OptORAMa: Optimal Oblivious RAM

Gilad Asharov[1], Ilan Komargodski[1], Wei-Kai Lin[2], Kartik Nayak[3], and Elaine Shi[2]

[1]Cornell Tech
[2]Cornell University
[3]VMware Research and Duke University

September 21, 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 $O(\log N \cdot \mathsf{poly}(\log \log N))$ amortized blowup, assuming one-way functions.

Contents

1 Introduction

Oblivious RAM (ORAM), first introduced and constructed by Goldreich and Ostrovsky [17] 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 outsource an encrypted database to an untrusted server and later to 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 *work overhead* or *work blowup*.

In their original work, Goldreich and Ostrovsky [17] presented the *hierarchical ORAM* construction that allows us to compile any program into an oblivious one with amortized poly-logarithmic work 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 [17] performs them obliviously with $O(\log^3 N)$ amortized work overhead.

By extending and optimizing the building blocks used in the hierarchical ORAM constructions, improved constructions were given in subsequent works [1, 10, 18, 31]. Until very recently, the best result was given by Kushilevitz, Lu, and Ostrovsky [22] who presented an ORAM with $O(\log^2 N/\log\log N)$ amortized work blowup. See Chan et al. [7] 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. [32] who achieved $O(\log^3 N)$ work blowup (but with super constant client memory). This approach was optimized (see e.g., Gentry et al. [14]) until the Path ORAM construction of Stefanov et al. [34] that had $O(\log^2 N)$ work blowup. The Circuit ORAM of Wang, Chan, and Shi [35] has the same work 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 work blowup is required by any ORAM. This was proven in the work of Goldreich and Ostrovsky [17] for statistically secure ORAM scheme in the so called *balls and bins* model [4].[1] More recently, Larsen and Nielsen [24] showed that the same lower bound holds even for computationally secure scheme and with no assumption on the model. Bridging the gap between the logarithmic work overhead lower bound and the poly-logarithmic work 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 [29] presented an ORAM scheme, in the hierarchical ORAM framework, with *quasi-logarithmic* overhead, assuming one-way functions. Namely, their scheme has a $O(\log N \cdot \mathsf{poly}\log\log N)$ amortized work blowup and it is computationally secure.[2] 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] The stated result in [29] (see also ePrint version from April 30th, 2018) is that their scheme has $O(\log N \cdot \log\log N)$ amortized work blowup, but we were not able to verify this until the date of publication. We are in discussions with the authors of [29] to clarify this issue.

Our contribution. We provide the first *optimal* ORAM construction, having only $O(\log N)$ amortized work blowup and matching the lower bounds of Goldreich and Ostrovsky [17] and Larsen and Nielsen [24]. Our construction assumes the existence of one-way functions. Moreover, our ORAM (similarly to the one of [29]) could be instantiated to obtain statistical security in the *balls and bins* model if the client is provided with access to a private random function, which matches the assumptions in the original lower bound of Goldreich and Ostrovsky [17] mentioned above.

Theorem 1.1. *There exists an ORAM (in the* balls and bins *model) with $O(\log N)$-amortized work 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. [29]. We are also influenced by ideas from the recent elegant work of Peserico [30] on oblivious tight compaction.

The lower bound of Larsen and Nielsen [24] also relates to the case where the client's internal memory is not a constant. If the client's memory size is m, then the lower bound implies that $\Omega(\log(N/m))$ amortized overhead is inherent. We also show how to extend our construction to this scenario, and obtain the following Corollary:

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

1.1 Overview of Our Construction and Techniques

We give a high-level overview of our techniques. We start with an overview of the hierarchical ORAM framework initially proposed by Goldreich and Ostrovsky [15, 17] and improved in subsequent works [7,19,23]. We then describe the elegant ideas in the very recent work by Patel et al. [29] in which they further improved the asymptotical performance of hierarchical ORAMs, achieving *quasi*-logarithmic blowup. We then explain the obstacles for getting rid of the extra $\mathsf{poly}\log\log$ factors that arise in their construction, and describe our set of algorithmic ideas for accomplishing this. As an independent contribution, we believe our constructions are significantly simpler than those of Patel et al.

Starting point: 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 ith 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 $\mathsf{read}/\mathsf{write}$ 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 $\ell := \mathsf{L}$. Merge all $\{T_j\}_{j \le \ell}$ into T_ℓ. Mark all levels $T_1, \ldots, T_{\ell-1}$ as empty and T_ℓ as full.

With each access, we perform in total $\log N$ lookups, one per hash table. Moreover, after every t accesses, we rebuild the ith table $\lceil t/2^i \rceil$ times. When implementing the hash table using the best known oblivious hash table (e.g., oblivious Cuckoo hashing [7, 19]), building a level with 2^k items obliviously requires $O(2^k \cdot \log(2^k)) = O(2^k \cdot k)$ work blowup. This building algorithm is based on oblivious sorting, and its work overhead is inherited from the work overhead of the oblivious sort procedure (specifically, the best known algorithm for obliviously sorting n elements takes $O(n \cdot \log n)$ work [2]). Thus, summing over all levels (and ignoring the $\log N$ lookup operations across different levels with each access), t accesses require $\sum_{i=1}^{\log N} \lceil \frac{t}{2^i} \rceil \cdot O(2^i \cdot i) = O(t \cdot \log^2 N)$ work blowup. On the other hand, lookup is essentially constant work per level (ignoring searching in stashes which introduce additive factor), and the cost 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. [29]. Earlier constructions of oblivious hash tables [7, 15, 17, 19, 23] work for *every* input array, and to obliviously build a hash table would require expensive oblivious sorting, causing the extra logarithmic factor. The key idea of Patel et al. [29] 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 $O(n \cdot \mathsf{poly} \log \log \lambda)$ total work[3]; further, lookup consumes $O(\log \log n)$ total work. Patel et al. argue that their hash table construction retains security not necessarily for every input, but when the input array is randomly permuted, and moreover the input permutation must be unknown to the adversary.

To be able to leverage this relaxed hash table in hierarchical ORAM, a remaining question is the following: whenever a level is being rebuilt in the ORAM (i.e., a new hash table is being constructed), how do we make sure that the input array is randomly and secretly shuffled? A naïve answer is to employ an oblivious random permutation to permute the input, but known oblivious random permutation constructions require oblivious sorting which brings us back to our starting point. Patel et al. solve this problem and show that there is no need to completely shuffle the input array. Recall that when building some level T_ℓ, the input array consists of 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 $O(n \cdot \mathsf{poly} \log \log \lambda)$ work, 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 significantly 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 $O(n)$. Our algorithm has perfect security and perfect correctness.
2. We develop a hash table that support build in $O(n)$ assuming that the input array is randomly shuffled.

In the following, we describe the core ideas behind these improvements.

[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.

[4] The work 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 the ORAM construction).

Multi-array shuffle in $O(n)$: Intersperse. The multi-array shuffle of Patel et al. takes as input k arrays, $\mathbf{I}_1, \ldots, \mathbf{I}_k$ each of size n_i and is assumed to already be randomly shuffled. The output of the procedure is a random shuffling of all elements in $\mathbf{I}_1 \cup \ldots \cup \mathbf{I}_k$. Ignoring security issues, we could first initialize an output array of size $n = |\mathbf{I}_1| + \ldots + |\mathbf{I}_k|$. Then, we assign for $\mathbf{I}_1$ exactly $|\mathbf{I}_1|$ 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 $|\mathbf{I}_2|$ random locations in the remaining $n - |\mathbf{I}_1|$ 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. The challenge is how to perform this placement obliviously, without revealing in the output array which one of the input array it is taken. Patel et al. shows a rather involved technique for achieving this placement obliviously in total work $O(n \log \log \lambda)$ with negligible probability of failure.

We significantly simplify this construction and provide a perfectly correct and perfectly secure procedure in constant work overhead. 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 element with key 1. Recently, Peserico [30] showed how to implement oblivious tight compaction of an array of size n in $O(n)$ work. We show how to use his algorithm "in reverse" to achieve our task. The reader is referred to Section 4 for further details.

Hash table build in $O(n)$ time. In previous works, hash tables were implemented as oblivious Cuckoo hash tables, in which the build time requires $O(n \cdot \log n)$ work and relies on oblivious sorts. The hash table of Patel et al. that requires $O(n \log \log \lambda)$ and works as long as the input array is a-priori shuffled is highly non-trivial. The high-level structure itself consists of $\log \log \lambda$ layers of bins, where each layer contains at most $n/\mathsf{poly} \log \lambda$ bins of size $\mathsf{poly} \log \lambda$ each. In addition, there is a layer we refer to as an overflow pile that consists of $n/\mathsf{poly} \log \lambda$ elements. An element can be placed in one bin in one of the layers, or in the overflow pile. Each bin (of size $\mathsf{poly} \log \lambda$) and the overflow pile are implemented using a secondary structure – an oblivious Cuckoo hash table.

There are two sources for the additional $\mathsf{poly} \log \log \lambda$ factor with this approach:

1. The number of layers in the hash table is $\log \log \lambda$. This implies that the running time for Lookup is at least $O(\log \log \lambda)$, as one has to visit at least one bin per layer. Thus, in the overall ORAM construction, lookup now requires $O(\log N \cdot \log \log \lambda)$ and not $O(\log N)$.
2. The size of each bin is $O(\mathsf{poly} \log \lambda)$, and the time required for building a Cuckoo hash table on such a bin is $O(\mathsf{poly} \log \lambda \cdot \log \log \lambda)$. As the total number of bins is roughly $O(n/\mathsf{poly} \log \lambda)$, the total time for building cuckoo hashing for all bins is $O(n \cdot \log \log \lambda)$. Therefore, the build time of the overall construction is $O(n \cdot \log \log \lambda)$.

We overcome the first problem by reducing the number of layers in the hash table to be just one. As a result, each element can be placed in exactly one bin, or in the overflow pile. This allows us to achieve Lookup in each level of the ORAM in effectively constant work overhead, and therefore the work overhead of Lookup in the final ORAM construction is $O(\log N)$. This idea on its own already significantly simplifies the construction of Patel et al. We refer the reader to Section 5 for the exact details of our solution.

As for the second problem, we present an oblivious Cuckoo hash table that works in *constant* work overhead in case that the number of elements is small, namely, about $O(\log^{12} \lambda)$. Recall

that we use such a scheme within each bin. Combined with the solution of the first problem, this allows us to get rid of the additional $\log\log\lambda$ work overhead of the Build procedure. An important observation for achieving this is that in the case where number of elements is $O(\log^{12}\lambda)$, the index of each item and its associated two bin choices in the cuckoo hash tables can be expressed in $O(\log\log\lambda)$ bits. As a result, a single memory word (which is $O(\log\lambda)$ bits long) can hold $O\left(\frac{\log\lambda}{\log\log\lambda}\right)$ many elements' metadata. This allows us to perform oblivious sorting in linear time, and eventually to build the Cuckoo hash table with linear work. A major challenge is the following: even though we can work on *metadata* within linear work, still we cannot securely route the original elements with similar work overhead, as the elements themselves are of block-size each. Achieving this routing in linear work requires a rather involved Build algorithm, and many new ideas, and we elaborate on it in Section 6.

We note that the above description glosses over many new challenges that arise and we resolve. We refer to Section 5 and 6, where we provide a more elaborate overview, the full details of our algorithms, and proofs of security. Our final ORAM construction, using our optimized oblivious hash table, appears in Section 7.

1.2 Paper Organization

In Section 2, we provide the definitions of oblivious simulation and in Section 3 we present several building blocks that we use in our construction (some are new and some are known). In Section 4, we present a procedure that randomly shuffles two given a-priori randomly shuffled arrays. In Section 5, we present our basic construction of an oblivious hash table and in Section 6 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 7, where we present our final construction of the ORAM scheme.

2 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.

2.1 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 [5, 6, 16]), 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 $\mathsf{mem}[N,\beta]$, 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 β 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 $\mathsf{data} \in \{0,1\}^\beta \cup \{\perp\}$. If $\mathsf{op} = \mathsf{read}$, then $\mathsf{data} = \perp$ and the returned value is the content of the block located in logical address addr in the memory. If $\mathsf{op} = \mathsf{write}$, then the memory data in logical address addr is updated to data.

Oblivious simulation of a (non-reactive) functionality. We consider machines that interact with the memory via read/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 2.1 (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 [5, 6, 16]). 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.

2.2 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 2.1 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 2.2 (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 $\mathsf{Expt}_{\mathcal{A}}^{\mathrm{real},M}(1^\lambda)$ and $\mathsf{Expt}_{\mathcal{A},\mathsf{Sim}}^{\mathrm{ideal},\mathcal{F}}(1^\lambda)$ is computationally indistinguishable:*

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

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

Definition 2.2 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 2.3 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 2.3: $\mathcal{F}_{\mathsf{ORAM}}$

The functionality is reactive, and holds an internal state – N memory blocks, each of size β. 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 $\mathsf{data} \in \{0,1\}^{\beta}$.
  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 2.2 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 read/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 2.2 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 2.2, where here the access pattern might also contain $\mathcal{F}$-commands.

Concurrent composition follows from [6], 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.

3 Oblivious Building Blocks

Our construction uses many building blocks, some of which are known from the literature and some of which are new to this work. 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 3.1): 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 3.2): We state the result of Peserico [30].
- **Oblivious Random Permutations** (Section 3.3): We show how to perform *efficient* oblivious random permutations in settings where each memory work can hold multiple elements.
- **Oblivious Bin Placement** (Section 3.4): We state the known results for oblivious bin placement of Chan et al. [7, 9].
- **Oblivious Hashing** (Section 3.5) 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 3.6): 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 3.7): 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.

3.1 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 and $O(\log n)$ depth that can sort any array of length n.

Theorem 3.1 (Ajtai et al. [2])**.** *There is a deterministic oblivious sorting algorithm that sorts n elements in $O(n \cdot \log n)$ work and $O(\log n)$ depth.*

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.

Theorem 3.2 (Packed oblivious sort)**.** *There is a deterministic packed oblivious sorting algorithm that sorts n elements in $O\left(\frac{n}{B} \cdot \log^2 n\right)$ work and $O(\log^2 n)$ depth, where B denotes the number of elements each memory word can pack.*

Proof. We use a variant of bitonic sort, introduced by Batcher [3]. It is well known that, given a list of n elements, bitonic sort runs in $O(n \cdot \log^2 n)$ work. The algorithm, viewed as a sorting network, proceeds in $O(\log^2 n)$ 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 work. We will show the following facts: (1) each iteration runs in $O\left(\frac{n}{B}\right)$ SIMD comparators and $O\left(\frac{n}{B}\right)$ work, and (2) each SIMD comparator can be instantiated by a constant number of word-level subtraction and bitwise operations.

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

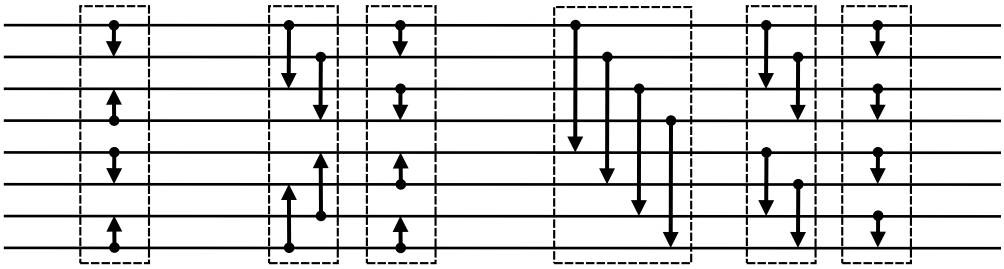

Figure 1: A bitonic sorting network for 8 inputs. Each horizontal line denotes an input from the left end and output to the right end. Each vertical arrow denotes a comparator such that compares two elements and then swaps the greater one to the pointed end. Each dashed box denotes an iteration in the algorithm. The figure is modified from [36].

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 $O(1)$ work 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 $O(1)$ work. 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 $O\left(\frac{n}{B}\right)$ work.

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. □

3.2 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 [25,26] showed how to construct *randomized* oblivious tight compaction algorithms running in time $O(n \cdot \log\log n)$. In an elegant recent work, Peserico [30] presented an $O(n)$-time, deterministic oblivious tight compaction algorithm. His result is stated in the following theorem.

Theorem 3.3 (Peserico [30])**.** *There exists a deterministic oblivious tight compaction algorithm that takes $O(n)$ work and $O(\log n)$ depth 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. [25] showed a $\Omega(n \cdot \log n)$ work 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 [25,26,30].

3.3 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 3.1) 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 $\Theta(\alpha(\lambda))$ work and $\Theta(\log \alpha(\lambda))$ depth, and using the standard AKS sort, the ORP runs in $O(\alpha(\lambda) \cdot n \cdot \log n)$ work and $O(\log \alpha(\lambda) \cdot \log n)$ depth.

Theorem 3.4. *There exists an algorithm that implements a statistically oblivious random permutation in $O(\alpha(\lambda) \cdot n \cdot \log n)$ work and $O(\log \alpha(\lambda) \cdot \log n)$ depth, 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 3.5 (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 $\ell = O(\log n \cdot \log\log \lambda)$ bits. There exists a statistically oblivious random permutation algorithm for any array of n elements that requires $O\left(\frac{n}{B} \cdot \log^2 n\right)$ work and $\mathsf{poly}\log n$ depth, 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 3.4 using the packed oblivious sort from Theorem 3.2. The only subtlety is that when $n = \mathsf{poly}\log \lambda$, random keys of length $O(\log n \cdot \log\log \lambda)$ 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 3.6: $\mathsf{ORPSmall}_{n,\lambda}(A, m, k)$ — **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 $m \leq \frac{n}{\log^3 \lambda}$, 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 3.3) 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 $O\left(\frac{n}{B} \cdot \log^2 n\right)$ work. Each iteration takes $O\left(\frac{n}{B} \cdot \log^2 n\right)$ work, the problem size shrinks by a factor of 2, and the base case takes $O(n)$ work since $m \leq \frac{n}{\log^3 \lambda}$ is a small fraction of n. The depth is $O\left(\log^2 n \cdot \log\log \lambda\right)$ as the packed oblivious sort has depth $O(\log^2 n)$, and there are $O\left(\log\log \lambda\right)$ iterations.

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 ith element is marked as colliding at Step 4. Then, it suffices to prove that for $m > n/\log^3 \lambda$, it holds that $\sum_{i=1}^{m} X_i \leq m/2$ with all but $\mathsf{negl}(\lambda)$ probability. For every set $S \subseteq [m]$, the X_i's satisfy the negative dependence such that $\Pr\left[\bigwedge_{i \in S} X_i = 1\right] \leq \mu^{|S|}$, 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 [12,28], $\Pr\left[\sum_{i=1}^{m} X_i > (1+\epsilon)m\mu\right] \leq e^{-2m(\epsilon\mu)^2}$, taking $\epsilon = \frac{1}{2\mu} - 1$, we have

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

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

As a special case, if $n = \Theta(\log^c \lambda)$, where $c \geq 5$ is a constant, and $B = O\left(\frac{\log \lambda}{\log^2 \log \lambda}\right)$, then the $\mathsf{ORPSmall}_{n,\lambda}$ requires $O(n)$ total work and $O\left(\log^3 \log \lambda\right)$ depth.

3.4 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 ith 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 3.1), where each oblivious sort operates over $O(|\mathbf{I}|)$ elements [7,9]. In fact, these works [7,9] 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.

3.5 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 3.7: $\mathcal{F}_{\mathsf{HT}}^{n,N}$ – 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 [8, 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, $\mathsf{Extract}$ can be performed in linear work if we adopt the perfect ORAM scheme by Chan et al. [8] which incurs constant space blowup. In more detail, we flatten the entire in-memory data structure into a single array, and apply oblivious tight compaction (Theorem 3.2) on the array, moving all the real elements to the front. We then truncate the array at length $|\mathbf{I}|$, apply a perfectly random permutation on the truncated array, and output the result. This gives the following construction.

Theorem 3.8 (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* $n \cdot \mathsf{poly}\log n$ *work;*
- *Each* Lookup *request consumes* $\mathsf{poly}\log n$ *work.*

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 [15, 17, 19, 23] or statistically secure [32,34,35] 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 [32].

3.6 Oblivious Cuckoo Hashing and Assignment

A Cuckoo hashing scheme [27] 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 [21]. 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 $O(n)$ work 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 [7,19,33]. We would like to *obliviously* build the Cuckoo hash table. Goodrich and Mitzenmacher [19] (see also the recent work of Chan et al. [7][5]) showed that a Cuckoo hash table containing $n \geq \log^8 \lambda$ balls can be obliviously built in $O(n \cdot \log n)$ total work and $O(\log n \cdot \log \lambda \cdot \log\log \lambda)$ depth.

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, $\mathbf{I} = ((u_1, v_1), \ldots (u_n, v_n))$ be the array of the two choices of the n balls, where $u_i, v_i \in [\mathsf{c}_{\mathsf{cuckoo}} \cdot n]$ 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 $a_i \in \{u_i, v_i, \texttt{stash}\}$ 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 3.4) that places each ball to its assigned bin or the stash.[6]

[5] Chan et al. [7] is a re-exposition and slight rectification of the elegant ideas of Goodrich and Mitzenmacher [19]; also note that the Cuckoo hashing appears only in the full version of Chan et al., http://eprint.iacr.org/2017/924.

[6] The description here slightly differs from previous works (e.g., [7]). 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

Theorem 3.9 (Obliviously Cuckoo assignment [7,19])**.** *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* cuckooAssign *such that, except with* negl(λ) *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, 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 3.1,* cuckooAssign *runs in $O(n \cdot \log n)$ work and $O(\alpha(\lambda) \cdot \log \lambda \cdot \log n)$ depth. In addition, for every input $\mathbf{I}$, the output of* cuckooAssign *is a deterministic function of $\mathbf{I}$.*

In 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 cuckooAssign in Appendix A.

Oblivious Cuckoo hashing. To obtain an oblivious hashing (Build, Lookup, Extract) as defined in Section 3.5, the construction (which includes cuckooAssign) given by Chan et al. [7] implements Build, but read-only Lookup and no 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 Extract, we obliviously shuffle all unvisited elements using oblivious random permutation (Theorem 3.4), which runs in $O(\alpha(\lambda) \cdot n \cdot \log n)$ work and $O(\log \alpha(\lambda) \cdot \log n)$ depth, where $\alpha(\lambda) = O(\log \log \lambda)$. We denote the resulting scheme cuckooHT and state its properties next.

Corollary 3.10 (cuckooHT)**.** *Let $\alpha(\lambda) = O(\log \log \lambda)$ be a fixed super-constant function, n be a number such that $\log^8 \lambda \leq n \leq$ poly(λ). Assume that one-way functions exist. Then,* cuckooHT $=$ (Build, Lookup, 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 $O(n \cdot \log n)$ work and $O(\alpha(\lambda) \cdot \log \lambda \cdot \log n)$ depth.*
- Lookup *queries the table* T *a constant number of times $O(1)$ and perfoms a linear scan of the stash* S*, which requires $O(\log \lambda)$ work.*
- Extract *requires $O(n \cdot \log^2 n)$ work and $O(\log^2 n)$ depth.*

Dummy elements. The above algorithms, including cuckooHT and cuckooAssign, naturally extend to input lists that consists of dummy elements. That is, 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 $\mathbf{I} = ((u_1, v_1), \ldots, (u_n, v_n))$) are could be dummy elements. This follows directly from the construction of cuckooAssign (see Appendix A for details).

fact that the Cuckoo assignment is only a function of $\mathbf{I}$ is crucial – see Section 3.6.1 for a discussion on this property that we call *indiscrimination*.

3.6.1 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 work 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 3.1) 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 $O(\log n)$ 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 $O\left(\frac{n}{\log \lambda} \cdot \log^3 n\right)$ work. For $n = \mathsf{poly} \log \lambda$, such $\mathsf{cuckooAssign}$ runs in $O(n)$ work. However, this does not imply that we have an oblivious Cuckoo hashing runs in $O(n)$ work. Indeed, oblivious bin placement still takes $O(n \cdot \log n)$ work 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 $\mathtt{stash}$, 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 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 $\mathsf{cuckooAssign}$ (Theorem 3.9) 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 $\mathtt{stash}$).
3. In one linear scan on $\mathbf{A}$, replace the i-th $\mathtt{stash}$ 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 3.4) 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 $\mathbf{Assign}$.
5. Output the array $\mathbf{Assign}$.

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 6 above).

Definition 3.11 (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 $\mathsf{cuckooAssign}$ is a function of the input $\mathbf{I}$ in the above abstraction.[7]

[7] This property is not guaranteed to hold in previous works [7, 19] 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.

Proposition 3.12. *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* $O(n)$ *work and satisfies the indiscriminate hashing property.*

Proof. The $O(n)$ work follows by the fact that any element in the procedure is $O(\log n)$ in bits, which implies that packed oblivious sorting (Theorem 3.2), $\mathsf{cuckooAssign}$ (Theorem 3.9, instantiated by packed oblivious sort), and oblivious bin placement (Section 3.4, instantiated by packed oblivious sort) all run in $O\left(\frac{n}{\log \lambda} \cdot \log^3 n\right)$ work, 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 $\mathsf{cuckooAssign}$ which fails with probability at most $\mathsf{negl}(\lambda)$.

To show the indiscriminate hashing property, recall that the result of $\mathsf{cuckooAssign}$ is determined by its input, the first n elements of $\widetilde{\mathbf{I}}$ (Theorem 3.9). 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, **Assign**$[i]$ is fully determined by i and $\mathbf{I}$. □

3.7 Oblivious Dictionary

As opposed to the oblivious hash table from Section 3.5, 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 predetermined capacity n. Also, the dictionary supports Lookup and $\mathsf{Extract}$ procedures as described in oblivious hashing. Note that there is no specific order in which Insert and Lookup requests have to be made and they could be mixed arbitrarily. Another difference between our hashing notion and the dictionary notion is that the $\mathsf{Extract}$ operation outputs all elements, including "visited" elements (while $\mathsf{Extract}$ of oblivious hashing outputs only "unvisited" elements). In summary, an oblivious dictionary realizes Functionality 3.13 described below.

Functionality 3.13: $\mathcal{F}^n_{\mathsf{Dict}}$ **– Dictionary Functionality**

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

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

- $\mathcal{F}^n_{\mathsf{Dict}}.\mathsf{Lookup}(k)$:
  - **Input:** The procedure receives as input a key k (that might be $\perp$, i.e., dummy).
  - **The procedure:**
    1. Initialize $v^* := \perp$.

2. If $q \in S$, set $v^* := \mathsf{T}[\mathsf{q}]$.

– **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 3.14 (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 work of each operation is $O(n \cdot \log^3 n), O(\log^4 n)$, $O(\log^4 n), O(n \cdot \log^3 n)$, 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 $O(\log n)$ work 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 [8], which incurs an $O(\log^3 n)$ work overhead factor. □

4 Interspersing Randomly Shuffled Arrays

4.1 Interspersing Two Arrays

We first describe a building block called $\mathsf{Intersperse}$ 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 $\mathsf{Intersperse}$ 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 $\mathsf{Intersperse}$ 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. In order to see that, let us count the number of options for ordering the output: A shuffling of an array of size n has $n!$ possible ordering. In order to intersperse two arrays of size n_0, n_1, we have to first randomly choose n_0 locations from the possible n, and we have $\binom{n}{n_0}$ possibilities for that. Each one of the input arrays is independent shuffled, and therefore the total number of possibilities is $\binom{n}{n_0} \cdot n_0! \cdot n_1! = n!$.

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 4.1). 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, we obliviously map each 0 (resp. 1) in Aux to a distinct element in $\mathbf{I}_0$ (resp. $\mathbf{I}_1$). For this we use the tight compaction circuit of Peserico [30] via *forth-and-back routing*. Peserico showed how to construct a tight compaction circuit for compacting n elements, consisting of only $O(n)$ word-level arithmetic and swap gates (see Section 3.2 for the precise statement). We run this compaction circuit on the array Aux, such that all the 0's will appear before the 1's. One can imagine that such a circuit is routing the 0's and 1's in Aux to an output such that: 1) every bit in Aux gets routed to a distinct position in the output, and 2) all the 0's appear before the 1's in the output. We map the resulting circuit to our input array $\mathbf{I}_0 \| \mathbf{I}_1$ by mapping the 0's to the elements of $\mathbf{I}_0$ and the 1's to the elements of $\mathbf{I}_1$. So, we need to run the circuit of Peserico *in reverse* on $\mathbf{I}$ and then the elements of $\mathbf{I}_b$ will land in the position where the bit is b.

To this end, as we run the compaction circuit of Peserico, upon each swap gate we additionally remembers its decision (simply by storing two positions of the swapped elements in the memory). Thus, we can run it in reverse by recovering the decisions of the swap gates. The result is that we route the elements in the concatenated array $\mathbf{I}$ to the corresponding input positions of Aux. That is, if $\mathsf{Aux}[i] = 0$, the i-th position receives a ball from $\mathbf{I}_0$; otherwise it receives a ball from $\mathbf{I}_1$. Furthermore, all positions receive distinct balls from $\mathbf{I}$.

The formal description of the algorithm for interspersing two arrays is given in Algorithm 4.1. The functionality that it implements (assuming that the two input arrays are randomly shuffled) is given in Functionality 4.2 and the proof that the algorithm implements the functionality is given in Claim 4.3.

Algorithm 4.1: $\mathsf{Intersperse}_n(\mathbf{I}_0 \| \mathbf{I}_1, n_0, n_1)$ – 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:** $n := \mathsf{n}_0 + \mathsf{n}_1$.
- **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 $m_0 := \mathsf{n}_0$ and $m_1 := \mathsf{n}_1$.
     (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. Let C_n be the tight compaction circuit on arrays of length n consisting of only $O(n)$ word-level arithmetic and swap gates (see Section 3.2). Simulate an evaluation of C_n on input Aux, and store all swap decisions in array $\mathbf{D}$. Note that the output array consists of exactly n_0 0's followed by n_1 1's.
  3. Run the circuit C_n in the *reverse* direction on input $\mathbf{I}_0 \| \mathbf{I}_1$, using all swap decisions stored in the array $\mathbf{D}$. Let $\mathbf{B}$ be the output array.
- **Output:** The array $\mathbf{B}$.

Functionality 4.2: $\mathcal{F}^n_{\mathsf{Shuffle}}(\mathbf{I})$ – 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 4.3. *Let $\mathbf{I}_0$ and $\mathbf{I}_1$ be two array of size n_0 and n_1, respectively, that satisfies the input assumption as in the description of Algorithm 4.1. The Algorithm $\mathsf{Shuffle}_n(\mathbf{I}_0, \mathbf{I}_1, n_0, n_1)$ obliviously implements functionality $\mathcal{F}^n_{\mathsf{Shuffle}}(\mathbf{I}_0 \| \mathbf{I}_1)$. The implementation has $O(n)$ work.*

Proof. We build a simulator that receives only $n := \mathsf{n_0} + \mathsf{n_1}$ and simulates the access pattern of Algorithm 4.1. 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.[8] 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 tight compaction circuit C_n on 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 an array for which all 0's appear before all 1's. Note that C_n is not a stable compaction circuit, so this defines some arbitrary mapping $\rho\colon [n] \to [n]$. Finally, the algorithm outputs an array $\mathbf{B}$, where $\mathbf{B}[i] = \rho^{-1}(\mathbf{I}'[i])$. 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 4.1 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

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

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

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

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

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

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

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

The implementation has $O(n)$ work since there are three main steps and each can be implemented in $O(n)$ work. Step 1 has work $O(n)$ since there are n coin flips and each can be done with $O(1)$ work (by just reading a word from the random tape). Steps 2 and 3 can be implemented in $O(n)$ work by Theorem 3.3. □

4.2 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 $\mathbf{I}_1 \| \ldots \| \mathbf{I}_k$ consisting of k different arrays of lengths $n_1, \ldots, n_k$, respectively. The parameters $n_1, \ldots, n_k$ are public.
- **Output**: An array $\mathbf{B}$ of size $\sum_{i=1}^{k} n_i$ that contains all elements of $\mathbf{I}_1, \ldots, \mathbf{I}_k$. 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 $\mathbf{I}_1, \ldots, \mathbf{I}_k$ 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 $\mathbf{I} := \mathbf{I}_1 \| \ldots \| \mathbf{I}_\mathsf{k}$ in the eyes of an adversary.

The algorithm. To intersperse k arrays $\mathbf{I}_1, \ldots, \mathbf{I}_k$, 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 4.4.

Algorithm 4.4: $\mathsf{Intersperse}^{(k)}_{n_1,\ldots,n_k}(\mathbf{I}_1 \| \ldots \| \mathbf{I}_k)$ – Shuffling an Array via Interspersing k Randomly Shuffled Subarrays

- **Input:** An array $\mathbf{I} := \mathbf{I}_1 \| \ldots \| \mathbf{I}_\mathsf{k}$ consisting of k arrays of sizes $n_1, \ldots, n_k$, respectively.
- **Public parameters:** $n_1, \ldots, n_k$.

- **Input assumption:** Each input array is independently randomly shuffled.
- **The algorithm:**
  1. Let $\mathbf{I}'_1 := \mathbf{I}_1$.
  2. For $i = 2, \ldots k$, 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 $\mathbf{B} := \mathbf{I}'_{\mathsf{k}}$.
- **Output:** The array $\mathbf{B}$.

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

Claim 4.5. *Let $k \in \mathbb{N}$ and let $\mathbf{I}_1, \ldots, \mathbf{I}_k$ be k arrays of $n_1, \ldots, n_k$ elements, respectively, that satisfy the input assumption as in the description of Algorithm 4.4. The Algorithm $\mathsf{Intersperse}^{(k)}_{n_1,\ldots,n_k}(\mathbf{I}_1 \| \ldots \| \mathbf{I}_k)$ obliviously implements the functionality $\mathcal{F}^{n}_{\mathsf{Shuffle}}(\mathbf{I})$. The implementation requires $O\left(\sum_{i=1}^{k-1}(k-i) \cdot n_i\right)$ work.*

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

Since the work of $\mathsf{Intersperse}_n$ is linear in n, the work required in the ith 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 work is $O\left(\sum_{i=1}^{k-1}(k-i) \cdot n_i\right)$, as required. □

4.3 Interspersing Reals and Dummies

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

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

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

The implementation of $\mathsf{IntersperseRD}$ is done by first running the deterministic tight compaction procedure on the input array such that all the real balls appear before the dummy ones. Next, we

count the number of real elements in this array run the $\mathsf{Intersperse}$ procedure from Algorithm 4.1 on this array with the calculated sizes. The formal implementation appears as Algorithm 4.6.

Algorithm 4.6: $\mathsf{IntersperseRD}_n(\mathbf{I})$ – 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 3), 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 4.7. *Let $\mathbf{I}$ be an array of n elements that satisfies the input assumption as in the description of Algorithm 4.6. The Algorithm $\mathsf{IntersperseRD}_n(\mathbf{I})$ obliviously implements the functionality $\mathcal{F}^n_{\mathsf{Shuffle}}(\mathbf{I})$. The implementation has $O(n)$ work.*

Proof. We build a simulator that receives only the size of $\mathbf{I}$ and simulates the access pattern of Algorithm 4.6. 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 4.3.

The proof that the output distribution of algorithm is identical to that of the ideal functionality follows immediately from Claim 4.3. 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 4.1) holds and thus the output is guaranteed to be randomly shuffled (by Claim 4.3).

The implementation has $O(n)$ work since the first two steps take $O(n)$ work (by Theorem 3.3) and $\mathsf{Intersperse}_n$ itself has $O(n)$ work (by Claim 4.3). □

5 BigHT: Oblivious Hashing for Non-Recurrent Lookups

Our final ORAM construction will be based on the hierarchical ORAM of Goldreich and Ostrovsky [17]. A core building block in that construction (to implement different "levels") is an oblivious hashing scheme for non-recurrent requests [7, 17].

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. [29] and can be used to get an ORAM with similar overhead to that of Patel et al. In

the next Section 6, we describe how to optimize the hash table and get rid of the extra $\mathsf{poly}\log\log$ factor.

As defined in Section 3.5, 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.[9] 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 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 3.7, where the input to Build is assumed to be *randomly shuffled*. In other words, as long as the input array $\mathbf{I}$ 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 $\mathbf{I}$ elements which include the unvisited elements and they are completely shuffled within (with hidden locations and order).

5.1 Intuition and Overview of the Construction

Let $\mathbf{I}$ 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 $\mathbf{I}$ 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 $\mathbf{I}$ 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 3.5). For $\mathsf{poly}\log(\lambda)$-sized bins, each Lookup request consumes $\mathsf{poly}\log\log\lambda$ work. In Section 6, 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 $\mathbf{I}$ provided to Build has been secretly randomly shuffled. Due to

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

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 $\mathbf{I}$ 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.[10] To illustrate the point, consider a special case where all elements of the input array $\mathbf{I}$ 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 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*.

[10] Formally, this scheme does not satisfy Definition 2.2. While each operation is simulatable by itself, there are sequences of operations that leak non-trivial information.

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 3.3) so that the real elements appear before the dummies. Then, we employ a standard oblivious Cuckoo hashing scheme (Corollary 3.10). 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 5.3 (after we present the full details of the construction in Section 5.2) and the proof there will additionally take into account the possibility of dummy items in the input array.

5.2 The Full Construction

In Construction 5.2 we give the implementation of Functionality 3.7. As a subroutine, we use a procedure (see Algorithm 5.1) to sample bin loads of a random balls-into-bins process.

Algorithm 5.1: 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 $m := \mathsf{n}$. 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 $m := \mathsf{m} - \mathsf{L_i}$.
- **Output:** $L_1, \ldots, L_B$.

Construction 5.2: 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 $B := \mathsf{n}/\mu$.
  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 5.1 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 3.10). Let $\mathsf{OF} = (\mathsf{OF}_{\mathsf{T}}, \mathsf{OF}_{\mathsf{S}})$ 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 $\mathsf{BigHT.Lookup}(k)$:

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

Procedure BigHT.Extract():

- **Input:** The secret state $(\mathsf{OBin}_1, \ldots, \mathsf{OBin}_B, \mathsf{OF}, \mathsf{sk})$.
- **The algorithm:**
  1. Let $T = \mathsf{OBin}_1.\mathsf{Extract}() \| \mathsf{OBin}_2.\mathsf{Extract}() \| \ldots \| \mathsf{OBin}_B.\mathsf{Extract}() \| \mathsf{OF}.\mathsf{Extract}()$.
  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 4.6.
- **Output**: $\mathbf{X}'$.

5.3 Efficiency and Security Analysis

We prove that our construction obliviously implements Functionality 3.7 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 3.5), an oblivious Cuckoo hashing scheme (Section 3.6) and an oblivious tight compaction algorithm (Section 3.2).

Theorem 5.3. *Construction 5.2 obliviously implement Functionality 3.7 for all $n \geq \log^{12} \lambda$, assuming that the input array $\mathbf{I}$ for* Build *is randomly shuffled, and assuming one-way function. Moreover, the construction consumes $O(n \cdot \mathsf{poly} \log\log \lambda)$ work for the* Build *phase and* Extract *phases.* Lookup *has $O(\mathsf{poly} \log\log \lambda)$ work in addition to linearly scanning a stash of size $O(\log \lambda)$.*

Remark 5.4. *As we mentioned, Construction 5.3 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 6. 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 5.2, 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 3.5) for each major bin, and thus their initialization incurs

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

work. The overflow pile is implemented via an oblivious Cuckoo hashing scheme (Corollary 3.10) so its initialization incurs $O(n)$ work. (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 $O(\mathsf{poly} \log\log \lambda)$ work from the major bins and $O(\log \lambda)$ work from the linear scan of OF_S the stash of the overflow pile (searching in OF_T incurs $O(1)$ work). The overhead of Extract is the same as that of Build.

In total, our oblivious hashing scheme consumes $O(n \cdot \mathsf{poly} \log\log \lambda)$ work for the Build phase and Extract phases. Lookup has $O(\mathsf{poly} \log\log \lambda)$ work 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 $\mathsf{Extract}$ 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 5.5. *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 5.6. $\Pr[\mathsf{Hyb}_1(\lambda) = 1] = \Pr[\mathsf{Hyb}_2(\lambda) = 1]$.

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 Extract. We first argue that the output of $\mathcal{F}_{\mathsf{HT}}^{n,N}$.Extract consists of the same elements as that of Extract. Indeed, both $\mathcal{F}_{\mathsf{HT}}^{n,N}$.Lookup and Lookup mark every visited item so when we execute Extract, the same set of elements will be in the output.

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

Consider tuples of the form $(\pi_{\mathsf{in}}, \mathcal{O}, R, \mathsf{T}, \pi_{\mathsf{out}})$, where (1) π_{in} is the permutation performed on $\mathbf{I}$ by the input assumption (prior to Build), (2) $\mathcal{O}$ is the random oracle, (3) R is the internal randomness of all intermediate functionalities and of the balls into bins choices of the dummy elements; (4) T is the access pattern of the entire sequence of commands $(\mathsf{Build}(\mathbf{I}), \mathsf{Lookup}(k_1^*), \ldots, \mathsf{Lookup}(k_m^*))$, and (5) π_{out} is the permutation on $\mathbf{I}' = \{(k, v) \in \mathbf{I} \mid k \notin \{k_1^*, \ldots, k_m^*\}\}$ which is the input to Extract. The algorithm defines a deterministic mapping $\psi_R(\pi_{\mathsf{in}}, \mathcal{O}) \to (\mathsf{T}, \pi_{\mathsf{out}})$.

To gain intuition, consider arbitrary R, π_{in}, and $\mathcal{O}$ such that $\psi_R(\pi_{\mathsf{in}}, \mathcal{O}) \to (\mathsf{T}, \pi_{\mathsf{out}})$ and two distinct existing keys k_i and k_j that are not queried during the Lookup stage (i.e., $k_i, k_j \notin \{k_1^*, \ldots, k_m^*\}$). We argue that from the point of view of the adversary, having seen the access pattern and all query results, he cannot distinguish whether $\pi_{\mathsf{out}}(i) < \pi_{\mathsf{out}}(j)$ or $\pi_{\mathsf{out}}(i) > \pi_{\mathsf{out}}(j)$. The argument will naturally generalize to arbitrary unqueried keys and an arbitrary ordering.

To this end, we show that there is π'_{in} and $\mathcal{O}'$ such that $\psi_R(\pi'_{\mathsf{in}}, \mathcal{O}') \to (\mathsf{T}, \pi'_{\mathsf{out}})$, where $\pi'_{\mathsf{out}}(\ell) = \pi_{\mathsf{out}}(\ell)$ for every $\ell \notin \{i, j\}$, and $\pi'_{\mathsf{out}}(i) = \pi_{\mathsf{out}}(j)$ and $\pi'_{\mathsf{out}}(j) = \pi_{\mathsf{out}}(i)$. That is, the access pattern is exactly the same and the output permutation switches the mappings of k_i and k_j. The permutation π'_{in} is the same as π_{in} except that $\pi'_{\mathsf{in}}(i) = \pi_{\mathsf{in}}(j)$ and $\pi'_{\mathsf{in}}(j) = \pi_{\mathsf{in}}(i)$, and $\mathcal{O}'$ is the same as $\mathcal{O}$ except that $\mathcal{O}'(k_i) = \mathcal{O}(k_j)$ and $\mathcal{O}'(k_j) = \mathcal{O}(k_i)$. This definition of π'_{in} together with $\mathcal{O}'$ ensure, by our construction, that the observed access pattern remains exactly the same. The mapping is also reversible so by symmetry all permutations have the same number of configurations of π_{in} and $\mathcal{O}$.

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

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

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

□

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

Recall that in this hybrid experiment the output of 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 5.7. $\Pr[\mathsf{Hyb}_2(\lambda) = 1] = \Pr[\mathsf{Hyb}_3(\lambda) = 1]$.

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 5.8. *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. 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. 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 bind, 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

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

Thus, following a fact on statistical distance,[11]

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

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 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 5.8. □

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

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

By combining Claims 5.5–5.10 we conclude the proof of Theorem 5.3.

[11] 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.

6 SmallHT: Oblivious Hashing for Small Bins

Recall that the warmup oblivious hashing scheme in Section 5 consumes $O(n \cdot \mathsf{poly}\log\log\lambda)$ work for the Build phase and Extract phase, and $O(\mathsf{poly}\log\log\lambda)$ work 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 polylogarithmic size. At a high level, the idea is that each major bin will no longer be implemented using a naïveHT (Section 3.5), but rather as an efficient oblivious hashing scheme, where Build and Extract consume linear amount of work and each Lookup consumes constant work (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 3.7.

6.1 Intuition and Overview of the Construction

We refer to Section 3.6 for background information on Cuckoo hashing. Here, we recall that, by [7, 19], a Cuckoo hash table containing $n \geq \log^8\lambda$ balls can be obliviously constructed in $O(n \cdot \log n)$ total work. For $n = \mathsf{poly}\log\lambda$, the total work would be $O(n \cdot \log\log\lambda)$ and we would like to get rid of the extra $\log\log\lambda$ factor.

Remark 6.1. *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 3, 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 $O\left(\frac{n}{B} \cdot \mathsf{poly}\log n\right)$ work. In particular, when $n = \mathsf{poly}\log\lambda$, packed oblivious sorting and oblivious random permutation can be accomplished in $O(n)$ work.

Our algorithm, called 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 6.2.1, we describe how to achieve this in linear total work 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 6.2.2, 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 work. 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 work, 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 3.3) 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 $\mathsf{IntersperseRD}$ 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 $\mathsf{Extract}$ 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 $\mathsf{Extract}$, 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 3.6, we showed how to slightly modify the Cuckoo hashing bin assignment procedure of [7, 19] to achieve this property, and we formally restated this property in Proposition 3.12.

We formalize the above intuition and present a detailed proof in Section 6.3, after we present the full details of the construction in Section 6.2.

6.2 The Full Construction

In this section we give the full description of our hash table, following the high-level overview given in Section 6.1. We start with formally describing the first two steps from the overview (Steps 1 and 2) in Sections 6.2.1 and 6.2.2, respectively. Then, in Section 6.2.3 (Construction 6.7), we give the full description of the Build, Lookup, and $\mathsf{Extract}$ procedures.

6.2.1 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 6.2) 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 6.2: 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 $(\perp, 1), (\perp, 2), \ldots, (\perp, n_{\mathsf{cuckoo}} - n_R)$.

3. Run packed oblivious random permutation (Theorem 3.5) on $\mathbf{MD}$, packing $O\left(\frac{\log \lambda}{\log n}\right)$ elements into a single memory word. Run oblivious tight compaction (Theorem 3.3) on the resulting array, moving all the dummy elements to the end.
4. Run tight compaction (Theorem 3.3) 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 4.6) mixing the reals and dummies at random. Let $\mathbf{X}$ denote the outcome array.

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

Claim 6.3. *If $n = \log^{12} \lambda$, then Algorithm 6.2 has $O(n)$ total work.*

Proof. All steps but 3 incur $O(n)$ work 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 $O\left(\frac{n}{\log \lambda / \log \log \lambda} \cdot \log^2 n\right) \leq O(n)$ work. □

6.2.2 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 3.2). At the end of this step, every element in the input array should learn which bin (either in the main table or the stash) it is destined for. Recall that the Cuckoo hashing consists of a main table of $\mathsf{c}_{\mathsf{cuckoo}} \cdot n$ bins and a stash of $\log \lambda$ bins.

Our input for this step is an array $\mathbf{MD}_{\mathbf{X}}$ of length n_{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 $\mathbf{MD}_{\mathbf{X}}$. Next, we obliviously assign the remaining dummy elements to the remaining available locations. We do so by a sequence of oblivious sort algorithms. See Algorithm 6.4.

Algorithm 6.4: Evaluate Cuckoo Hash Assignment on Metadata

- **Input:** An array $\mathbf{MD}_{\mathbf{X}}$ of length $n_{\mathsf{cuckoo}} = \mathsf{c}_{\mathsf{cuckoo}} \cdot n + \log \lambda$, where each element is either dummy or a pair $(\mathsf{choice}_{i,1}, \mathsf{choice}_{i,2})$, where $\mathsf{choice}_{i,b} \in [\mathsf{c}_{\mathsf{cuckoo}} \cdot n]$ for every $b \in \{1, 2\}$, and the number of real pairs is at most n.
- **The algorithm:**
  1. Run the indiscriminate oblivious Cuckoo assignment, $\overline{\mathsf{cuckooAssign}}$ (Proposition 3.12), and let $\mathbf{Assign}_{\mathbf{X}}$ be the result. For every i for which $\mathbf{MD}_{\mathbf{X}}[i] = (\mathsf{choice}_{i,1}, \mathsf{choice}_{i,2})$, we have that $\mathbf{Assign}_{\mathbf{X}}[i] = \mathsf{assignment}_i$ 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 $\mathbf{MD}_{\mathbf{X}}[i]$ is dummy we have that $\mathbf{Assign}_{\mathbf{X}}[i] = \perp$.

2. Run oblivious bin assignment (Section 3.4) on $\mathbf{Assign}_{\mathbf{X}}$, and let **Occupied** be the output array (of length n_{cuckoo}). For every index j we have $\mathbf{Occupied}[j] = i$ if $\mathbf{Assign}_{\mathbf{X}}[i] = j$ for some i. Otherwise, $\mathbf{Occupied}[j] = \perp$.
3. Label the i-th element in $\mathbf{Assign}_{\mathbf{X}}[i]$ with a tag $t = i$ for all i. Run oblivious sorting on $\mathbf{Assign}_{\mathbf{X}}$ 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., the tag t).
4. Label the i-th element in **Occupied** with a tag $t = i$ for all i. Run oblivious sorting on **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 **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 $\mathbf{Assign}_{\mathbf{X}}$) according to the tag labeled in Step 3. Update the assignments of all dummy elements in $\mathbf{Assign}_{\mathbf{X}}$ according to the output array of this step.

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

Claim 6.5. *If $n = \log^8 \lambda$, then Algorithm 6.4 has $O(n)$ total work.*

Proof. The input arrays is of size $\mathsf{c}_{\mathsf{cuckoo}} \cdot n + \log \lambda$ and each entry is of size $O(\log n)$. By Proposition 3.12, Step 1 runs in $O(n)$ work. Steps 2 through 6 consist of a constant number of packed oblivious sorting (Theorem 3.2) invocations so they consume altogether $O(n)$ work. □

Remark 6.6 (On the indiscriminate property)**.** *This property of indiscriminate hashing (Definition 3.11) is necessary in our construction of* SmallHT *(Theorem 6.8), but was not needed in previous works that use oblivious Cuckoo hashing [7, 19, 29]. This is since we reveal $\mathbf{Assign}_{\mathbf{X}}$ 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 $\mathbf{Assign}_{\mathbf{X}}$ 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.*

6.2.3 Combining it All Together

The full description of the construction is given next. It invokes Algorithms 6.2 and 6.4.

Construction 6.7: 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 6.2 (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 $\{0,1\}^{\log N} \to [\mathsf{c}_{\mathsf{cuckoo}} \cdot n]$.
  3. Create a new metadata array $\mathbf{MD}_{\mathbf{X}}$ of length n. Iterate over the the array $\mathbf{X}$ and for each real element $\mathbf{X}[i] = (k_i, v_i)$ compute two values $(\mathsf{choice}_{i,1}, \mathsf{choice}_{i,2}) \leftarrow \mathsf{PRF}_{\mathsf{sk}}(k_i)$, and write $(\mathsf{choice}_{i,1}, \mathsf{choice}_{i,2})$ in the i-th location of $\mathbf{MD}_{\mathbf{X}}$. If $\mathbf{X}[i]$ is dummy, write $(\perp, \perp)$ in the i-th location of $\mathbf{MD}_{\mathbf{X}}$.
  4. Run Algorithm 6.4 on $\mathbf{MD}_{\mathbf{X}}$ to compute the assignment for every element in $\mathbf{X}$. The output of this algorithm, denoted $\mathbf{Assign}_{\mathbf{X}}$, is an array of length n, where in the ith position we have the destination location of element $\mathbf{X}[i]$.
  5. Route the elements of $\mathbf{X}$, in the clear, according to $\mathbf{Assign}_{\mathbf{X}}$, into an array $\mathbf{Y}$ of size $\mathsf{c}_{\mathsf{cuckoo}} \cdot n$ 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 3.3) 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 4.6).
- **Output:** The array $\mathbf{X}'$.

6.3 Efficiency and Security Analysis

We prove that our construction obliviously implements Functionality 3.7 for every sequence of instructions with non-recurrent lookups between two Build operations. As in Section 5, we view

our construction in a hybrid model, in which we have ideal implementations of the underlying building blocks.

Theorem 6.8. *Construction 6.7 obliviously implement Functionality 3.7, 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* $O(n)$ *work,* Lookup *has constant work in addition to linearly scanning a stash of size* $O(\log \lambda)$, *and* Extract *incurs* $O(n)$ *work.*

We start with the efficiency analysis. The Build operation executes Algorithm 6.2 that consumes $O(n)$ work (by Claim 6.3), then performs additional $O(n)$ work, then executes Algorithm 6.4 that consumes $O(n)$ work (by Claim 6.5), and finally performs additional $O(n)$ work. Thus, the total work is $O(n)$. For Lookup, by construction, we perform constant work in addition to lookup in the stash S which is of size $O(\log \lambda)$. The cost of Extract is $O(n)$ work by construction.

For security, we present a simulator Sim that simulates Build, Lookup and Extract procedures of 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 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 SmallHT.Lookup on input $\perp$ (i.e., a dummy element) with the state $(\mathbf{Y}, \mathsf{S}, \mathsf{sk})$ (which was generated while simulating the the Build operation).
- **Simulating Extract.** When the adversary submits an Extract command, the simulator Sim executes the real 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 6.9. *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 6.10. $\Pr[\mathsf{Hyb}_1(\lambda) = 1] = \Pr[\mathsf{Hyb}_2(\lambda) = 1]$.

Proof. Recall that we assume that the lookup queries of the adversary are non-recurring. Our goal is to show that the output distribution of the extract procedure is a uniform permutation of the unvisited items even given the access patter of the previous Build and Lookup operations. By doing so, we can replace the $\mathsf{Extract}$ 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 $\mathsf{SmallHT}.\mathsf{Lookup}$ 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 6.2 and 6.4, 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., $\mathbf{MD}_{\mathbf{X}}$), 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 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 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 6.11. $\Pr[\mathsf{Hyb}_2(\lambda) = 1] = \Pr[\mathsf{Hyb}_3(\lambda) = 1]$.

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

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

Claim 6.13. $\Pr[\mathsf{Hyb}_4(\lambda) = 1] = \Pr[\mathsf{Hyb}_5(\lambda) = 1]$.

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

By combining Claims 6.9–6.14 we conclude the proof of Theorem 6.8.

6.4 CombHT: Combining BigHT with SmallHT

We use SmallHT in place of naïveHT for each of the major bins in the BigHT construction from Section 5. 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 5.3).

However, we make one additional modification that will be useful for us later in the construction of the ORAM scheme (Section 7). 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 3.6). (A similar idea has also been applied in several prior works [9, 19, 20, 23].) 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 work.

Construction 6.15: CombHT: combining BigHT with SmallHT

Procedure CombHT.Build(I): Run Steps 1–7 of Procedure BigHT.Build in Construction 5.2, 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 3.6), 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 Lookup(k_i): The procedure is the same as in Construction 5.2, 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 Extract(). The procedure Extract is the same as in Construction 5.2, except that $T=\mathsf{OBin}_1.\mathsf{Extract}()\|\ldots\|\mathsf{OBin}_B.\mathsf{Extract}()\|\mathsf{OF.Extract}()\|\mathsf{CombS.Extract}()$.

Theorem 6.16. *Construction 6.15 obliviously implement Functionality 3.7, assuming that the input for* CombHT.Build *is randomly shuffled, and assuming one-way functions. Assuming that* $|\mathbf{I}|=n$ *and* $n\geq\log^{12}\lambda$, *procedures* CombHT.Build($\mathbf{I}$) *and* CombHT.Extract *perform* $O(n)$ *work, and* CombHT.Lookup *has constant work 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 $O(|\mathbf{I}|)$ work. 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 work 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 3.7.

- **Construction I:** This construction is the same as Construction 5.2, 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 [7] 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 6.15.

As follows from Theorems 5.3 and 6.8, Construction I obliviously implements Functionality 3.7 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 3.7.

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 3.7. 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 3.7. The obliviousness of Construction IV stems directly from the obliviousness of the oblivious Cuckoo hash. We conclude that Construction II obliviously implements Functionality 3.7, as well. □

7 Oblivious RAM

In this section we present our final ORAM construction.

7.1 Overview and Intuition

Recall that Sections 5 and 6 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 [17] 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 (addr, 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) $O(\log N)$ work overhead per access. In the hierarchical ORAM framework, if an oblivious hash table can perform lookup with $O(1)$ work, 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 $O(1)$ work 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 $O(1)$ work lookup and linear work build. Specifically, the known candidate $O(1)$ constructions, oblivious Cuckoo hashing and two-tier hashing, do not seem to be secure in this range [7]. 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 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 3.14), 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 $O(\log \lambda)$ work, which exceeds our budget. We address this by using the same technique as in Kushilevitz et al. [23]. 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. [29]. 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 work $\mathsf{Intersperse}$ procedure (Section 4) to create a single randomly-shuffled array of blocks.

7.2 The Construction

ORAM Initialization. Let N be the memory size. Our structure consists of one dictionary D (see Section 3.7), 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 $\mathsf{data} \in \{0,1\}^\beta$.
- Each level $i \in \{\ell + 1, \ldots, L\}$ consists of an instance, called T_i, of the oblivious hash table CombHT from Section 6.4 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 7.1: Oblivious RAM $\mathsf{Access}(\mathsf{op}, \mathsf{addr}, \mathsf{data})$**.**

- **Input:** $\mathsf{op} \in \{\mathsf{read}, \mathsf{write}\}$, $\mathsf{addr} \in [N]$ and $\mathsf{data} \in \{0,1\}^\beta$.
- **Secret state:** The dictionary D, levels $T_{\ell+1}, \ldots, T_L$, the bits $\mathsf{full}_{\ell+1}, \ldots, \mathsf{full}_L$ and counter ctr.
- **The algorithm:**
  1. Initialize $\mathsf{found} := \mathsf{false}$, $\mathsf{data}^* := \perp$, $\mathsf{levelIndex} := \perp$ and $\mathsf{whichStash} := \perp$.
  2. Perform $\mathsf{fetched} := \mathsf{D.Lookup}(\mathsf{addr})$. If $\mathsf{fetched} \neq \perp$:
     (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 $i \in \{\ell + 1, \ldots, L\}$ in increasing order, do:
     (a) If $\mathsf{found} = \mathsf{false}$:
        i. Run $\mathsf{fetched} := \mathsf{T_i.Lookup}(\mathsf{addr})$ with the following modifications:
           – Instead of visiting the stash of OF, namely $\mathsf{OF_S}$, in Construction 6.15, 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 $\perp$.
           – 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 $\perp$.
        ii. If $\mathsf{fetched} \neq \perp$, let $\mathsf{found} := \mathsf{true}$ and $\mathsf{data}^* := \mathsf{fetched}$.
     (b) Else, $T_i.\mathsf{Lookup}(\perp)$.

4. Let $(k, v) := \{(\mathsf{addr}, \mathsf{data}^*)\}$ if this is a read operation; else let $(k, v) := \{(\mathsf{addr}, \mathsf{data})\}$. Insert $(k, (\ell, \perp, v))$ into oblivious dictionary D using $D.\mathsf{Insert}(k, (\ell, \perp, v))$.
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 $j := \mathsf{L}$. In other words, j is the target level to be rebuilt.
   (b) Let $\mathbf{U} := \mathsf{D.Extract}() \| \mathsf{T}_{\ell+1}.\mathsf{Extract}() \| \ldots \| \mathsf{T}_{\mathsf{j}-1}.\mathsf{Extract}()$ and set $j^* := \mathsf{j} - 1$. If all levels are marked full, then additionally let $\mathbf{U} := \mathbf{U} \| \mathsf{T}_{\mathsf{L}}.\mathsf{Extract}()$ and set $j^* := \mathsf{L}$.
   (c) Run $\mathsf{Intersperse}^{(j^*-\ell)}_{2^{\ell+1}, 2^{\ell+1}, 2^{\ell+2}, \ldots, 2^{j^*}}(\mathbf{U})$ (Algorithm 4.4). 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 $\tilde{\mathbf{U}} \leftarrow \mathsf{InterspersRD}_N(\tilde{\mathbf{U}})$ (Algorithm 4.6).
   (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 6.15) and let $\mathsf{OF}_\mathsf{S}, \mathsf{CombS}_\mathsf{S}$ be the associated stashes (of size $O(\log \lambda)$ each). Mark $\mathsf{full}_j := 1$.
      i. For each element (k, v) in the stash OF_S, run $D.\mathsf{Insert}(k, (j, \mathsf{overflow}, v))$.
      ii. For each element (k, v) in the stash $\mathsf{CombS}_\mathsf{S}$, 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^*.

7.3 Efficiency and Security Analysis

We prove that our construction obliviously implements the ORAM functionality (Functionality 2.3) and its amortized overhead is logarithmic.

Theorem 7.2. *Construction 7.1 obliviously implements Functionality 2.3. Moreover, the construction performs $O(\log N)$ amortized work.*

Proof. We start with the analysis of work. 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 $O(\log^4 \log \lambda) + O(\log N)$ work. 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 $O\left(2^\ell \cdot \log^3(2^{\ell+1}) + \log^2 N \cdot \log^4(2^{\ell+1}) + 2^{\ell+1}\right) = O(2^\ell \cdot \log^4 \log \lambda)$ work. 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 $\mathsf{InterspersRD}_N$, and one tight compaction, all of which require linear work overhead and thus the total work overhead is $O(2^j)$. Hence, over t requests, the amortized work 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 2.3.

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 7.1. 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 $\mathcal{F}_{\mathsf{Dict}}^{2^\ell}$, handles $T_{\ell+1}, \ldots, T_L$ to $\mathcal{F}_{\mathsf{HT}}{}^{2^{\ell+1},N}, \ldots, \mathcal{F}_{\mathsf{HT}}{}^{2^L,N}$, respectively, a counter ctr and flags $\mathsf{full}_{\ell+1}, \ldots, \mathsf{full}_L$. Upon receiving a command $\mathsf{Access}(\mathsf{op}, \mathsf{addr}, \mathsf{data})$:

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

Claim 7.3. *Construction 1 obliviously implements Functionality 2.3.*

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 $T_{\ell+1}, \ldots, T_L$, bits $\mathsf{full}_{\ell+1}, \ldots, \mathsf{full}_L$ and counter ctr exactly as the real construction. Upon receiving a command $\mathsf{Access}(\perp, \perp, \perp)$, the simulator runs Construction 1 on input $(\perp, \perp, \perp)$.

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

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 2.3, we proceed with a sequence of constructions and show that each and one of them obliviously implements Functionality 2.3 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 4.4), instantiate $\mathcal{F}_{\mathsf{Shuffle}}$ in Step 5(c)ii with IntersperseRD (Algorithm 4.6, and instantiate $\mathcal{F}_{\mathsf{compaction}}$ in Step 5(c)i with an algorithm for tight compaction (Theorem 3.3). 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 6.15 (i.e., CombHT from Theorem 6.16). 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 Extract()) in Construction 1 with the implementation of Construction 6.15 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 7.1: We add all elements in OF_{S} and $\mathsf{CombS}_{\mathsf{S}}$ into D, marked with their level index and what stash they are coming from (where whichStash = overflow in case that the element comes from OF_{S}, and whichStash = stashes in case that the element comes from $\mathsf{CombS}_{\mathsf{S}}$).
- **Construction 5.** In this construction, we follow Construction 4, but make the following change: In Step 3, we modify the Lookup procedure of Construction 6.15, and whenever accessing OF_{S} and $\mathsf{CombS}_{\mathsf{S}}$, we perform lookup at the stored values levelIndex and 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 7.1.

The theorem is obtained using a sequence of simple claims, given that Construction 1 obliviously implements Functionality 2.3.

Claim 7.4. *Construction 2 obliviously implements Functionality 2.3.*

Proof. This follows by composing Claims 4.7,4.5 and Theorem 3.3. 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 4.4.
- 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 4.6.

□

Claim 7.5. *Construction 3 obliviously implements Functionality 2.3*

Proof. This follows from Theorem 6.16 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 7.6. *Construction 4 obliviously implements Functionality 2.3.*

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 2.3. □

Claim 7.7. *Construction 5 obliviously implements Functionality 2.3.*

Proof. The construction is just as Construction 4, where instead of searching in each level for the elements in the stashes OF_{S} and $\mathsf{CombS}_{\mathsf{S}}$, 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 7.8. *Construction 6 obliviously implements Functionality 2.3*

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 7.2. □

Remark 7.9 (Using More CPU Registers)**.** *Our construction can be slightly modified to obtain optimal amortized work overhead (up to constant factors) for* any number of CPU registers*, as given by the lower bound of Larsen and Nielsen [24]. Specifically, if the number of CPU registers is m, then we can achieve a scheme with $O(\log(N/m))$ amortized work overhead.*

If $m \in N^{1-\epsilon}$ for $\epsilon > 0$, then the lower bound still says that $\Omega(\log N)$ amortized work overhead is required so we can use Construction 7.1 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 7.1 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 $O(\log N - \log m) = \log(N/m)$ which is the significant factor in the overhead analysis (as the amortized work 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 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. ACM, 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. ACM, 1983.

[3] 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.

[4] 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. ACM, 2016.

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

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

[7] 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.

[8] T.-H. Hubert Chan, Kartik Nayak, and Elaine Shi. Perfectly secure oblivious parallel RAM. *IACR Cryptology ePrint Archive*, 2018:364, 2018. To appear in TCC 2018.

[9] 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.

[10] 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.

[11] Ivan Damgård, Sigurd Meldgaard, and Jesper Buus Nielsen. Perfectly secure oblivious RAM without random oracles. In *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] 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.

[15] 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. ACM, 1987.

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

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

[18] Michael T. Goodrich and Michael Mitzenmacher. Privacy-preserving access of outsourced data via oblivious RAM simulation. In Luca Aceto, Monika Henzinger, and Jirí Sgall, editors, *Automata, Languages and Programming - 38th International Colloquium, ICALP*, pages 576–587, 2011.

[19] 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.

[20] 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. SIAM, 2012.

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

[22] 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. SIAM, 2012.

[23] 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. SIAM, 2012.

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

[25] 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.

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

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

[28] 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, Vancouver, British Columbia, Canada, August 10-12, 1992*, pages 251–262. ACM, 1992.

[29] Sarvar Patel, Giuseppe Persiano, Mariana Raykova, and Kevin Yeo. Panorama: Oblivious RAM with logarithmic overhead. *IACR Cryptology ePrint Archive*, 2018:373, 2018. To appear in FOCS 2018.

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

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

[32] 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.

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

[34] 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. ACM, 2013.

[35] 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. ACM, 2015.

[36] 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.[12] 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.

[12] 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. [7], 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 3.6.1.