in the stash $\mathsf{CombS_S}$, run $D.\mathsf{Insert}(k, (j, \mathsf{stashes}, v))$.

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

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

9.3 Efficiency and Security Analysis

We prove that our construction obliviously implements the ORAM functionality (Functionality 3.3) and its amortized overhead is logarithmic.

Theorem 9.2 (Restatement of Theorem 1.1)**.** *Construction 9.1 obliviously implements Functionality 3.3. Moreover, the construction has $O(\log N)$ amortized time overhead.*

Proof. We start with the analysis of time. For the sake of amortization, let $t \geq N$ be the number of requests to Access. For each Access, Step 1–4 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 5, for every 2^ℓ requests of Access, one $\mathsf{Extract}$ and at most $O(\log^2 N)$ Insert operations are performed on the oblivious dictionary D, and at most one $\mathsf{CombHT.Build}$ on $T_{\ell+1}$. These require $O\left(2^\ell \cdot \log^3(2^{\ell+1}) + \log^2 N \cdot \log^4(2^{\ell+1}) + 2^{\ell+1}\right) = O(2^\ell \cdot \log^4 \log\lambda)$ 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 $\mathsf{IntersperseRD}_N$, and one tight compaction, all of which require linear time and thus the total time is $O(2^j)$. Hence, over t requests, the amortized time is

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

We prove obliviousness via a sequence of construction, and show that each one of them obliviously implements Functionality 3.3.

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 5c and 5(c)ii are implemented using $\mathcal{F}_{\mathsf{Shuffle}}$ of the respective size, and the compaction in Step 5(c)i is implemented using $\mathcal{F}_{\mathsf{compaction}}$. Note that in this construction, Step 5(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 $\ell = 12 \log\log\lambda$ and $L = \log N$. The internal state include an handle D to $\mathcal{F}^{2^\ell}_{\mathsf{Dict}}$, 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 $\mathsf{found} := \mathsf{false}$, $\mathsf{data}^* := \bot$, $\mathsf{levelIndex} := \bot$ and $\mathsf{whichStash} := \bot$.
2. Perform $\mathsf{fetched} := D.\mathsf{Lookup}(\mathsf{addr})$. If $\mathsf{fetched} \neq \bot$:
   (a) Interpret $\mathsf{fetched}$ as $(\mathsf{levelIndex}, \mathsf{whichStash}, \mathsf{data}^*)$.
   (b) If $\mathsf{levelIndex} = \ell$, then set $\mathsf{found} := \mathsf{true}$.
3. For each $i \in \{\ell+1, \dots, L\}$ 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. 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 $D.\mathsf{Insert}(k, (\ell, \bot, v))$.
5. 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 $\mathbf{U} := D.\mathsf{Extract}() \| T_{\ell+1}.\mathsf{Extract}() \| \dots \| T_{j-1}.\mathsf{Extract}()$ 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 $\tilde{\mathbf{U}} \leftarrow \mathcal{F}^N_{\mathsf{Shuffle}}(\tilde{\mathbf{U}})$
   (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 $i \in \{l, \dots, j-1\}$, reset T_i to empty structure and set $\mathsf{full}_i := 0$.
6. Output data^*.

Claim 9.3. *Construction 1 obliviously implements Functionality 3.3.*

Proof. Since the functionality $\mathcal{F}_{\mathsf{ORAM}}$ is deterministic, it suffices to show that the construction is correct (i.e., it computes the same output as the ideal functionality), and to present a simulator that produces an access pattern that is computationally-indistinguishable from the one produced by the real construction. 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 $T_{\ell+1}, \dots, T_L$, bits $\mathsf{full}_{\ell+1}, \dots, \mathsf{full}_L$ and counter ctr exactly as the real construction. Upon receiving a command $\mathsf{Access}(\bot, \bot, \bot)$, the simulator runs Construction 1 on input $(\bot, \bot, \bot)$.

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 $\mathsf{ctr}, \mathsf{full}_{\ell+1}, \dots, \mathsf{full}_L$. 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.

For completeness, we show that the algorithm outputs the exact same output as the functionality. Here, we rely on the correctness of the ideal functionalities of the building blocks. In particular, if some address addr has been written to the ORAM, it is never deleted. Moreover, only a single copy of the data appears in the system, as whenever an addr is been accessed, it is being deleted from its level and written to a higher level. □

Give that Construction 1 obliviously implement Functionality 3.3, we proceed with a sequence of constructions and show that each and one of them obliviously implements Functionality 3.3 as well. The constructions are listed next.

- **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 5c with $\mathsf{Intersperse}^{(j^*-\ell)}$ (Algorithm 6.4), instantiate $\mathcal{F}_{\mathsf{Shuffle}}$ in Step 5(c)ii with $\mathsf{IntersperseRD}$ (Algorithm 6.6, and instantiate $\mathcal{F}_{\mathsf{compaction}}$ in Step 5(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.** In this construction, we follow Construction 2 but instantiate $\mathcal{F}_{\mathsf{HT}}$ with Construction 8.15 (i.e., CombHT from Theorem 8.16). Note that we do not combine the stashes yet. That is, we simply replace Step 5d (as Build), Step 3 (as Lookup) and Step 5b (as $\mathsf{Extract}()$) in Construction 1 with the implementation of Construction 8.15 instead of the ideal functionality $\mathcal{F}_{\mathsf{HT}}$.
- **Construction 4.** In this construction, we follow Construction 3 but change Step 5d as in Construction 9.1: We add all elements in $\mathsf{OF_S}$ and $\mathsf{CombS_S}$ into D, marked with their level index and what stash they are coming from (where $\mathsf{whichStash} = \mathsf{overflow}$ in case that the element comes from $\mathsf{OF_S}$, and $\mathsf{whichStash} = \mathsf{stashes}$ in case that the element comes from $\mathsf{CombS_S}$).
- **Construction 5.** In this construction, we follow Construction 4, but make the following change: In Step 3, we modify the Lookup procedure of Construction 8.15, and whenever accessing $\mathsf{OF_S}$ and $\mathsf{CombS_S}$, we perform lookup at the stored values $\mathsf{levelIndex}$ and $\mathsf{whichStash}$.
- **Construction 6.** This is the same as Construction 5, where we replace the ideal implementation $\mathcal{F}_{\mathsf{Dict}}$ of the dictionary D with a real implementation. Note that this is exactly Construction 9.1.

The theorem is obtained using a sequence of simple claims, given that Construction 1 obliviously implements Functionality 3.3.

Claim 9.4. *Construction 2 obliviously implements Functionality 3.3.*

Proof. This follows by composing Claims 6.7,6.5 and Theorem 5.1. It is important to note that the input assumptions are preserved, and therefore we can replace the functionality with the respective algorithm:

- We invoke $\mathsf{Intersperse}^{(j^*-\ell)}$ (in Step 5c instead of $\mathcal{F}_{\mathsf{Shuffle}}$) on arrays that are output of $\mathsf{Extract}$ and therefore are randomly shuffled, maintaining the input assumption of Algorithm 6.4.
- We invoke $\mathsf{IntersperseRD}$ (in Step 5(c)ii on an array in which the real elements are randomly shuffled, as this is an output of compaction on a randomly shuffled array. Therefore, this maintains the input assumption of Algorithm 6.6.

□

Claim 9.5. *Construction 3 obliviously implements Functionality 3.3*

Proof. This follows from Theorem 8.16 and using composition. In particular, the input for Build in Step 5d is always randomly permuted, as this is an output of $\mathsf{IntersperseRD}$. □

Claim 9.6. *Construction 4 obliviously implements Functionality 3.3.*

Proof. The difference from Construction 3 is only by adding more elements into D, however, note that the size of D never exceeds its capacity $2^{\ell+1} = O(\log^{12}\lambda)$. This is because there are $O(\log N)$ levels and we add at most $O(\log\lambda)$ elements from each level. Moreover, note that we do not consider these added elements when we perform lookups in D. In terms of functionality, we compute exactly the same input/output behavior as in Construction 3. As for the access pattern, the change is just by adding more accesses into D which can be trivially simulated. We therefore conclude that Construction 4 obliviously implements Functionality 3.3. □

Claim 9.7. *Construction 5 obliviously implements Functionality 3.3.*

Proof. The construction is just as Construction 4, where instead of searching in each level for the elements in the stashes $\mathsf{OF_S}$ and $\mathsf{CombS_S}$, we look at the stored values $\mathsf{levelIndex}$ and $\mathsf{whichStash}$ and data^*. In case one of the elements appear in one of the stashes, it also appears in the dictionary D. In terms of functionality, the construction has the exact same input/output behavior as Construction 4. In terms of the access pattern, we skip visiting of the stashes in each level, which is just omitting (a deterministic and well defined) part of the access pattern and therefore is simulatable. □

Claim 9.8. *Construction 6 obliviously implements Functionality 3.3*

Proof. As Construction 5 is a construction in the $\mathcal{F}_{\mathsf{Dict}}$-hybrid model, the claim follows by using composition. □

This completes the proof of Theorem 9.2. □

Remark 9.9 (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 [26]. 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 $\ell = \lceil 12\log\log\lambda\rceil$ 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 $O\left(\log N - \log m)\right) = O\left(\log(N/m)\right)$ 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, a Packard Fellowship, a Sloan Fellowship, Google Faculty Research Awards, a VMware Research Award, and a Baidu Research Award.

References

[1] Miklós Ajtai. Oblivious rams without cryptogrpahic assumptions. In *Proceedings of the 42nd ACM Symposium on Theory of Computing, STOC*, pages 181–190, 2010.

[2] Miklós Ajtai, János Komlós, and Endre Szemerédi. An $o(n \log n)$ sorting network. In *Proceedings of the 15th Annual ACM Symposium on Theory of Computing, STOC*, pages 1–9, 1983.

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

[4] 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.

[5] Elette Boyle and Moni Naor. Is there an oblivious RAM lower bound? In *Proceedings of the 2016 ACM Conference on Innovations in Theoretical Computer Science, ITCS*, pages 357–368, 2016.

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

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

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

[9] T.-H. Hubert Chan, Kartik Nayak, and Elaine Shi. Perfectly secure oblivious parallel RAM. In *Theory of Cryptography - 16th International Conference, TCC 2018*, pages 636–668, 2018.

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

[11] Ivan Damgård, Sigurd Meldgaard, and Jesper Buus Nielsen. Perfectly secure oblivious RAM without random oracles. In *Theory of Cryptography - 8th Theory of Cryptography Conference, TCC*, pages 144–163, 2011.

[12] Devdatt P. Dubhashi and Desh Ranjan. Balls and bins: A study in negative dependence. *Random Struct. Algorithms*, 13(2):99–124, 1998.

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

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

[15] Craig Gentry, Kenny A. Goldman, Shai Halevi, Charanjit S. Jutla, Mariana Raykova, and Daniel Wichs. Optimizing ORAM and using it efficiently for secure computation. In *Privacy Enhancing Technologies - 13th International Symposium, PETS*, pages 1–18, 2013.

[16] Oded Goldreich. Towards a theory of software protection and simulation by oblivious rams. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing, STOC*, pages 182–194, 1987.

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

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

[19] Michael T. Goodrich. Zig-zag Sort: A Simple Deterministic Data-oblivious Sorting Algorithm Running in O(N Log N) Time. In *Proceedings of the Forty-sixth Annual ACM Symposium on Theory of Computing*, STOC '14, pages 684–693, 2014.

[20] Michael T. Goodrich and Michael Mitzenmacher. Privacy-preserving access of outsourced data via oblivious RAM simulation. In *Automata, Languages and Programming - 38th International Colloquium, ICALP*, pages 576–587, 2011.

[21] Michael T. Goodrich, Michael Mitzenmacher, Olga Ohrimenko, and Roberto Tamassia. Privacy-preserving group data access via stateless oblivious RAM simulation. In *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms, SODA*, pages 157–167, 2012.

[22] Riko Jacob, Kasper Green Larsen, and Jesper Buus Nielsen. Lower bounds for oblivious data structures. *CoRR*, abs/1810.10635, 2018. To appear in SODA 2019.

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

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

[25] Eyal Kushilevitz, Steve Lu, and Rafail Ostrovsky. On the (in)security of hash-based oblivious RAM and a new balancing scheme. In *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms, SODA*, pages 143–156, 2012.

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

[27] Wei-Kai Lin, Elaine Shi, and Tiancheng Xie. Can we overcome the $n \log n$ barrier for oblivious sorting? *IACR Cryptology ePrint Archive*, 2018:227, 2018. To appear in SODA 2019.

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

[29] 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.

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

[31] Alessandro Panconesi and Aravind Srinivasan. Fast randomized algorithms for distributed edge coloring (extended abstract). In *Proceedings of the Eleventh Annual ACM Symposium on Principles of Distributed Computing*, pages 251–262, 1992.

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

[33] Enoch Peserico. Deterministic oblivious distribution (and tight compaction) in linear time. *CoRR*, abs/1807.06719, 2018.

[34] Benny Pinkas and Tzachy Reinman. Oblivious RAM revisited. In *Advances in Cryptology - CRYPTO*, pages 502–519, 2010.

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

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

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

[38] 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 SIGSAC Conference on Computer and Communications Security, CCS*, pages 299–310, 2013.

[39] Xiao Wang, T.-H. Hubert Chan, and Elaine Shi. Circuit ORAM: on tightness of the goldreich-ostrovsky lower bound. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, CCS*, pages 850–861, 2015.

[40] Wikipedia. Bitonic sorter. https://en.wikipedia.org/w/index.php?title=Bitonic_sorter. Accessed: Aug 13, 2018.

A 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, $\mathsf{c}_{\mathsf{cuckoo}}$.

- For every integer k, the size of the largest connected component of G is greater than k with probability $O(\gamma^{-k})$
- For every integer k, let C_k be the total number of edges of all components such that the size is at least k. Then, for every integer k such that $n\beta^k > \Theta(n^{0.87})$, it holds that C_k is at most $O(n\beta^k)$ with overwhelming probability (in the security parameter λ).

Using the second tail bound, in each iteration, the BFS is pre-programmed to eliminate a constant fraction of edges such that is in a small component until the problem size is only $\Theta(n^{0.87})$; then, using the first tail bound, the BFS works on the array of $\Theta(n^{0.87})$ edges for additional $O(\alpha(\lambda) \cdot \log \lambda)$ iterations, where $\alpha(\lambda) = O(\log \log \lambda)$.

Therefore, the access pattern of the oblivious BFS is pre-determined and does not depend on the input $\mathbf{I}$. The tail bounds incurs failure in correctness for a negligible 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 $\mathsf{cuckooAssign}$ at a very high level.

[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. [8], 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 Section 4.6.1.