OptORAMa: Optimal Oblivious RAM

Gilad Asharov[1], Ilan Komargodski[2], Wei-Kai Lin[3], Kartik Nayak[4], Enoch Peserico[5], and Elaine Shi[3]

[1] Bar-Ilan University, Ramat Gan 52900, Israel
gilad.asharov@biu.ac.il
[2] NTT Research, Palo Alto, CA 94303, USA
ilan.komargodski@ntt-research.ac.il
[3] Cornell University, Ithaca, NY 14850, USA
w1572@cornell.edu
runting@gmail.com
[4] Duke University, Durham, NC 27708, USA
kartik@cs.duke.edu
[5] Università degli Studi di Padova, Padova (PD), Italy
enoch@dei.unipd.it

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

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

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

Keywords: Oblivious RAM, randomized algorithms, tight compaction.

1 Introduction

Oblivious RAM (ORAM), first proposed by Goldreich and Ostrovsky [22, 23], is a technique to compile *any* program into a functionally equivalent one, but whose memory access patterns are independent of the program's secret inputs. The overhead of an ORAM is defined as the (multiplicative) blowup in runtime of the compiled program. Since Goldreich and Ostrovsky's seminal work, ORAM has received much attention due to its applications in cloud computing, secure processor design, multi-party computation, and theoretical cryptography (for example, [7, 19–21, 34–36, 39, 45, 47, 48, 52–54])

For more than three decades, the biggest open question in this line of work is regarding the *optimal* overhead of ORAM. Goldreich and Ostrovsky's original work [22,23] showed a construction with $O(\log^3 N)$ blowup in runtime, assuming the existence of one-way functions, where N denotes the memory size consumed by the original non-oblivious program. On the other hand, they proved that any ORAM scheme must incur at least $\Omega(\log N)$ overhead, but their lower bound is restricted to schemes that treat the contents of each memory word as "indivisible" (see Boyle and Naor [8]) and make no cryptographic assumptions. In a recent work, Larsen and Nielsen [30] showed that $\Omega(\log N)$ overhead is necessary for all *online* ORAM schemes,[6] even ones that use cryptographic assumptions and might perform non-trivial encodings on the contents of the memory. Since Goldreich and Ostrovsky's work, a long line of research has been dedicated to improving the asymptotic efficiency of ORAM [10, 25, 29, 46, 49, 51]. Prior to our work, the best known scheme, allowing computational assumptions, is the elegant work by Patel et al. [40]: they showed the existence of an ORAM with $O(\log N \cdot \log\log N)$ overhead, assuming one-way functions. In comparison with Goldreich and Ostrovksy's original $O(\log^3 N)$ result, Patel's result seems tantalizingly close to matching the lower bound, but unfortunately we are still not there yet and the construction of an optimal ORAM continues to elude us even after more than 30 years.

1.1 Our Results: Optimal Oblivious RAM

We resolve this long-standing problem by showing a matching upper bound to Larsen and Nielsen's [30] lower bound: an ORAM scheme with $O(\log N)$ overhead and negligible security in λ, where N is the size of the memory and λ is the security parameter, assuming one-way functions. More concretely, we show: [7]

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

[7] Note that for the (sub-)exponential security regime, e.g., failure probability of $2^{-\lambda}$ or $2^{-\lambda^\epsilon}$ for some $\epsilon \in (0, 1)$, perfectly secure ORAM schemes [12, 16] asymptotically outperform known statistically or computationally secure constructions assuming that $N = \mathsf{poly}(\lambda)$.

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

In the aforementioned results and throughout this paper, unless otherwise noted, we shall assume a standard word-RAM where each memory word has at least $w = \log N$ bits, i.e., large enough to store its own logical address. We assume that word-level addition and boolean operations can be done in unit cost. We assume that the CPU has constant number of private registers. For our ORAM construction, we additionally assume that a single evaluation of a pseudorandom function (PRF), resulting in at least word-size number of pseudo-random bits, can be done in unit cost.[8] Note that all earlier computationally secure ORAM schemes, starting with the work of Goldreich and Ostrovsky [22, 23], make the same set of assumptions. Additionally, we remark that our result can be made statistically secure if one assumes a private random oracle to replace the PRF (the known logarithmic ORAM lower bound [22,23,30] still hold in this setting). Finally, we note that our construction suffers from huge constants due to the use of certain expander graphs; improving the concrete constant is left for future work.

In the full version [5], we provide a comparison with previous works, where we make the comparison more accurate and meaningful by explicitly stating the dependence on the error probability (which was assumed to be some negligible functions in previous works).

1.2 Our Results: Optimal Oblivious Tight Compaction

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

Tight compaction is the following task: given an input array of size n containing either real or dummy elements, output a permutation of the input array where all real elements appear in the front. Tight compaction can be considered as a restricted form of sorting, where each element in the input array receives a 1-bit key, indicating whether it is real or dummy. One naïve solution for tight compaction, therefore, is to rely on oblivious sorting to sort the input array [2,24]; unfortunately, due to recent lower bounds [17, 33], we know that any oblivious sorting scheme must incur $\Omega(n \cdot \log n)$ time on a word-RAM, either assuming

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

that the algorithm treats each element as "indivisible" [33] or assuming that the famous Li-Li network coding conjecture [32] is true [17].

A natural question, therefore, is whether we can do asymptotically better than just naïvely sorting the input. It turns out that this question is related to a line of work in the classical algorithms literature, that is, the design of switching networks and routing on such networks [2,4,6,18,43,44]. First, a line of combinatorial works showed the existence of linear-sized super-concentrators [42, 43, 50], i.e., switching networks with n inputs and n outputs such that vertex-disjoint paths exist from any k elements in the inputs to any k positions in the outputs. One could leverage a linear-sized super-concentrator construction to *obliviously* route all the real elements in the input to the front of the output array deterministically and in linear time (by routing elements along the routes), but it is not clear yet how to find routes (i.e., a set of vertex-disjoint paths) from the real input positions to the front of the output array.

In an elegant work in 1996, Pippenger [44] showed a deterministic, linear-time algorithm for route-finding but unfortunately the algorithm is *not oblivious*. Shortly afterwards, Leighton et al. [31] showed a probabilistic algorithm that tightly compacts n elements in $O(n \cdot \log \log \lambda)$ time with $1 - \mathsf{negl}(\lambda)$ probability — their algorithm is *almost oblivious* except for leaking the number of reals and dummies. After Leighton et al. [31], this line of work remained somewhat stagnant for almost two decades. Only recently, did we see some new results: Mitchell and Zimmerman [38] as well as Lin et al. [33] showed how to achieve the same asymptotics as Leighton et al. [31] but now making the algorithm fully oblivious.

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

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

Our algorithm is *not comparison-based* and *not stable* and this is inherent. Specifically, Lin et al. [33] recently showed that any stable, oblivious tight compaction algorithm (that treats elements as indivisible) must incur $\Omega(n \cdot \log n)$ runtime, where stability requires that the real elements in the output must appear in the same order as the input. Further, due to the well-known 0-1 principle [1,15], any comparison-based tight compaction algorithm must incur at least $\Omega(n \cdot \log n)$ runtime as well.[9]

Not only our ORAM construction relies on the above compaction algorithm in several key points, but it is a useful primitive independently. For example, we use our compaction algorithm to give a perfectly oblivious algorithm that

[9] Although the algorithm of Leighton et al. [31] appears to be comparison-based, it is in fact not since the algorithm must tally the number of reals/dummies and make use of this number.

randomly permutes arrays of n elements in (worst-case) $O(n \cdot \log n)$ time. All previously known such constructions have some probability of failure.

2 Technical Roadmap

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

2.1 Oblivious RAM

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

Hierarchical ORAM. The hierarchical ORAM framework, introduced by Goldreich and Ostrovsky [22, 23] and improved in subsequent works (e.g., [10, 25, 29]), works as follows. For a logical memory of N blocks, we construct a hierarchy of hash tables, henceforth denoted $T_1, \ldots, T_L$ where $L = \log N$. Each T_i stores 2^i memory blocks. We refer to table T_i as the i-th level. In addition, we store next to each table a flag indicating whether the table is *full* or *empty*. When receiving an access request to $\mathsf{read}/\mathsf{write}$ 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 $\ell := L$. Merge all $\{T_j\}_{0 \le j \le \ell}$ into T_ℓ. Mark all levels $T_1, \ldots, T_{\ell-1}$ as empty and T_ℓ as full.

For each access, we perform $\log N$ lookups, one per hash table. Moreover, after t accesses, we rebuild the i-th table $\lceil t/2^i \rceil$ times. When implementing the hash table using the best known oblivious hash table (e.g., oblivious Cuckoo hashing [10, 25, 29]), building a level with 2^k items obliviously requires $O(2^k \cdot \log(2^k)) = O(2^k \cdot k)$ time. This building algorithm is based on oblivious sorting, and its time overhead is inherited from the time overhead of the oblivious sort procedure (specifically, the best known algorithm for obliviously sorting n elements takes $O(n \cdot \log n)$ time [2, 24]). Thus, summing over all levels (and ignoring the $\log N$ lookup operations across different levels for each access), t accesses require $\sum_{i=1}^{\log N} \left\lceil \frac{t}{2^i} \right\rceil \cdot O(2^i \cdot i) = O(t \cdot \log^2 N)$ time. On the other hand, lookup takes

essentially constant time per level (ignoring searching in stashes which introduce an additive factor) and thus $O(\log N)$ per access. Thus, there is an asymmetry between build time and lookup time, and the main overhead is the build.

The work of Patel et al. [40]. Classically (e.g., [10,22,23,25,29]), oblivious hash tables were built to support (and be secure for) *every* input array. This required expensive oblivious sorting, causing the extra logarithmic factor. The key idea of Patel et al. [40] is to modify the hierarchical ORAM framework to realize ORAM from a weaker primitive: an oblivious hash table that works only for *randomly shuffled input* arrays. Patel et al. describe a novel oblivious hash table such that building a hash table containing n elements can be accomplished without oblivious sorting and consumes only $O(n \cdot \log\log\lambda)$ total time[10] and lookup consumes $O(\log\log n)$ total time. Patel et al. argue that their hash table construction retains security not necessarily for every input, but when the input array is randomly permuted, and moreover the input permutation must be unknown to the adversary.

To be able to leverage this relaxed hash table in hierarchical ORAM, a remaining question is the following: whenever a level is being rebuilt in the ORAM (i.e., a new hash table is being constructed), how do we make sure that the input array is randomly and secretly shuffled? A naïve answer is to employ an oblivious random permutation to permute the input, but known oblivious random permutation constructions require oblivious sorting which brings us back to our starting point. Patel et al. solve this problem and show that there is no need to completely shuffle the input array. Recall that when building some level T_ℓ, the input array consists of only unvisited elements in tables $T_0, \ldots, T_{\ell-1}$ (and T_ℓ too if ℓ is the largest level). Patel et al. argue that the unvisited elements in tables $T_0, \ldots, T_{\ell-1}$ are already randomly permuted *within each table* and the permutation is unknown to the adversary. Then, they presented a new algorithm, called *multi-array shuffle*, that combines these arrays to a shuffled array within $O(n \cdot \log\log\lambda)$ time, where $n = |T_0| + |T_1| + \ldots + |T_{\ell-1}|$.[11] The algorithm is somewhat involved, randomized, and has a negligible probability of failure.

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

1. We show how to implement our variant of multi-array shuffle (called *intersperse*) in $O(n)$ time. Specifically, we show a new reduction from *intersperse* to tight compaction.
2. We develop a hash table that supports build in $O(n)$ time assuming that the input array is randomly shuffled. The lookup is $O(1)$, ignoring time spent

[10] 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.

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

on looking in stashes. Achieving this is rather non-trivial: first we use a "packing" style trick to construct oblivious Cuckoo hash tables for small sizes where $n \leq \mathsf{poly}\log\lambda$, achieving linear-time build and constant-time lookup. Relying on the advantage we gain for problems of small sizes, we then show how to solve problems of medium and large sizes, again relying on oblivious tight compaction as a building block. The bootstrapping step from medium to large is inspired by Patel et al. [40] at a very high level, but our concrete construction differs from Patel et al. [40] in many technical details.

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

2.1.1 Interspersing Randomly Shuffled Arrays Given two arrays, $\mathbf{I}_1$ and $\mathbf{I}_2$, of size n_1, n_2, respectively, where each array is randomly shuffled, our goal is to output a single array that contains all elements from $\mathbf{I}_1$ and $\mathbf{I}_2$ in a randomly shuffled order. Ignoring obliviousness, we could first initialize an output array of size $n = n_1 + n_2$, mark exactly n_1 random locations in the output array, and place the elements from $\mathbf{I}_1$ arbitrarily in these locations. The elements from $\mathbf{I}_2$ are placed in the unmarked locations.[12] The challenge is how to perform this placement obliviously, without revealing the mapping from the input array to the output array.

We observe that this routing problem is exactly the "reverse" problem of oblivious tight compaction, where one is given an input array of size n containing keys that are 1-bit and the goal is to sort the array such that all elements with key 0 appear before all elements with key 1. Intuitively, by running this algorithm "in reverse", we obtain a linear time algorithm for *obliviously* routing marked elements to an array with marked positions (that are not necessarily at the front). Since we believe that this procedure is useful in its own right, we formalize it independently and call it *oblivious distribution*. The full details appear in the full version [5].

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

Warmup: oblivious hash table with $\mathbf{poly\log\log}\boldsymbol{\lambda}$ *slack.* Intuitively, to build a hash table, the idea is to randomly distribute the n elements in the input into $B := n/\mathsf{poly}\log\lambda$ bins of size $\mathsf{poly}\log\lambda$ in the clear. The distribution is

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

done according to a pseudorandom function with some secret key K, where an element with address addr is placed in the bin with index $\mathsf{PRF}_K(\mathsf{addr})$. Whenever we lookup for a real element addr', we access the bin $\mathsf{PRF}_K(\mathsf{addr}')$; in which case, we might either find the element there (if it was originally one of the n elements in the input) or we might not find it in the accessed bin (in the case where the element is not part of the input array). Whenever we perform a dummy lookup, we just access a random bin.

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

To this end, we first throw the n balls into the B bins as before, revealing loads $n_1, \ldots, n_B$. Then, we sample new *secret* loads $L_1, \ldots, L_B$ corresponding to an independent process of throwing $n' := n \cdot (1 - 1/\mathsf{poly}\log\lambda)$ balls into B bins. By a Chernoff bound, with overwhelming probability $L_i < n_i$ for every $i \in [B]$. We extract from each bin arbitrary $n_i - L_i$ balls obliviously and move them to an overflow pile (without revealing the L_i's). The overflow pile contains only $n/\mathsf{poly}\log\lambda$ elements so we use a standard Cuckoo hashing scheme such that it can be built in $O(m \cdot \log m) = O(n)$ time and supports lookups effectively in $O(1)$ time (ignoring the stash).[13] The crux of the security proof is showing that since the secret loads $L_1, \ldots, L_B$ are never revealed, they are large enough to mask the access pattern in the lookup phase so that it looks independent of the one leaked in the build phase.

We glossed over many technical details, the most important ones being how the bin sizes are truncated to the secret loads $L_1, \ldots, L_B$, and how each bin is being implemented. For the second question, since the bins are of $O(\mathsf{poly}\log\lambda)$ size, we support lookups using a perfectly secure ORAM constructions that can be built in $O(\mathsf{poly}\log\lambda \cdot \mathsf{poly}\log\log\lambda)$ and looked up in $O(\mathsf{poly}\log\log\lambda)$ time [12,16] (this is essentially where our $\mathsf{poly}\log\log$ factor comes from in this warmup). The first question is solved by employing our linear time tight compaction algorithm to extract the number of elements we want from each bin.

The full details of the construction appear in Section 5.

Remark 2.1 (Comparison of the warmup construction with Patel et al. [40])**.** *Our warmup construction borrows the idea of revealing loads and then sampling new secret loads from Patel et al. However, our concrete instantiation is different and this difference is crucial for the next step where we get an optimal hash table. Particularly, the construction of Patel et al. has* $\log\log\lambda$ *layers of hash tables of decreasing sizes, and one has to look for an element in each one of these hash tables, i.e., searching within* $\log\log\lambda$ *bins. In our solution, by tightening the analysis (that is, the Chernoff bound), we show that a single layer of hash*

[13] We refer to the full version [5] for background information on Cuckoo hashing.

tables suffices; thus, lookup accesses only a single bin. This allows us to focus on optimizing the implementation of a bin towards the optimal construction.

Oblivious hash table with linear build time and constant lookup time. In the warmup construction, (ignoring the lookup time in the stash of the overflow pile[14]), the only super-linear operation that we have is the use of a perfectly secure ORAM, which we employ for bins of size $O(\mathsf{poly}\log\lambda)$. In this step, we replace this with a data structure with linear time build and constant time lookup: a Cuckoo hash table for lists of polylogarithmic size.

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

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

1. For step (1) the $n = \mathsf{poly}\log\lambda$ elements in the input array can each evaluate the PRF on its associated key, and write down its two bin choices (this takes linear time).
2. Implementing step (2) in linear time is harder as this step is dominated by a sequence of oblivious sorts. To overcome this, we use the fact that the problem size n is of size $\mathsf{poly}\log\lambda$. As a result, the index of each item and its two bin choices can be expressed using $O(\log\log\lambda)$ bits which means that a single memory word (which is $\log\lambda$ bits long) can hold $O\left(\frac{\log\lambda}{\log\log\lambda}\right)$ many elements' metadata. We can now apply a "packed sorting" type of idea [3,11,14,28] where we use the RAM's word-level instructions to perform SIMD-style operations. Through this packing trick, we show that oblivious

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

sorting and oblivious random permutation (of the elements' metadata) can be accomplished in $O(n)$ time!

3. Step (3) is classically implemented using oblivious bin distribution which again uses oblivious sorts. Here, we cannot use the packing trick since we operate on the elements themselves, so we use the fact that the input array is randomly shuffled and just route the elements in the clear.

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

We refer the reader to Section 6 for the full details of the construction.

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

Additionally, some technicalities remain in how we treat the smallest level of the ORAM and the stashes. The smallest level in the ORAM construction cannot use the hash table construction described earlier. This is because elements are added to the smallest level as soon as they are accessed and our hash table does not support such an insertion. We address this by using an oblivious dictionary built atop a perfectly secure ORAM for the smallest level of the ORAM. This incurs an additive $O(\mathsf{poly}\log\log\lambda)$ blowup. Finally, the stashes for each of the Cuckoo hash tables (at every level and every bin within the level) incur $O(\log\lambda)$ time. We leverage the techniques from Kushilevitz et al. [29] to merge all stashes into a common stash of size $O(\log^2\lambda)$, which is added to the smallest level when it is rebuilt.

On deamortization. As the overhead of our ORAM is amortized over several accesses, it is natural to ask whether we can deamortize the construction to achieve the same overhead in the worst case, per access. Historically, Ostrovsky and Shoup [39] deamortized the hierarchical ORAM of Goldreich and Ostrovsky [23], and related techniques were later applied on other hierarchical ORAM

schemes [10, 26, 29]. Unfortunately, the technique fails for our ORAM as we explain below (it fails for Patel et al. [40], as well, by the same reason).

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

2.2 Tight Compaction

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

Reduction to loose compaction. Pippenger's self-routing super-concentrator construction [44] proposes a technique that reduces the task of tight compaction to that of loose compaction. Informally speaking, loose compaction receives as input a sparse array, containing a few real elements and many dummy elements. The output is a compressed output array, containing all real elements but the procedure does not necessarily remove all the dummy elements. More concretely, we care about a specific form of loose compactor (parametrized by n): consider a suitable bipartite expander graph that has n vertices on the left and $n/2$ vertices on the right where each node has constant degree. At most $1/128$ fraction of the vertices on the left will receive a real element, and we would like to route *all* real elements over vertex-disjoint paths to the right side such that every right vertex receives at most 1 element. The crux is to find a set of satisfying routes in linear time and obliviously. Once a set of feasible routes have been identified, it is easy to see that performing the actual routing can be done obliviously in linear time (and for obliviousness we need to route a dummy element over an edge that bears 0 load). During this process, we effectively compress the sparse input array (represented by vertices on the left) by $1/2$ without losing any element.

Using Pippenger's techniques [44] and with a little extra work, we can derive the following claim — at this point we simply state the claim while deferring algorithmic details to subsequent technical sections. Below D denotes the number of bits it takes to encode an element and w denotes the word size:

> *Claim:* There exist appropriate constants $C, C' > 6$ such that the following holds: if we can solve the aforementioned loose compaction problem oblivi-

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

As mentioned, the crux is to find satisfying routes for such a "loose compactor" bipartite graph obliviously and in linear time. Achieving this is non-trivial: for example, the recent work of Chan et al. [12] attempted to do this but their route-finding algorithm requires $O(n \log n)$ runtime — thus Chan et al. [12]'s work also implies a loose compaction algorithm that runs in time $O(n \log n + \lceil D/w \rceil \cdot n)$. To remove the extra $\log n$ factor, we introduce two new ideas, *packing*, and *decomposition* — in fact both ideas are remotely reminiscent of a line of works in the core algorithms literature on (non-comparison-based, non-oblivious) integer sorting on RAMs [3, 14, 28] but obviously we apply these techniques to a different context.

Packing: linear-time compaction for small instances. We observe that the offline route-finding phase operates only on metadata. Specifically, the route-finding phase receives the following as input: an array of n bits where the i-th bit indicates whether the i-th input position is real or dummy. If the problem size n is small, specifically, if $n \leq w/\log w$ where w denotes the width of a memory word, we can pack the entire problem into a single memory word (since each element's index can be described in $\log n$ bits). In our technical sections we will show how to rely on word-level addition and boolean operations to solve such small problem instances in $O(n)$ time. At a high level, we follow the slow route-finding algorithm by Chan et al. [12], but now within a single memory word, we can effectively perform SIMD-style operations and we exploit this to speed up Chan et al. [12]'s algorithm by a logarithmic factor for small instances.

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

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

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

1. *Medium instances:* $n \leq (w/\log w)^2$. For medium instances, our idea is to divide the input array into $\sqrt{n}$ segments each of $B := \sqrt{n}$ size. As long as the input array has only $n/128$ or fewer real elements, then at most $\sqrt{n}/4$ segments can be dense, i.e., each containing more than $\sqrt{n}/4$ real elements ($1/4$ is loose but sufficient). We rely on tight compaction for small instances to move the dense segments in front of the sparse ones. For each

of $3\sqrt{n}/4$ sparse segments, we next compress away 3/4 of the space using tight compaction for small instances. Clearly, the above procedure is a loose compaction and consumes at most $2 \cdot C \cdot C' \cdot C_0 \cdot \lceil D/w \rceil \cdot n + 6\lceil D/w \rceil \cdot n \leq 2.5 \cdot C \cdot C' \cdot C_0 \cdot \lceil D/w \rceil \cdot n$ runtime.

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

2. *Large instances: arbitrary n.* We can now bootstrap to arbitrary choices of n by dividing the problem into $m := n/(\frac{w}{\log w})^2$ segments where each segment contains at most $(\frac{w}{\log w})^2$ elements. Similar to the medium case, at most 1/4 fraction of the segments can have real density exceeding 1/4 — which we call such segments *dense*. As before, we would like to move the dense segments in the front and the sparse ones to the end. Recall that Chan et al. [12]'s algorithm solves loose compaction for problems of arbitrary size m in time $C_1 \cdot (m \log m + \lceil D/w \rceil m)$ Thus due to the above claim we can solve tight compaction for problems of any size m in time $C \cdot C_1 \cdot (m \log m + \lceil D/w \rceil \cdot m) + C' \cdot \lceil D/w \rceil \cdot m$. Thus, in $O(\lceil D/w \rceil \cdot n)$ time we can move all the dense instances to the front and the sparse instances to the end. Finally, by invoking medium instances of tight compaction, we can compact within each segment in time that is linear in the size of the segment. This allows us to compress away 3/4 of the space from the last 3/4 segments which are guaranteed to be sparse. This gives us loose compaction for large instances in $O(\lceil D/w \rceil \cdot n)$ time — from here we can construct oblivious tight compaction for large instances using the above Claim.[15]

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

Organization. In Section 3 we highlight several building blocks that are necessary for our construction. In Section 4 we describe our oblivious tight compaction algorithm informally (due to lack of space). In Section 5 we provide our construction of hash table for shuffled input for long size inputs, and in Section 6 we provide our construction of hash table for small size input, i.e., how we organize the bins. Our ORAM construction is provided in Section 7.

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

3 Oblivious Building Blocks

Our ORAM construction uses many building blocks, some of which new to this work and some of which are known from the literature. The building blocks are listed next. Due to lack of space, we just mention the building blocks and refer the reader to the full paper [5] for formal definitions.

Oblivious Sorting Algorithms: 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 Random Permutations: We show how to perform *efficient* oblivious random permutations in settings where each memory word can hold multiple elements.

Oblivious Bin Placement: We state the known results for oblivious bin placement of Chan et al. [10, 13].

Oblivious Hashing: 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. Due to its importance, we provide some high level of the functionality here.

In a nutshell, the functionality of hash function supports three commands: A "constructor" $\mathsf{Build}(\mathbf{I})$, receiving an array of n pairs of key/value (k_i, v_i). The array $\mathbf{I}$ is assumed to be randomly shuffled, and the instruction builds some internal structure for supporting fast (and oblivious) future accesses. Then, the construction supports several $\mathsf{Lookup}(k)$ instructions, where if $k \in \mathbf{I}$ then the corresponding v should be returned, and otherwise $\perp$ is returned. Importantly, the Lookup should also support fictitious lookups, i.e., supports $k = \perp$. The construction should not leak whether $k \in \mathbf{I}, k \notin \mathbf{I}$ or $k = \perp$. Finally, the hash function also supports the "destructor" function $\mathsf{Extract}()$ – which returns a permuted array of size n consisting of all elements in $\mathbf{I}$ that were not accessed padded with dummies. The security definition requires that the joint distribution of access pattern, where the adversary can choose the sequence of instructions and the inputs to the instructions, is simulatable. The only restriction is that the adversary cannot ask for the same key more than once. We call this functionality "oblivious hash table for non-recurrent lookups", see full paper for formal definition.

Oblivious Cuckoo Hashing: 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: We present and analyze a simple construction of a dictionary that is achieved by compiling a non-oblivious dictionary (e.g., a red-black tree) inside an existing ORAM construction.

Oblivious Balls-into-Bins Sampling: We present an oblivious sampling of the approximated bin loads of throwing independently n balls into m bins, which uses the binomial sampling of Bringmann et al. [9].

Oblivious tight compaction: As was mentioned in the introduction, one of our main contributions is a deterministic linear time procedure (in the balls and bins model) for the following problem: given an input array containing n balls, each of which marked with a 1-bit label that is either 0 or 1, output a permutation of the array such that all the 1 balls are moved to the front.

Intersperse: Given two arrays that are assumed to be randomly shuffled $\mathbf{I}_0, \mathbf{I}_1$ of sizes n_0, n_1, resp., we show a procedure $\mathsf{Intersperse}_{n_0+n_1}(\mathbf{I}_0 \| \mathbf{I}_1, n_0, n_1)$ that returns a random permutation of $\mathbf{I}_0 \| \mathbf{I}_1$. We generalize it also for interspersing k arrays $\mathsf{Intersperse}^{(k)}_{n_1,\ldots,n_k}(\mathbf{I}_1 \| \ldots \| \mathbf{I}_k)$, each of which is randomly shuffled, and for interspersing real and dummy elements $\mathsf{IntersperseRD}$, assuming that the real elements are randomly shuffled but in which we have no guarantee of the relative positions of the real elements with respect to the dummy ones. In all of these variants, the goal is to return a random permutation of all elements in the input array, while the assumption on the input helps to reduces the running time.

Notation. Throughout the paper, we use the notation $\delta^{\mathcal{A}}$-*secure* PRF to mean that for every (non-uniform) probabilistic polynomial-time algorithm $\mathcal{A}$ has advantage at most $\delta^{\mathcal{A}}$ in distinguishing an output of the PRF from random. We additionally say that an algorithm is $(1 - \delta^{\mathcal{A}})$-*oblivious* if no (non-uniform) probabilistic polynomial-time algorithm $\mathcal{A}$ can distinguish its access pattern from a simulated one with probability better than $\delta^{\mathcal{A}}$. Formal definitions appear in the full paper.

4 Oblivious Tight Compaction in Linear Time

Our tight compaction algorithm works in two main steps. We first reduce the problem (in linear time) to a relaxed problem called *loose compaction*. Here, one is given an array $\mathbf{I}$ with n elements in which it is guaranteed that at most n/ℓ elements are real for some constant $\ell > 2$, and the goal is to return an array of size $n/2$ that contain all the real elements. Second, we implement loose compaction in linear time.

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

The main idea here is to perform the swaps along the edges of a bipartite expander graph. Consider a bipartite expander graph where the left nodes are associated with the elements. The edges of the graph are the access pattern of our algorithm. We will swap two misplaced elements that have a different mark

if they have a common neighbor on the right. To make sure this algorithm runs in linear time, the graph has to be d-regular for $d = O(1)$ and that the list of neighbors of every node can be computed using $O(1)$ basic operations. Such explicit expander graphs are known to exist (for example, Margulis [37]).

Using the expansion properties of the graph, we can upper bound the number of misplaced elements that were not swapped by this process: at most n/ℓ for some $\ell > 2$. Thus, we can invoke loose compaction where we consider the remaining misplaced element as real elements and the rest being dummy. This process reduced the problem from n elements to $n/2$ and we proceed in recursion to swap the misplaced elements on that first half of the array, until all 0-elements and 1-elements are swapped. (The reduction is of logarithmic depth and the problem size is shrunk by a factor two in each step so the complexity is linear overall.)

Loose compaction. The loose compaction algorithm $\mathsf{LooseCompaction}_\ell$ receives as input an array $\mathbf{I}$ consisting of n balls, where at most n/ℓ are real and the rest are dummies for some $\ell > 2$. The goal is to return an array of size $n/2$ where all the real balls reside in the returned array. In this algorithm we again use a bipartite expander and combine it with ideas coming from the matching algorithm of Pippenger [44]. The main idea of the procedure is to first distribute the real balls to many bins, while ensuring that no bin consists of too many real balls. Then, as all bins have small load, we can merge several bins together and compact the array, as required.

The input/output of step 1, namely of the balanced distribution, is as follows. The input is an array $\mathbf{I}$ of size n that we interpret as n/B bins of size B each (simply by considering the array $\mathbf{I}[(i-1)\cdot B + 1, \ldots, iB]$ as the i-th bin, for $i = 1, \ldots, n/B$). If a bin contains more than $B/4$ real balls, then we call it "dense", and otherwise we call it "sparse". Our goal is to distribute all *dense* bins in $\mathbf{I}$ into another array $\mathbf{I}'$ of size n which we think about as split into n/B bins of size B each. The procedure computes which target bins in $\mathbf{I}'$ we should distribute each one of the dense bins in $\mathbf{I}$, such that, the distribution would be balanced. In particular, the balanced distribution guarantees that no bin in $\mathbf{I}'$ receives more than $B/4$ real balls. Let us explain why this balanced distribution is enough.

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

Balanced distribution of the dense bins. The distribution of dense bins in $\mathbf{I}$ into $\mathbf{I}'$ relies (again) on a good expander graph. Fixing a proper constant ϵ, we consider a d_ϵ-regular graph $G_{\epsilon,n/B} = (L, R, E)$ with $|L| = |R| = n/B$, where L corresponds to $\mathbf{I}$, R corresponds to $\mathbf{I}'$ and we let $B = d_\epsilon/2$. Let $S \subset L$ be

the set of dense bins in L. We look for a $(B, B/4)$-*matching for* S: We look for a set of edges $M \subseteq E$ such that (1) from every bin in S there are at least B out edges, and (2) for every bin in R there are at most $B/4$ incoming edges. Given such a matching M, every dense bin in $\mathbf{I}$ can be distributed to $\mathbf{I}'$ while guaranteeing that no bin in $\mathbf{I}'$ will have load greater than $B/4$, while the access pattern corresponds to edges in the graph which is public and known to the adversary.

Computing the matching. We first describe a non-oblivious algorithm for finding the matching; the algorithm is due to Pippenger [44]. Let $m = |L| = |R|$.[16] The algorithm proceeds in rounds, where initially all *dense* vertices in L are "unsatisfied", and in each round:

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

The output is the edges in the matching. In each round, there are $O(m)$ transmitted messages, where each message is a single bit. Using properties of the expander graphs, in each round the number of unsatisfied vertices decreases by a factor of 2. Thus, the algorithm proceeds in $O(\log m)$ rounds, and the total runtime of the algorithm is $O(m)$.[17] However, the algorithm is non-oblivious.

Oblivious slow matching (for any $\boldsymbol{m}$). A simple way to make this algorithm oblivious (as observed by [12]) is by sending a message from every vertex in L to the relevant vertices in R in each round, that is, even a vertex is satisfied it still sends fictitious messages in the proceeding rounds. In particular, in each round the algorithm hides whether a vertex $v \in L$ is in the set of satisfied vertices ($v \notin L'$) or is still unsatisfied ($v \in L'$), and in fact, we run each iteration on the entire graph. This results in algorithm that takes overall $O(m \cdot \log m)$ time.

Oblivious fast matching (for small $\boldsymbol{m}$). When m is really small, we use the packing trick. Concretely, when $m \leq \frac{w}{\log w}$, where w is the word size, all the information required for the algorithm can be packed into $O(1)$ words. Thus, when accessing information related to one node $u \in L$, we can access at the same time all the information regarding all other nodes in L. This enables us to hide which node is being visited (i.e., whether a node is in L' or not) and

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

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

therefore the algorithm can now just visit the nodes in L' and does not have to make fictitious accesses on the entire graph. As a result, when m is small (as above) we are able to compute the matching in $O(m)$ word-level operations.

Combining slow match and fast match. We achieve loose compaction as follows:

- Given the array $\mathbf{I}$ of size n and word size $w = \Omega(\log n)$, we first break it into blocks of size $p^2 = (w/\log w)^2$, and our goal is to move all "dense" blocks to the beginning of the array. We find the matching obliviously using the "slow" matching algorithm, that takes $O(m \cdot \log m) = O\left(\frac{n}{p^2} \cdot \log \frac{n}{p^2}\right)$ time, which is linear. Then, compaction given the matching (by folding) takes $O(n)$. By running this compaction twice we get an output of size $n/4$, consisting of all dense blocks of size p^2.
- At this point, we want to run compaction on each one of the sparse blocks in $\mathbf{I}$ (where again, blocks are of size p^2) independently, and then take only the result of the compaction of each block for the remaining part of the output array.
  In order to run the compaction on each block of size p^2, we perform the same trick again. We break each instance into p sub-blocks of a *smaller* size p, and mark each sub-block as dense or sparse. As the number of sub-blocks we have in each instance is $p = w/\log w$, we can find the matching using the fast matching algorithm. Note that as previously, we did not handle the real balls in the sparse sub-blocks.
- Finally, we have to solve compaction of all sparse sub-blocks of the previous step. Each sparse sub-block is of size $p = w/\log w$, and thus can be solved in linear time using the fast matching algorithm.

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

We refer to the full version [5] for the formal description and analysis.

5 BigHT: Oblivious Hashing for Non-Recurrent Lookups

The hash table construction we describe in this section suffers from $\mathsf{poly}\log\log\lambda$ extra multiplicative factor in Build and Lookup (which lead to similar overhead in the implied ORAM construction). Nevertheless, this hash table serves as a first step and we will get rid of the extra factor in Section 6. Hence, the parameter of expected bin load $\mu = \log^9 \lambda$ is seemingly loose in this section but is necessary later in Section 6 (to apply Cuckoo hash). Additionally, note that this hash table captures and simplifies many of the ideas in the oblivious hash table of Patel et al. [40] and can be used to get an ORAM with similar overhead to theirs.

Construction 5.1: Hash Table for Shuffled Inputs

Procedure BigHT.Build(I):

- **Input:** An array $\mathbf{I} = (a_1, \ldots, a_n)$ containing n elements, where each a_i is either dummy or a (key, value) pair denoted (k_i, v_i), where both the key k and the value v are D-bit strings where $D := O(1) \cdot w$.
- **Input assumption:** The elements in the array $\mathbf{I}$ are uniformly shuffled.
- **The algorithm:**
  1. Let $\mu := \log^9 \lambda$, $\epsilon := \frac{1}{\log^2 \lambda}$, $\delta := e^{-\log \lambda \cdot \log \log \lambda}$, and $B := \lceil n/\mu \rceil$.
  2. *Sample PRF key.* Sample a random PRF secret key sk.
  3. *Directly hash into major bins.* Throw the real $a_i = (k_i, v_i)$ into B bins using $\mathsf{PRF}_{\mathsf{sk}}(k_i)$. If $a_i = \mathsf{dummy}$, throw it to a uniformly random bin. Let $\mathsf{Bin}_1, \ldots, \mathsf{Bin}_B$ be the resulted bins.
  4. *Sample independent smaller loads.* Sample[18] the load of throwing n' balls into B bins with failure probability δ, where $n' = n \cdot (1 - \epsilon)$. Let $(L_1, \ldots, L_B)$ be the resulted loads. If there exists $i \in [B]$ such that $||\mathsf{Bin}_i| - \mu| > 0.5 \cdot \epsilon\mu$ or $\left|L_i - \frac{n'}{B}\right| > 0.5 \cdot \epsilon\mu$, then abort.
  5. *Create major bins.* Allocate new arrays $(\mathsf{Bin}'_1, \ldots, \mathsf{Bin}'_B)$, each of size μ. For every i, iterate in parallel on both Bin_i and Bin'_i, and copy the first L_i elements in Bin_i to Bin'_i. Fill the empty slots in Bin'_i with dummy. (L_i is not revealed during this process, by continuing to iterate over Bin_i after we cross the threshold L_i.)
  6. *Create overflow pile.* Obliviously merge all of the last $|\mathsf{Bin}_i| - L_i$ elements in each bin $\mathsf{Bin}_1, \ldots, \mathsf{Bin}_B$ into an overflow pile:
     - For each $i \in [B]$, replace the first L_i positions with dummy.
     - Concatenate all of the resulting bins and perform oblivious tight compaction on the resulting array such that the real balls appear in the front. Truncate the outcome to be of length ϵn.
  7. Prepare an oblivious hash table for elements in the overflow pile by calling the Build algorithm of the $(1 - O(\delta) - \delta^{\mathcal{A}}_{\mathsf{PRF}})$-oblivious Cuckoo hashing scheme (see Building blocks, Section 3) parameterized by δ (recall that $\delta = e^{-\Omega(\log \lambda \cdot \log \log \lambda)}$) and the stash size $\log(1/\delta)/\log n$. Let $\mathsf{OF} = (\mathsf{OF}_{\mathsf{T}}, \mathsf{OF}_{\mathsf{S}})$ denote the outcome data structure. Henceforth, we use $\mathsf{OF.Lookup}$ to denote a lookup operation to this oblivious Cuckoo hashing scheme.
  8. *Prepare data structure for efficient lookup.* For $i = 1, \ldots, B$, call $\mathsf{naïveHT.Build}(\mathsf{Bin}'_i)$ on each major bin to construct an oblivious hash table, and let OBin_i denote the outcome for the i-th bin.
- **Output:** The algorithm stores in the memory a state that consists of $(\mathsf{OBin}_1, \ldots, \mathsf{OBin}_B, \mathsf{OF}, \mathsf{sk})$.

Procedure BigHT.Lookup(k):

- **Input:** The secret state $(\mathsf{OBin}_1, \ldots, \mathsf{OBin}_B, \mathsf{OF}, \mathsf{sk})$, and a key k to look for (that may be $\perp$, i.e., dummy).

[18] See the full paper for more information on how to perform this step obliviously.

– **The algorithm:**
  1. Call $v \leftarrow \mathsf{OF.Lookup}(k)$.
  2. If $k = \perp$, choose a random bin $i \xleftarrow{\$} [B]$ and call $\mathsf{OBin}_i.\mathsf{Lookup}(\perp)$.
  3. If $k \neq \perp$ and $v \neq \perp$ (i.e., v was found in OF), choose a random bin $i \xleftarrow{\$} [B]$ and call $\mathsf{OBin}_i.\mathsf{Lookup}(\perp)$.
  4. If $k \neq \perp$ and $v = \perp$ (i.e., v was not found in OF), let $i := \mathsf{PRF}_{\mathsf{sk}}(k)$ and call $v \leftarrow \mathsf{OBin}_i.\mathsf{Lookup}(k)$.
– **Output:** The value v.

Procedure BigHT.Extract():

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

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

Theorem 5.2. *Assume a $\delta^{\mathcal{A}}_{\mathsf{PRF}}$-secure PRF. Then, Construction 5.1 $(1 - n^2 \cdot e^{-\Omega(\log \lambda \cdot \log\log \lambda)} - \delta^{\mathcal{A}}_{\mathsf{PRF}})$-obliviously implements the hash table functionality (see Section 3) for all $n \geq \log^{11} \lambda$, assuming that the input array (of size n) for* Build *is randomly shuffled. Moreover,*

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

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

The proof of security is given in the full version [5].

Remark 5.3. *As we mentioned, Construction 5.1 is only the first step towards the final oblivious hash table that we use in the final ORAM construction. We make significant optimizations in Section 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 (called* SmallHT*) that is more*

efficient for small lists. Additionally, while it may now seem that the $O(\log \lambda)$-stash overhead of Lookup *is problematic, we will "merge" the stashes for different hash tables in our final ORAM construction and store them again in an oblivious hash table.*

6 SmallHT: Oblivious Hashing for Small Bins

In Section 5, we constructed an oblivious hashing scheme for randomly shuffled inputs where Build and Extract consumes $n \cdot \mathsf{poly} \log \log \lambda$ time and Lookup consumes $\mathsf{poly} \log \log \lambda$. The extra $\mathsf{poly} \log \log \lambda$ factors arise from the oblivious hashing scheme (denoted naïveHT) which we use for each major bin of size $\approx \log^9 \lambda$. To get rid of the extra $\mathsf{poly} \log \log \lambda$ factors, in this section, we will construct a new oblivious hashing scheme for $\mathsf{poly} \log \lambda$-sized arrays which are randomly shuffled. In our new construction, Build and Extract takes linear time and Lookup takes constant time (ignoring the stash which we will treat separately later).

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

We first describe two helpful procedures (mentioned in Section 2.1.2) in Sections 6.1 and 6.2. Then, in Section 6.3, we give the full description of the Build, Lookup, and Extract procedures (Construction 6.5). Throughout this section, we assume for simplicity that $n = \log^9 \lambda$ (while in reality $n \in \log^9 \lambda \pm \log^7 \lambda$).

6.1 Step 1 – Add Dummies and Shuffle

We are given a randomly shuffled array $\mathbf{I}$ of length n that contains real and dummy elements. In Algorithm 6.1, we pad the input array with dummies to match the size of the hash-table to be built. Each dummy will receive a unique index label, and we rely on packed oblivious random permutation to permute the labeled dummies. Finally, we rely on Intersperse on the real balls to make sure that all elements, including reals and dummies, are randomly shuffled.

More formally, the output of Algorithm 6.1 is an array of size $n_{\mathsf{cuckoo}} = \mathsf{c}_{\mathsf{cuckoo}} \cdot n + \log \lambda$, where $\mathsf{c}_{\mathsf{cuckoo}}$ is the constant required for Cuckoo hashing, which

contains all the real elements from $\mathbf{I}$ and the rest are dummies. Furthermore, each dummy receives a distinct random index from $\{1, \ldots, n_{\mathsf{cuckoo}} - n_R\}$, where n_R is the number of real elements in $\mathbf{I}$. Assuming that the real elements in $\mathbf{I}$ are a-priori uniformly shuffled, then the output array is randomly shuffled.

Algorithm 6.1: Shuffle the Real and Dummy Elements

Input: An input array $\mathbf{I}$ of length n consisting of real and dummy elements.
Input Assumption: The real elements among $\mathbf{I}$ are randomly shuffled.
The algorithm:

1. Count the number of real elements in $\mathbf{I}$. Let n_R be the output.
2. Write down a metadata array $\mathbf{MD}$ of length n_{cuckoo}, where the first n_R elements contain only a symbol `real`, and the remaining $n_{\mathsf{cuckoo}} - n_R$ elements are of the form $(\perp, 1), (\perp, 2), \ldots, (\perp, n_{\mathsf{cuckoo}} - n_R)$, i.e., each element is a $\perp$ symbol tagged with a dummy index.
3. Run packed oblivious random permutation (see Section 3)on $\mathbf{MD}$, packing $O\left(\frac{w}{\log n}\right)$ elements into a single memory word. Run oblivious tight compaction on the resulting array, moving all the dummy elements to the end.
4. Run tight compaction on the input $\mathbf{I}$ to move all the real elements to the front.
5. Obliviously write down an array $\mathbf{I}'$ of length n_{cuckoo}, where the first n_R elements are the first n_R elements of $\mathbf{I}$ and the last $n_{\mathsf{cuckoo}} - n_R$ elements are the last $n_{\mathsf{cuckoo}} - n_R$ elements of $\mathbf{MD}$, decompressed to the original length as every entry in the input $\mathbf{I}$.
6. Run $\mathsf{Intersperse}$ on $\mathbf{I}'$ letting $n_1 := n_R$ and $n_2 := n_{\mathsf{cuckoo}} - n_R$. Let $\mathbf{X}$ denote the outcome (permuted) array.

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

Claim 6.2. *Algorithm 6.1 fails with probability at most $e^{-\Omega(\sqrt{n})}$ and completes in $O(n + \frac{n}{w} \cdot \log^3 n)$ time. Specifically, for $n = \log^9 \lambda$ and $w \geq \log^3 \log \lambda$, the algorithm completes in $O(n)$ time and fails with probability $e^{-\Omega(\log^{9/2} \lambda)}$.*

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

6.2 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 (see Section 3). At the end of this step, every element in the input array should learn which bin (either in the main table or

the stash) it is destined for. Recall that the Cuckoo hashing consists of a main table of $\mathsf{c}_{\mathsf{cuckoo}} \cdot n$ bins and a stash of $\log \lambda$ bins.

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

To compute the bin assignments we start with obliviously assigning the bin choices of the real elements in $\mathbf{MD}_{\mathbf{X}}$. Next, we obliviously assign the remaining dummy elements to the remaining available locations. We do so by a sequence of oblivious sort algorithms. See Algorithm 6.3.

Algorithm 6.3: Evaluate Cuckoo Hash Assignment on Metadata

Input: An array $\mathbf{MD}_{\mathbf{X}}$ of length $n_{\mathsf{cuckoo}} = \mathsf{c}_{\mathsf{cuckoo}} \cdot n + \log \lambda$, where each element is either dummy or a pair $(\mathsf{choice}_{i,1}, \mathsf{choice}_{i,2})$, where $\mathsf{choice}_{i,b} \in [\mathsf{c}_{\mathsf{cuckoo}} \cdot n]$ for every $b \in \{1, 2\}$, and the number of real pairs is at most n.

Remark: All oblivious sorting in the algorithm below will be instantiated using packed oblivious sorting (including those called by $\overline{\mathsf{cuckooAssign}}$ and oblivious bin placement).

The algorithm:

1. Run the indiscriminate oblivious Cuckoo assignment algorithm $\overline{\mathsf{cuckooAssign}}$ with parameter $\delta = e^{-\log \lambda \log \log \lambda}$ (where $\overline{\mathsf{cuckooAssign}}$ is formally described in the full paper) 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 $\mathbf{Assign}_{\mathbf{X}}[i] \in \{\mathsf{choice}_{i,1}, \mathsf{choice}_{i,2}\} \cup S_{\mathsf{stash}}$, 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 placement on $\mathbf{Assign}_{\mathbf{X}}$, and let $\mathbf{Occupied}$ be the output array (of length n_{cuckoo}). For every index j we have $\mathbf{Occupied}[j] = i$ if $\mathbf{Assign}_{\mathbf{X}}[i] = j$ for some i. Otherwise, $\mathbf{Occupied}[j] = \perp$.
3. Label the i-th element in $\mathbf{Assign}_{\mathbf{X}}$ with a tag $t = i$ for all i. Run oblivious sorting on $\mathbf{Assign}_{\mathbf{X}}$ and let $\widetilde{\mathbf{Assign}}$ be the resulting array, such that all real elements appear in the front, and all dummies appear at the end, and ordered by their respective dummy-index (i.e. given in Algorithm 6.1, Step 2).
4. Label the i-th element in $\mathbf{Occupied}$ with a tag $t = i$ for all i. Run oblivious sorting on $\mathbf{Occupied}$ and let $\widetilde{\mathbf{Occupied}}$ be the resulting array, such that all occupied bins appear in the front and all empty bins appear at the end (where each empty bin contains an index (i.e., a tag t) of an empty bin in $\mathbf{Occupied}$).
5. Scan both arrays $\widetilde{\mathbf{Assign}}$ and $\widetilde{\mathbf{Occupied}}$ in parallel, updating the destined bin of each dummy element in $\widetilde{\mathbf{Assign}}$ with the respective tag in $\widetilde{\mathbf{Occupied}}$ (and each real element pretends to be updated).

6. Run oblivious sorting on the array $\widetilde{\mathbf{Assign}}$ (back to the original ordering in the array $\mathbf{Assign_X}$) according to the tag labeled in Step 3. Update the assignments of all dummy elements in $\mathbf{Assign_X}$ according to the output array of this step.

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

Claim 6.4. *For $n \geq \log^9 \lambda$, Algorithm 6.3 fails with probability at most $e^{-\Omega(\log\lambda \cdot \log\log\lambda)}$ and completes in $O\left(n \cdot (1 + \frac{\log^3 n}{w})\right)$ time. Specifically, for $n = \log^9 \lambda$ and $w \geq \log^3 \log \lambda$, Algorithm 6.3 completes in $O(n)$ time.*

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

6.3 SmallHT Construction

The full description of the construction is given next. It invokes Algorithms 6.1 and 6.3.

Construction 6.5: SmallHT – **Hash table for Small Bins**

Procedure **SmallHT.Build(I)**:

- **Input:** An input array $\mathbf{I}$ of length n consisting of real and dummy elements. Each real element is of the form (k, v) where both the key k and the value v are D-bit strings where $D := O(1) \cdot w$.
- **Input Assumption:** The real elements among $\mathbf{I}$ are randomly shuffled.
- **The algorithm:**
  1. Run Algorithm 6.1 (prepare real and dummy elements) on input $\mathbf{I}$, and receive back an array $\mathbf{X}$.
  2. Choose a PRF key sk where PRF maps $\{0,1\}^D \to [\mathsf{c}_{\mathsf{cuckoo}} \cdot n]$.
  3. Create a new metadata array $\mathbf{MD_X}$ of length n. Iterate over the the array $\mathbf{X}$ and for each real element $\mathbf{X}[i] = (k_i, v_i)$ compute two values $(\mathsf{choice}_{i,1}, \mathsf{choice}_{i,2}) \leftarrow \mathsf{PRF}_{\mathsf{sk}}(k_i)$, and write $(\mathsf{choice}_{i,1}, \mathsf{choice}_{i,2})$ in the i-th location of $\mathbf{MD_X}$. If $\mathbf{X}[i]$ is dummy, write $(\perp, \perp)$ in the i-th location of $\mathbf{MD_X}$.
  4. Run Algorithm 6.3 on $\mathbf{MD_X}$ to compute the assignment for every element in $\mathbf{X}$. The output of this algorithm, denoted $\mathbf{Assign_X}$, is an array of length n, where in the i-th position we have the destination location of element $\mathbf{X}[i]$.

5. Route the elements of $\mathbf{X}$, in the clear, according to $\mathbf{Assign}_{\mathbf{X}}$, 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 **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 **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 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 $\mathbf{X}' \leftarrow \mathsf{IntersperseRD}_n(\mathbf{X})$ to get a permuted array.
– **Output:** The array $\mathbf{X}'$.

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

Theorem 6.6. *Assume a $\delta^{\mathcal{A}}_{\mathsf{PRF}}$-secure PRF. Suppose that $n = \log^9 \lambda$ and $w \geq \log^3 \log \lambda$. Then, Construction 6.5 $(1 - n \cdot e^{-\Omega(\log \lambda \cdot \log \log \lambda)} - \delta^{\mathcal{A}}_{\mathsf{PRF}})$-obliviously implements the non-recurrent hash table functionality assuming that the input for* Build *(of size n) is randomly shuffled. Moreover,* Build *and* Extract *incur $O(n)$ time,* Lookup *has constant time in addition to linearly scanning a stash of size $O(\log \lambda)$.*

Proof. The proof of security is given in the full version [5]. We proceed with the efficiency analysis. The Build operation executes Algorithm 6.1 that consumes

$O(n)$ time (by Claim 6.2), then performs additional $O(n)$ time, then executes Algorithm 6.3 that consumes $O(n)$ time (by Claim 6.4), and finally performs additional $O(n)$ time. Thus, the total time is $O(n)$. Lookup, by construction, incurs $O(1)$ time in addition to linearly scanning the stash S which is of size $O(\log \lambda)$. The time of Extract is $O(n)$ by construction. □

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 $n = \log^9 \lambda$, this modification is valid. Note that we still assume that the number of elements in the input to CombHT, is at least $\log^{11} \lambda$ (as in Theorem 5.2).

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. To this end, we merge the different stashes of the major bins and store the merged list in an oblivious Cuckoo hash (denoted as CombS later). (A similar idea has also been applied in several prior works [13,25,27,29].) This results with a new hash table scheme we call CombHT. See the full version [5] for precise details.

7 Oblivious RAM

In this section, we utilize CombHT in the hierarchical framework of Goldreich and Ostrovsky [23] to construct our ORAM scheme. We denote by λ the security parameter. For simplicity, we assume that N, the size of the logical memory, is a power of 2. Additionally, we assume that w, the word size is $\Theta(\log N)$.

ORAM Initialization. Our structure consists of one dictionary D (see Section 3), and $O(\log N)$ levels numbered $\ell + 1, \ldots, L$ respectively, where $\ell = \lceil 11 \log \log \lambda \rceil$, and $L = \lceil \log N \rceil$ is the maximal level.

- The dictionary D is an oblivious dictionary storing $2^{\ell+1}$ elements.
- Each level $i \in \{\ell+1, \ldots, L\}$ consists of an instance, called T_i, of the oblivious hash table CombHT from Section 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 $\mathsf{data} \in \{0,1\}^w$.
Secret state: The dictionary D, levels $T_{\ell+1}, \ldots, T_L$, the bits $\mathsf{full}_{\ell+1}, \ldots, \mathsf{full}_L$ and counter ctr.
The algorithm:

1. Initialize $\mathsf{found} := \mathsf{false}$, $\mathsf{data}^* := \perp$.
2. Perform $\mathsf{fetched} := D.\mathsf{Lookup}(\mathsf{addr})$. If $\mathsf{fetched} \neq \perp$, then set $\mathsf{found} := \mathsf{true}$.
3. For each $i \in \{\ell+1, \ldots, L\}$ in increasing order, do:
   (a) If $\mathsf{found} = \mathsf{false}$:
      i. Run $\mathsf{fetched} := T_i.\mathsf{Lookup}(\mathsf{addr})$ with the following modifications:
         – Do not visit the stash of OF.
         – Do not visit the stash of CombS.
         (below, these stashes $(\mathsf{OF_S}, \mathsf{CombS_S})$ are merged into previous levels.)
      ii. If $\mathsf{fetched} \neq \perp$, let $\mathsf{found} := \mathsf{true}$ and $\mathsf{data}^* := \mathsf{fetched}$.
   (b) Else, $T_i.\mathsf{Lookup}(\perp)$.
4. If $\mathsf{found} = \mathsf{false}$, i.e., this is the first time addr is being accessed, set $\mathsf{data}^* = 0$.
5. Let $(k, v) := \{(\mathsf{addr}, \mathsf{data}^*)\}$ if this is a read operation; else let $(k, v) := \{(\mathsf{addr}, \mathsf{data})\}$. Insert $(k, (\ell, \perp, v))$ into oblivious dictionary D using $D.\mathsf{Insert}(k, (\ell, \perp, v))$.
6. Increment ctr by 1. If $\mathsf{ctr} \equiv 0 \mod 2^\ell$, perform the following.
   (a) Let j be the smallest level index such that $\mathsf{full}_j = 0$ (i.e., available). If all levels are marked full, then $j := L$. In other words, j is the target level to be rebuilt.
   (b) Let $\mathbf{U} := D.\mathsf{Extract}() \| T_{\ell+1}.\mathsf{Extract}() \| \ldots \| T_{j-1}.\mathsf{Extract}()$ and set $j^* := j-1$. If all levels are marked full, then additionally let $\mathbf{U} := \mathbf{U} \| T_L.\mathsf{Extract}()$ and set $j^* := L$. (Here, $\mathsf{Extract}()$ of CombHT does not extract the element from the stashes.)
   (c) Run $\mathsf{Intersperse}^{(j^*-\ell)}_{2^{\ell+1}, 2^{\ell+1}, 2^{\ell+2}, \ldots, 2^{j^*}}(\mathbf{U})$ (see Section 3). 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 $\tilde{\mathbf{U}} \leftarrow \mathsf{InterspreseRD}_N(\tilde{\mathbf{U}})$ to get a permuted $\mathbf{U}$.
   (d) Rebuild the jth hash table with the 2^j elements from $\tilde{\mathbf{U}}$ via $T_j := \mathsf{CombHT.Build}(\tilde{\mathbf{U}})$ and let $\mathsf{OF_S}, \mathsf{CombS_S}$ be the associated stashes (of size $O(\log \lambda)$ each). Mark $\mathsf{full}_j := 1$.
      i. For each element (k, v) in the stash $\mathsf{OF_S}$, run $D.\mathsf{Insert}(k, v)$.
      ii. For each element (k, v) in the stash $\mathsf{CombS_S}$, run $D.\mathsf{Insert}(k, v)$.
   (e) For $i \in \{\ell+1, \ldots, j-1\}$, reset T_i to be empty structure and set $\mathsf{full}_i := 0$.

Output: Return data^*.

The next theorem is proven in the full version [5]

Theorem 7.2. *Let $N \in \mathbb{N}$ be the capacity of ORAM and $\lambda \in \mathbb{N}$ be a security parameter. Assume a $\delta^{\mathcal{A}}_{\mathsf{PRF}}$-secure PRF. For any number of queries $T = T(N, \lambda) \geq N$, Construction 7.1 $(1 - T \cdot N^2 \cdot e^{-\Omega(\log \lambda \cdot \log \log \lambda)} - \delta^{\mathcal{A}}_{\mathsf{PRF}})$-obliviously implements the ORAM functionality. Moreover, the construction has $O\left(\log N \cdot \left(1 + \frac{\log N}{\log^2 \lambda}\right) + \log^9 \log \lambda\right)$ amortized time overhead.*

Acknowledgments

We are grateful to Hubert Chan, Kai-Min Chung, Yue Guo, and Rafael Pass for helpful discussions. This work is supported in part by a Simons Foundation junior fellow award awarded to G.A., an AFOSR Award FA9550-18-1-0267, NSF grant CNS-1601879, a DARPA Brandeis award, a Packard Fellowship, a Sloan Fellowship, Google Faculty Research Awards, a VMware Research Award, and a Baidu Research Award. G.A. and I.K. were with Cornell Tech during most this research.

References

1. https://en.wikipedia.org/wiki/Sorting_network.
2. Miklós Ajtai, János Komlós, and Endre Szemerédi. An $O(n \log n)$ sorting network. In *ACM STOC*, pages 1–9, 1983.
3. Arne Andersson, Torben Hagerup, Stefan Nilsson, and Rajeev Raman. Sorting in linear time? In *ACM STOC*, pages 427–436, 1995.
4. S. Arora, T. Leighton, and B. Maggs. On-line algorithms for path selection in a nonblocking network. In *ACM STOC*, 1990.
5. Gilad Asharov, Ilan Komargodski, Wei-Kai Lin, Kartik Nayak, Enoch Peserico, and Elaine Shi. Optorama: Optimal oblivious RAM. *IACR Cryptology ePrint Archive*, 2018:892, 2018.
6. 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, 1968.
7. Vincent Bindschaedler, Muhammad Naveed, Xiaorui Pan, XiaoFeng Wang, and Yan Huang. Practicing oblivious access on cloud storage: the gap, the fallacy, and the new way forward. In *ACM CCS*, pages 837–849, 2015.
8. Elette Boyle and Moni Naor. Is there an oblivious RAM lower bound? In *ACM ITCS*, pages 357–368, 2016.
9. Karl Bringmann, Fabian Kuhn, Konstantinos Panagiotou, Ueli Peter, and Henning Thomas. Internal DLA: Efficient Simulation of a Physical Growth Model. In *ICALP*, pages 247–258, 2014.
10. T.-H. Hubert Chan, Yue Guo, Wei-Kai Lin, and Elaine Shi. Oblivious hashing revisited, and applications to asymptotically efficient ORAM and OPRAM. In *ASIACRYPT*, pages 660–690, 2017.
11. T.-H. Hubert Chan, Yue Guo, Wei-Kai Lin, and Elaine Shi. Cache-oblivious and data-oblivious sorting and applications. In *SODA*, pages 2201–2220, 2018.

12. T.-H. Hubert Chan, Kartik Nayak, and Elaine Shi. Perfectly secure oblivious parallel RAM. In *TCC*, pages 636–668, 2018.
13. T.-H. Hubert Chan and Elaine Shi. Circuit OPRAM: unifying statistically and computationally secure orams and oprams. In *TCC*, pages 72–107, 2017.
14. Jean claude Paul and Wilhelm Simon. Decision trees and random access machines. 1980.
15. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*, pages 428–436. MIT Press, third edition, 2009.
16. Ivan Damgård, Sigurd Meldgaard, and Jesper Buus Nielsen. Perfectly secure oblivious RAM without random oracles. In *TCC*, pages 144–163, 2011.
17. Alireza Farhadi, MohammadTaghi Hajiaghayi, Kasper Green Larsen, and Elaine Shi. Lower bounds for external memory integer sorting via network coding. In *ACM STOC*, 2019.
18. P Feldman, J Friedman, and N Pippenger. Non-blocking networks. In *ACM STOC*, pages 247–254, 1986.
19. Christopher W Fletcher, Marten van Dijk, and Srinivas Devadas. A secure processor architecture for encrypted computation on untrusted programs. In *Seventh ACM workshop on Scalable trusted computing*, pages 3–8. ACM, 2012.
20. Christopher W. Fletcher, Ling Ren, Albert Kwon, Marten van Dijk, and Srinivas Devadas. Freecursive ORAM: [nearly] free recursion and integrity verification for position-based oblivious RAM. In *ACM ASPLOS*, pages 103–116, 2015.
21. Craig Gentry, Shai Halevi, Charanjit Jutla, and Mariana Raykova. Private database access with he-over-oram architecture. In *International Conference on Applied Cryptography and Network Security*, pages 172–191. Springer, 2015.
22. Oded Goldreich. Towards a theory of software protection and simulation by oblivious rams. In *ACM STOC*, pages 182–194, 1987.
23. Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious RAMs. *J. ACM*, 43(3):431–473, May 1996.
24. Michael T. Goodrich. Zig-zag Sort: A Simple Deterministic Data-oblivious Sorting Algorithm Running in O(N Log N) Time. In *ACM STOC*, pages 684–693, 2014.
25. Michael T. Goodrich and Michael Mitzenmacher. Privacy-preserving access of outsourced data via oblivious RAM simulation. In *ICALP*, pages 576–587, 2011.
26. Michael T. Goodrich, Michael Mitzenmacher, Olga Ohrimenko, and Roberto Tamassia. Oblivious ram simulation with efficient worst-case access overhead. In *Proceedings of the 3rd ACM Workshop on Cloud Computing Security Workshop*, CCSW '11, page 95–100, 2011.
27. Michael T. Goodrich, Michael Mitzenmacher, Olga Ohrimenko, and Roberto Tamassia. Privacy-preserving group data access via stateless oblivious RAM simulation. In *SODA*, pages 157–167, 2012.
28. Torben Hagerup and Hong Shen. Improved nonconservative sequential and parallel integer sorting. *Inf. Process. Lett.*, 36(2):57–63, 1990.
29. Eyal Kushilevitz, Steve Lu, and Rafail Ostrovsky. On the (in)security of hash-based oblivious RAM and a new balancing scheme. In *SODA*, pages 143–156, 2012.
30. Kasper Green Larsen and Jesper Buus Nielsen. Yes, there is an oblivious RAM lower bound! In *CRYPTO*, pages 523–542, 2018.
31. Frank Thomson Leighton, Yuan Ma, and Torsten Suel. On probabilistic networks for selection, merging, and sorting. *Theory Comput. Syst.*, 30(6):559–582, 1997.
32. Zongpeng Li and Baochun Li. Network coding : The case of multiple unicast sessions. 2004.
33. Wei-Kai Lin, Elaine Shi, and Tiancheng Xie. Can we overcome the $n \log n$ barrier for oblivious sorting? In *SODA*, 2019.

34. Chang Liu, Xiao Shaun Wang, Kartik Nayak, Yan Huang, and Elaine Shi. ObliVM: A programming framework for secure computation. In *IEEE S&P*, 2015.
35. Steve Lu and Rafail Ostrovsky. Distributed oblivious RAM for secure two-party computation. In *TCC*, pages 377–396, 2013.
36. Martin Maas, Eric Love, Emil Stefanov, Mohit Tiwari, Elaine Shi, Krste Asanovic, John Kubiatowicz, and Dawn Song. PHANTOM: practical oblivious computation in a secure processor. In *ACM CCS*, pages 311–324, 2013.
37. Grigorii Aleksandrovich Margulis. Explicit constructions of concentrators. *Problemy Peredachi Informatsii*, 9(4):71–80, 1973.
38. John C. Mitchell and Joe Zimmerman. Data-oblivious data structures. In *STACS*, pages 554–565, 2014.
39. Rafail Ostrovsky and Victor Shoup. Private information storage. In *ACM STOC*, pages 294–303, 1997.
40. Sarvar Patel, Giuseppe Persiano, Mariana Raykova, and Kevin Yeo. Panorama: Oblivious RAM with logarithmic overhead. In *IEEE FOCS*, 2018.
41. Sarvar Patel, Giuseppe Persiano, and Kevin Yeo. CacheShuffle: A Family of Oblivious Shuffles. In *ICALP*, pages 161:1–161:13, 2018.
42. Mark S. Pinsker. On the complexity of a concentrator. In *7th International Teletraffic Conference*, 1973.
43. Nicholas Pippenger. Superconcentrators. *SIAM J. Comput.*, 6(2):298–304, 1977.
44. Nicholas Pippenger. Self-routing superconcentrators. *J. Comput. Syst. Sci.*, 52(1):53–60, 1996.
45. Ling Ren, Xiangyao Yu, Christopher W. Fletcher, Marten van Dijk, and Srinivas Devadas. Design space exploration and optimization of path oblivious RAM in secure processors. In *ACM ISCA*, pages 571–582, 2013.
46. Elaine Shi, T.-H. Hubert Chan, Emil Stefanov, and Mingfei Li. Oblivious RAM with $O((\log N)^3)$ worst-case cost. In *ASIACRYPT*, pages 197–214, 2011.
47. Emil Stefanov and Elaine Shi. Oblivistore: High performance oblivious cloud storage. In *IEEE S&P*, pages 253–267, 2013.
48. Emil Stefanov, Elaine Shi, and Dawn Xiaodong Song. Towards practical oblivious RAM. In *NDSS*, 2012.
49. Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher W. Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: an extremely simple oblivious RAM protocol. In *ACM CCS*, pages 299–310, 2013.
50. Leslie G. Valiant. Graph-theoretic properties in computational complexity. *J. Comput. Syst. Sci.*, 13(3):278–285, December 1976.
51. Xiao Wang, T.-H. Hubert Chan, and Elaine Shi. Circuit ORAM: on tightness of the goldreich-ostrovsky lower bound. In *ACM CCS*, pages 850–861, 2015.
52. Xiao Shaun Wang, Yan Huang, T.-H. Hubert Chan, Abhi Shelat, and Elaine Shi. SCORAM: oblivious RAM for secure computation. In *ACM CCS*, pages 191–202, 2014.
53. Peter Williams, Radu Sion, and Alin Tomescu. Privatefs: A parallel oblivious file system. In *ACM CCS*, 2012.
54. Samee Zahur, Xiao Shaun Wang, Mariana Raykova, Adria Gascón, Jack Doerner, David Evans, and Jonathan Katz. Revisiting square-root ORAM: efficient random access in multi-party computation. In *IEEE S&P*, pages 218–234, 2016.