PanORAMa: Oblivious RAM with Logarithmic Overhead

Sarvar Patel*[1], Giuseppe Persiano†[1,2], Mariana Raykova‡[1,3], and Kevin Yeo§[1]

[1]Google LLC
[2]Università di Salerno
[3]Yale University

Abstract

We present PanORAMa, the first Oblivious RAM construction that achieves communication overhead $O(\log N \cdot \log\log N)$ for database of N blocks and for any block size $B = \Omega(\log N)$ while requiring client memory of only a constant number of memory blocks. Our scheme can be instantiated in the "balls and bins" model in which Goldreich and Ostrovsky [JACM 96] showed an $\Omega(\log N)$ lower bound for ORAM communication.

Our construction follows the hierarchical approach to ORAM design and relies on two main building blocks of independent interest: a *new oblivious hash table construction* with improved amortized $O(\log N + \mathsf{poly}(\log\log\lambda))$ communication overhead for security parameter λ and $N = \mathsf{poly}(\lambda)$, assuming its input is randomly shuffled; and a complementary *new oblivious random multi-array shuffle construction*, which shuffles N blocks of data with communication $O(N \log\log\lambda + \frac{N \log N}{\log \lambda})$ when the input has a certain level of entropy. We combine these two primitives to improve the shuffle time in our hierarchical ORAM construction by avoiding heavy oblivious shuffles and leveraging entropy remaining in the merged levels from previous shuffles. As a result, the amortized shuffle cost is asymptotically the same as the lookup complexity in our construction.

*sarvar@google.com
†giuper@gmail.com
‡mariana.raykova@yale.edu
§kwlyeo@google.com

1 Introduction

The cryptographic primitive of *Oblivious RAM* (ORAM) considers the question of how to enable a client to outsource its database to an untrusted server and to query it subsequently without any privacy leakage related to the queries and the database. While encryption can help with hiding the outsourced content, a much more challenging question is how to hide the leakage from the access patterns induced by the queries' execution. This is leakage that is not only ruled out by the strong formal definition of privacy preserving data outsourcing but has also been proven to be detrimental in many practical outsourcing settings [25, 6]. Hiding access patterns is only interesting when coupled with efficiency guarantees for the query execution in the following sense: there is a trivial access hiding solution which requires linear scan of the whole data at every access. Once such efficiency properties are in place, ORAM constructions also have another extremely important application as they are a critical component for secure computation solutions that achieve sublinear complexity in their input size [31, 24, 40].

Thus, the study of ORAM constructions has been driven by the goal of improving their bandwidth overhead per access while providing hiding properties for the access patterns. This study has been a main research area in Cryptography for the past thirty years, since the notion was introduced by Goldreich [18], and it has turned ORAM into one of the classical cryptographic concepts. The seminal works of Goldreich and Ostrovsky [18, 30, 19] introduced the first ORAM constructions achieving square root and polylogarithmic amortized query efficiency. These early works also considered the question of a lower bound on the amortized communication complexity required to maintain obliviousness. They presented a lower bound result of $O(\log_C N)$ blocks of bandwidth overhead for any ORAM construction for a database of size N blocks and a client with memory that can store C blocks. However, this result came with a few caveats, which were clarified and more carefully analyzed in the recent work by Boyle and Naor [5]. A recent work of Larsen and Nielsen [27] removed these caveats in the online model and presented an $\Omega(\log N)$ block communication lower bound for general storage models and computationally bounded adversaries.

ORAM Communication Lower Bound. The ORAM communication lower bound presented by Goldreich and Ostrovsky [19] applied to constructions using restricted manipulation on the underlying data (i.e., treating the data as monolithic blocks that are read from, written to and moved between different memory positions), achieving statistical security and any block size. Boyle and Naor [5] formalized the model of this lower bound as the "balls and bins" storage model and gave much more insight into understanding the lower bound of Goldreich and Ostrovsky. They provided evidence that extending the lower bound beyond the restricted model in the original result will be a challenging task by showing a reduction from sorting circuits to offline ORAM where all queries are given ahead of time, which essentially means that extending the offline ORAM lower bound will imply new lower bounds for sorting circuits. At the same time, Boyle and Naor introduced an online model for ORAM where queries are selected adaptively during execution. This model avoids the relation to the lower bound on sorting circuits while reflecting the functionality of most existing ORAM constructions, which opened the possibility that improving the lower bound in the online model might be easier than in the offline setting. The recent work of Larsen and Nielsen [27] proved the best ORAM lower bound which applies to the online model with computational guarantees, and for any general storage model. However, the server is assumed to act only as storage and it is known that allowing server-side computation can bypass the lower bound [14].

As we discussed above, the lower bound results apply to all algorithms that work for any block size. While there are constructions [38, 40] that match the lower bound in regimes when they are instantiated with appropriately big block sizes, there is no known result that meets the lower bound for every block size $\Omega(\log N)$.

ORAM Constructions. While the lower bound of Goldreich and Ostrovsky has its caveats, it has become the measure to compare with for every new construction with improved complexity. We next overview the known ORAM constructions, their models and efficiency with respect to the requirements for the lower bound. Existing ORAM schemes could be roughly divided into two categories: constructions [35, 1, 13, 22, 26] that follow the hierarchical blueprint introduced in the work of Goldreich and Ostrovsky [30, 19], and constructions [37, 17, 38, 12, 40, 36, 7] that follow the tree-based template with a recursive position map, which was introduced in the work of Shi et al. [37].

The idea underlying the first class of constructions is to divide the data in levels of increasing size that form a hierarchy and to instantiate each level with an oblivious access structure that allows each item to be accessed obliviously only once in that level. The smallest level in the hierarchy is linearly scanned at each access and each block that is accessed is subsequently moved to this level. To prevent overflowing of the top levels, there is a deterministic schedule, independent of the actual accesses, that prescribes how blocks move from the smaller to the bigger levels.

Within the general hierarchical framework, the main optimization question considered in different constructions is how to instantiate the oblivious structure in each level. The original Goldreich-Ostrovsky construction [19] used pseudorandom functions (PRFs) resulting in $O(\log^3 N)$ amortized communication overhead. Later, the work of Pinkas and Reinman [35] proposed the use of Cuckoo hash tables. This work suffered from a subtle issue related to the obliviousness of the Cuckoo hash tables, which was later fixed in the work of Goodrich and Mitzenmacher [22] who showed an elegant algorithm to obliviously construct a Cuckoo hash table. Their main argument was a reduction of the Cuckoo hash table construction to oblivious sorting that resulted in a $O(\log^2 N)$-overhead ORAM. Subsequently, Kushilevitz et al. [26] devised a balancing scheme that further improved the bandwidth to $O(\log^2 N/\log\log N)$. The work of Chan et al. [7] presented a unified framework for hierarchical ORAM constructions and made explicit the notion of an oblivious hash table in order to capture the properties of the oblivious structure needed for each level. All the above constructions require that the client's memory can hold only a constant number of blocks.

The known hierarchical ORAM constructions do not make any assumptions about the block sizes used to store data in memory and can be instantiated with any block size $B = \Omega(\log N)$. Even the construction with best asymptotic efficiency among existing schemes, when instantiated with a private random function in the balls and bins model, does not meet the logarithmic lower bound in the case of client's memory that holds only a constant number of blocks.

The other main construction template for ORAM schemes leverages the idea of mapping blocks to random position map (PMAP) indices and then arranging the data in a binary tree with leaves indexed according to the position map, where each block can reside only in a node on the path to its corresponding PMAP leaf. Thus, in order to access a block it is sufficient to read the path indexed by its PMAP value. In order to access efficiently the PMAP for each query, the construction stores recursively the position map by partitioning it into blocks each of which contains at least two PMAP indices. After the recursion the construction consists of a logarithmic number of trees of decreasing size. Every time a block is accessed, it is assigned a new PMAP index and is moved to the root of the tree. In order to prevent overflowing of nodes, the constructions periodically evict blocks down their corresponding tree paths. The main difference between different tree-based ORAMs is related to the concrete eviction algorithms they use, which have been evolving and improving the ORAM access overhead. The work of Shi et al. [37] that pioneered the tree-based approach achieved $O(\log^3 N)$ communication. Gentry *et al.* [17] improved the overhead to $O(\frac{\log^3 N}{\log\log N})$. Currently the most efficient tree-based construction is the Path ORAM construction [38], which achieves $O(\log^2 N)$ bandwidth overhead for general block sizes. If instantiated with blocks of size $\Omega(\log^2 N)$, Path ORAM has bandwidth overhead of $O(\log N)$ blocks. The same efficiency holds for Circuit ORAM [40], which optimizes circuit sizes for ORAM access functionality in secure computation.

The only computational assumption of the above tree-based constructions is related to the encryption used to hide the content and thus they do offer statistical guarantees in the balls and bins model. Several works [14, 28] also demonstrate how to bypass the lower bound of communication complexity, if the server is allowed to do computation on the data it stores, which is enabled by homomorphic encryption.

Our Contributions. In this paper we present PanORAMa, a computationally secure oblivious RAM with $O(\log N \cdot \log\log N)$ bandwidth overhead and constant client memory[1]. Our construction works for any block size $B = \Omega(\log N)$. This assumption is very natural since all known ORAM constructions including ours require that the blocks store their own addresses for correctness and this already takes $\Theta(\log N)$ bits. In addition, PanORAMa is in the balls and bins model of Boyle and Naor [5] as it treats each data block as an atomic piece of data and the server only fetches blocks from memory, writes blocks to memory and moves blocks between different memory positions. Thus, PanORAMa achieves currently the best asymptotic communication overhead among constructions that work with general block sizes, operate in the balls and bins model and require constant number of blocks client memory.

Our construction can be modified in a straightforward way to obtain statistical security in the balls and bins model if the client is provided with access to a private random function, which matches the assumptions in the original lower bound (see Theorem 6 in [19]). In this case, we obtain a balls and bins construction that is only $O(\log\log N)$ away from the lower bound overhead proven by Goldreich and Ostrovsky [19]. Our construction is also $O(\log\log N)$ away from the lower bound by Larsen and Nielsen [27] for general storage models and computational adversaries. As a result, we show that the balls and bins model of computation is almost as strong as any general storage model and can only require at most $O(\log\log N)$ extra communication overhead.

The PanORAMa construction relies on two main building blocks, which can have applications outside ORAM of independent interest: an *oblivious hash table (OHT)* and an *oblivious random multi-array shuffle algorithm.* For both, we provide new efficient constructions. Specifically,

- *Oblivious Hash Table (OHT).* An oblivious hash table offers the same functionalities as a regular hash table (efficient storage and access) while guaranteeing access obliviousness for non-repeating patterns. We extend the definition of OHT [7] that consists of initialization and query algorithms as follows. We split the initialization into Init, which shuffles the input items and inserts dummies, and Build, which uses the output of Init to create the OHT storage structure. We add an algorithm Extract, which obliviously returns all unqueried items from the OHT appropriately padded. Our OHT construction offers an amortized access efficiency of $O(\log N + \log\log\lambda)$ blocks assuming the starting data is randomly shuffled, where λ is the security parameter.

[1] We measure bandwidth and client memory using the size B of a block as a unit.

- *Oblivious Random Multi-Array Shuffle.* Complementary to the OHT primitive is our efficient oblivious random multi-array shuffle algorithm that shuffles together data which initial order has partial entropy with respect to the adversary. More precisely, our algorithm shuffles together $A_1, \ldots, A_L$ arrays of total size N, each of which is independently and randomly shuffled. Suppose that the L arrays are arranged in decreasing size. Then, our shuffle requires $O(N \log\log\lambda + \frac{N\log N}{\log\lambda})$ blocks of communication when there exists $\mathsf{cutoff} = O(\log\log\lambda)$ such that $|\mathsf{A}_{\mathsf{cutoff}}| + \ldots + |\mathsf{A}_L| = O(\frac{N\log\log\lambda}{\log N})$.

Technical Overview of Our Result. Our ORAM construction follows the general paradigm of hierarchical ORAM constructions as laid out by Goldreich and Ostrovsky [19] (see Chan et al. [7] for a formalized presentation of the framework). As we discussed above, the hierarchical constructions distribute the data in several levels, which are instantiated with oblivious hash tables that provide access obliviousness for non-repeating queries.

In order to prevent overflow of the OHTs implementing the ORAM levels, every 2^i accesses, all levels of size less or equal than 2^i are merged and shuffled together and placed in an oblivious data structure in the level of capacity 2^{i+1}. While a level of capacity $2^j \leq 2^i$ services exactly 2^j queries before shuffling, the number of real items retrieved from this level can range from 0 to 2^j. All queried items have been moved to smaller levels and thus should not be included in the shuffle as items coming from this level. The remaining at most 2^j unqueried items in the oblivious data structure need to be extracted and included in the larger capacity level for future queries. As a result, the shuffle step can be broken down into three phases: extracting unqueried items from each level, merging the content of multiple levels and initializing a oblivious hash table for the new level. For many existing hierarchical ORAMs, the dominant cost in the communication complexity arises from to the use of several oblivious sorts that are used to implement the shuffling functionality while removing queried items. The best known data-oblivious sorting algorithms [39, 20, 21, 8] used in these constructions require communication $O(N \log N)$. In our work, we show that all three shuffle phases can be achieved without the use of expensive oblivious sorts by leveraging and maintaining entropy from previous shuffles, which is manifested in the fact that the unqueried items in each level are essentially "randomly shuffled". Similar ideas were previously explored for simpler scenarios in [34].

The first phase for the ORAM shuffle step is the OHT extraction for all shuffled levels. Consider the extract step for level $j \leq i$ during a shuffle of levels from 1 to i. Recall that the OHT at each level offers oblivious access for any sequence of non-repeating queries. This is typically achieved by obliviously shuffling all real items and $\Theta(2^j)$ dummy items together during initialization. As a result, the remaining unqueried real and dummy items persist in some obliviously shuffled manner. We use this remaining entropy of the unqueried items to construct an OHT extraction algorithm that efficiently extracts 2^j items consisting of all unqueried real items and a sufficient number of unqueried dummy items in a random order oblivious to the adversary without the use of expensive oblivious sorts. It suffices to only extract unqueried items as all queried items and their possibly updated version will be appended to the smallest level after querying. For the smallest level, all items must be extracted as well as deduplicated. We use an oblivious sort for this since the smallest level will only contain $O(\log N)$ items. The OHT extraction mechanism on each level entering the shuffle pads the unqueried items with dummy items up to the total capacity of the OHT. Thus, the extracted items from each level will have different numbers of dummies. However, when we add all extracted items from all levels, we will have equal numbers of dummy and real items, which is exactly the distribution we need for the new level that will be initialized after the shuffle. We discuss the intuition for efficient extraction in the overview of our OHT construction in Section 4.

The next step of the shuffle is to obliviously merge all extracted items from multiple levels. One way to achieve this is using an oblivious sort over all items, which would require $O(N \log N)$ for the largest levels. Oblivious sorting achieves very strong hiding guarantees even against an adversary that knows the entire initial order of the data. However, in the case of the ORAM shuffle mixing together the items from the shuffled ORAM levels, we have the additional leverage that the inputs for the shuffle coming from the extraction of each OHT at each level are already randomly shuffled arrays. More specifically these are multiple arrays of geometrically decreasing size where each array is randomly ordered in a manner oblivious to the adversary. We design an oblivious random multi-array shuffle that obliviously merges the randomly ordered input arrays into a single array, which is a random shuffle of all elements. This algorithm leverages the entropy coming from the random shuffles of each input array to avoid the cost of expensive oblivious sort. We discuss the intuition behind this algorithm in the overview our multi-array shuffle in Section 3.

The final phase in the ORAM step shuffle is the OHT initialization for level $i+1$ using the randomly permuted array that is output from the multi-array shuffle. We manage to construct an efficient algorithm for the initialization that avoids oblivious shuffles of the whole input by crucially relying on the fact that the input is already randomly shuffled. We further discuss the intuition for the initialization algorithm of our OHT construction in Section 4.

Oblivious Hash Table. Our oblivious hash table construction is inspired by the two-tier hash scheme proposed by Chan et al. [9], however, with some significant changes that enable constructing the OHT without using an oblivious sort on all the data blocks. The idea of Chan et al. [9] is to allocate the database items into bins on the first level of the hash table using a PRF, where the size of the bins is set to be $O(\log^{\delta} \lambda)$, for some constant $0 \leq \delta < 1$, which does not guarantee non-negligible overflow probability. All overflow items are allocated to a second level where they are distributed using a second PRF. In order to initialize this two-tier hash scheme, the authors use an oblivious sort which comes at a cost $O(N \log N)$ for N blocks of data.

Our goal is to obtain a construction of an oblivious hash table that allows more efficient oblivious initialization assuming that the database items are already randomly shuffled. The assumption of the randomly shuffled input is not arbitrary. We will use our oblivious random multi-array shuffle to construct this random shuffle of the input in the context of our ORAM construction.

Our initialization algorithm sequentially distributes input items in $O(\log \log \lambda)$ levels. At each level, all remaining items are distributed into small bins according to a secret PRF where the bin sizes are not hidden. A secret distribution for each bin's real size is sampled and several small oblivious shuffles are employed to remove additional blocks from each bin for the next level. In more detail, we first assign items into buckets according to a PRF non-obliviously taking linear time in the size of the data. Then, we sample from a binomial distribution loads for all bins that correspond to randomly distributing only ϵ fraction of the total number of items. We choose a cutoff point, thrsh, such that with overwhelming probability, it is larger than any bin load sampled from the binomial distribution in the second step and, at the same time, is smaller than any load from the distribution of items induced by the PRF in the first step. We cut the size of each bin to exactly thrsh items, among which there will be as many real items as the loads sampled from the binomial distribution and the rest will be dummy items. The remaining overflow items are distributed recursively in following levels of the OHT where the size of the smallest level is $O(N/\log N)$. This step guarantees that the oblivious property for the query access patterns since they will be distributed according to the bin loads induced from the binomial samples, which are independent from the loads due to the PRF that the server observed in the clear.

The items assigned to each bin in each level of the OHT are instantiated with another oblivious hash table construction which we call *oblivious bin*. An oblivious bin is an OHT with small input size $O(\mathsf{poly}(\log \lambda))$ for which we can afford to use an oblivious sort for initialization and extraction without incurring prohibitive efficiency cost. In the full version [32] we present an instantiation for oblivious bins using a Cuckoo hash table.

The OHT query algorithm consists of one oblivious bin query in each level. The single bin in the smallest level is always queried. In every other level, we either query the bin determined by the corresponding PRF if the item has not been found yet, or query a random bin otherwise.

Last but not least, our oblivious hash tables have an oblivious extraction procedure that allows to separate the unqueried items in the OHT with just an overhead of $O(\log \log \lambda)$ per item. Additionally we guarantee that the extracted items are randomly shuffled and, thus, we can use them directly as input for our multi-array shuffle. The extraction procedure for our OHT can be done by implementing the extraction on each of the oblivious bins and concatenating the outputs, since the items were distributed to bins using a secret distribution function. We obliviously extract each bin using an oblivious sort.

Oblivious Random Multi-Array Shuffle. Our multi-array random shuffle relies on the observation that we do not need to hide the access pattern within each of the input arrays since they are already shuffled. Recall that in the context of ORAM, these input arrays represent the unqueried items extracted from OHTs of smaller levels. Since our shuffle algorithm will be accessing each entry of each input array only once, its initial random shuffle suffices for the obliviousness of these accesses. However, the multi-array shuffle algorithm still needs to hide the interleaving accesses to the different input arrays. One way to achieve this is to obliviously shuffle the accesses to different input arrays. If we do this, in general, we will end up doing an oblivious shuffle on the whole input data, which is too expensive.

Instead, we partition the input arrays by distributing their items at random into a number of bins of size $O(\frac{\log^3 \lambda}{1-2\epsilon})$. We also partition the output array into bins of size $O(\log^3 \lambda)$, where each item of the output array is assigned an input array tag that is encrypted and remains hidden. With all but negligible probability each resulting input bin contains a sufficient number of items from each input array in order to initialize each output bin. The partitioning into input and output bins is performed non-obliviously but does not cause any additional leakage as the inputs arrays are shuffled and the input arrays tags for all output array items are encrypted.

We pair input and output bins and we use items from an input bin to initialize the items in the corresponding output bin using a sequence of oblivious sorts. In each such initialization, we also have a number of leftover real items that were not needed for the output bin (the sizes of the input and output bins were chosen in a way that guarantees that we always have at least as many items from each input array in the input bin as needed in the output segment). We apply the multi-array random shuffling algorithm recursively on the arrays containing leftover items from the executions filling different output bins in order to initialize the remaining output bins (there were more output bins than input bins but output bins had smaller sizes). After all items have been distributed from input bins to output bins using the above construction, we use to reverse mapping from output bins to the output array to place the items in the output array.

The intuition why the above approach helps us to improve our efficiency is that we are using oblivious sort on small arrays that are of size $O(\log^3 \lambda)$ and we perform $O(\frac{N}{\log^3 \lambda})$ of these shuffles.

Thus, the total shuffle cost remains $O(N \log\log \lambda)$.

Our resulting shuffling procedure achieves such efficiency that it is no longer the dominant cost in the amortized query complexity for our final ORAM construction. As a result, the optimization technique presented in the work of Kushilevitz *et al.* [26], which balances the cost of the lookups and the cost of the shuffle by splitting each level into several disjoint oblivious hash tables that get shuffled into separately, does not result in any efficiency improvement when applied to our scheme.

Paper Organization. We present in Section 2 the definitions of the primitives we use in the rest of the paper. Section 3 describes our construction of an oblivious random multi-array shuffle, which relies on permutation decomposition lemmas that provide alternative ways to sample randomly a permutation, which are presented in Appendix A. In Section 4 we describe our general oblivious hash table construction and its building block the oblivious bin primitive. Finally, we present our ORAM construction in Section 5. In Appendix B we present an additional overview of related work.

2 Definitions

In this section we present the definitions for the existing primitives that we use for our constructions as well as definitions of the new primitives that we introduce in our work.

Notation. We denote $\mathsf{Binomial}[n, p]$ the binomial distribution with parameters: n trials each of which with success probability p. We use $X \leftarrow \mathsf{Binomial}[n, p]$ to denote that the variable X is sampled from the binomial distribution according to its probability mass function $\mathsf{Pr}(k; n, p) = \mathsf{Pr}[X = k] = \binom{n}{k} p^k (1-p)^{n-k}$. In a setting where an algorithm Alg is executing using external memory, we denote by $\mathsf{Addrs}[\mathsf{Alg}]$ the memory access pattern that consists of all accessed addresses in the memory. We use PPT as a shorthand for "probabilistic polynomial-time."

In analyzing our constructions, we express bandwidth and client memory using the size B of a block as a unit.

2.1 Oblivious RAM

Definition 1 (Oblivious RAM)**.** An oblivious RAM scheme $\mathsf{ORAM} = (\mathsf{ORAM.Init}, \mathsf{ORAM.Access})$ consists the following two algorithms:

- $(\tilde{D}, \mathsf{st}) \leftarrow \mathsf{ORAM.Init}(1^\lambda, D)$: the initialization algorithm takes as input a database D and outputs a initialized memory structure $\tilde{D}$.
- $(v, \mathsf{st}') \leftarrow \mathsf{ORAM.Access}(\mathsf{st}, \tilde{D}, \mathsf{I})$: the ORAM access algorithm takes as input the ORAM database $\tilde{D}$, the current state st as well as an instruction $\mathsf{I} = (\mathsf{op}, \mathsf{addr}, \mathsf{data})$, where $\mathsf{op} \in \{\mathsf{read}, \mathsf{write}\}$ and if $\mathsf{op} = \mathsf{read}$, then $\mathsf{data} = \perp$. If $\mathsf{op} = \mathsf{read}$, the access algorithm returns as v the data stored at address addr in the database. Else, if $\mathsf{op} = \mathsf{write}$ the access algorithm writes data in location addr in the database. Furthermore, an updated state st' is returned.

The resulting construction is oblivious if there exists a PPT simulator $\mathsf{Sim} = (\mathsf{Sim}_{\mathsf{Init}}, \mathsf{Sim}_{\mathsf{Access}})$ such that for any PPT adversarial algorithm $\mathcal{A}$ and for any $n = \mathsf{poly}(\lambda)$,

$$\left| \mathsf{Pr}\left[b = 1 \mid b \leftarrow \mathbf{Expt}_{\mathcal{A}}^{\mathsf{Real},\mathsf{ORAM}}(\lambda, n) \right] - \mathsf{Pr}\left[b' = 1 \mid b' \leftarrow \mathbf{Expt}_{\mathsf{Sim},\mathcal{A}}^{\mathsf{Ideal},\mathsf{ORAM}}(\lambda, n) \right] \right| < \mathsf{negl}(\lambda),$$

where the real and ideal executions are defined as follows:

$\mathbf{Expt}_{\mathcal{A}}^{\mathsf{Real,ORAM}}(\lambda, n)$	$\mathbf{Expt}_{\mathsf{Sim},\mathcal{A}}^{\mathsf{Ideal,ORAM}}(\lambda, n)$
$(D, \mathsf{st}_{\mathcal{A}}) \leftarrow \mathcal{A}_1(1^\lambda)$	$(D, \mathsf{st}_{\mathcal{A}}) \leftarrow \mathcal{A}_1(1^\lambda)$
$\chi \leftarrow \mathsf{Addrs}[(\tilde{D}, \mathsf{st}) \leftarrow \mathsf{ORAM.Init}(1^\lambda, D)]$	$\chi \leftarrow \mathsf{Addrs}[(\tilde{D}, \mathsf{st}_{\mathsf{Sim}}) \leftarrow \mathsf{Sim}_{\mathsf{Init}}(1^\lambda, \|D\|)]$
for $j = 1$ to n	for $i = 1$ to n
$(\mathsf{I}_j, \mathsf{st}_{\mathcal{A}}) \leftarrow \mathcal{A}_2(\mathsf{st}_{\mathcal{A}}, \tilde{D}, \chi)$	$(\mathsf{I}_j, \mathsf{st}_{\mathcal{A}}) \leftarrow \mathcal{A}_2(\mathsf{st}_{\mathcal{A}}, \tilde{D}, \chi)$
$\chi_{\mathsf{Access}} \leftarrow \mathsf{Addrs}[(\mathsf{st}, \tilde{D}) \leftarrow \mathsf{ORAM.Access}(\mathsf{st}, \tilde{D}, \mathsf{I}_j)]$	$\chi_{\mathsf{Access}} \leftarrow \mathsf{Addrs}[(\mathsf{st}, \tilde{D}) \leftarrow \mathsf{Sim}_{\mathsf{Access}}(\mathsf{st}_{\mathsf{Sim}})]$
$\chi \leftarrow \chi \cup \chi_{\mathsf{Access}}$	$\chi \leftarrow \chi \cup \chi_{\mathsf{Access}}$

2.2 Oblivious Random Multi-Array Shuffle

In oblivious shuffling, we have one array of N data blocks and the task of the algorithm is to shuffle the blocks into a destination array so that an adversary observing the accesses of the algorithm to the blocks does not obtain any information regarding the final permutation. The security guarantee holds even if the initial arrangement of the blocks is known to the adversary. This is sufficient to be used in the design of Oblivious RAM. An oblivious random multi-array shuffle instead offers a weaker security guarantee, which we show also suffices for the design of an Oblivious RAM. For a range of parameters of interest, we present an implementation with improved efficiency in terms of bandwidth overhead compared to oblivious shuffling. Roughly speaking, in an oblivious random multi-array random shuffle, we have N blocks partitioned into L arrays $\mathsf{A}_1, \ldots, \mathsf{A}_L$ and the task is to shuffle the N blocks into a destination array according to a permutation chosen uniformly at random. The associated security notion still guarantees that no information regarding the final permutation of the blocks is leaked. However, the adversary's knowledge is limited to the distribution of blocks to each array and does not include each block's specific location within the array. Let us now proceed more formally.

2.3 Oblivious Bin

We introduce a slight modification of our oblivious hash table definition, which we call *oblivious bin*. As the name implies, we will use this oblivious structure to store and access data in oblivious manner within bins which will be used as building blocks in out oblivious hash table scheme. We will use the oblivious bin for data of smaller size.

Definition 2 (Oblivious Bin)**.** An oblivious hash table scheme $\mathsf{OblivBin} = (\mathsf{OblivBin.Init}, \mathsf{OblivBin.Build}, \mathsf{OblivBin.Lookup}, \mathsf{OblivBin.Extract})$ consists of the following algorithm:

- $(\tilde{D}, \mathsf{st}) \leftarrow \mathsf{OblivBin.Init}(D)$: an algorithm that takes as input an array of key-value pairs $D = \{(k_i, v_i)\}_{i=1}^N$ and outputs a processed version of it $\tilde{D}$.
- $(\tilde{S}, \tilde{H}, \mathsf{st}') \leftarrow \mathsf{OblivBin.Build}(\tilde{D}, \mathsf{st})$: an algorithm that takes as input a processed database $\tilde{D}$ and a state and initializes the hash table $\tilde{H}$ and an additional array $\tilde{S}$ and updates the state st.
- $(v, \tilde{S}', \tilde{H}', \mathsf{st}') \leftarrow \mathsf{OblivBin.Lookup}(k, \tilde{H}, \tilde{S}, \mathsf{st})$: an algorithm that takes as input the oblivious hash table, the additional array, the state produced in the build stage and a lookup key, and outputs the value v_i corresponding to the key k_i together with updated hash table $\tilde{H}'$ and state st'. If the k is not found, then $v := \perp$.

$\mathbf{Expt}_{\mathcal{A}}^{\mathsf{Real,OblMultArrShuff}}(\lambda)$	$\mathbf{Expt}_{\mathsf{Sim},\mathcal{A}}^{\mathsf{Ideal,OblMultArrShuff}}(\lambda)$
$(\mathsf{A}_1, \ldots, \mathsf{A}_L, \mathsf{st}_{\mathcal{A}}) \leftarrow \mathcal{A}_1(1^\lambda)$	$(\mathsf{A}_1, \ldots, \mathsf{A}_L, \mathsf{st}_{\mathcal{A}}) \leftarrow \mathcal{A}_1(1^\lambda)$
for $i = 1$ to L	
$\tilde{\mathsf{A}}_i \leftarrow \tau_i(\mathsf{A}_i)$ where τ_i is a random permutation on $\lvert\mathsf{A}_i\rvert$ elements	
$\chi \leftarrow \mathsf{Addrs}[\mathsf{D} \leftarrow \mathsf{OblMultArrShuff}(\mathsf{A}_1, \ldots, \mathsf{A}_L)]$	$\chi \leftarrow \mathsf{Addrs}[\mathsf{D} \leftarrow \mathsf{Sim}(\lvert\mathsf{A}_1\rvert, \ldots, \lvert\mathsf{A}_L\rvert)]$
Let π be the induced permutation of the elements in $\mathsf{A}_1, \ldots, \mathsf{A}_L$ mapped to D	Let π' be a random permutation on $\sum_{i=1}^{L} \lvert\mathsf{A}_i\rvert$ elements
Output $b \leftarrow \mathcal{A}_2(\mathsf{D}, \pi, \mathsf{st}_{\mathcal{A}}, \chi)$	Output $b \leftarrow \mathcal{A}_2(\mathsf{D}, \pi', \mathsf{st}_{\mathcal{A}}, \chi)$

Figure 1: Real and ideal executions for $\mathsf{OblMultArrShuff}$.

- $(\tilde{D}, \mathsf{st}') \leftarrow \mathsf{OblivBin.Extract}(\tilde{H}, \tilde{S}, \mathsf{st})$: this is an algorithm that takes the hash table, the auxiliary array and the state after the execution of a number of queries and outputs a database, which contains only the unqueried items $(k_i, v_i) \in D$ and is padded to size N.

The oblivious property for a bin is identical to that of the oblivious hash table.

2.4 Oblivious Random Multi-Array Shuffle

We start with the definition of a random multi-array shuffling algorithm that obliviously shuffles together the content of several input array each of which is independently shuffled (see full version for more detailed discussion of the functionality).

Definition 3 (Oblivious Random Multi-Array Shuffle)**.** A *random multi-array shuffle* algorithm is an algorithm $\mathsf{D} \leftarrow \mathsf{OblMultArrShuff}(\mathsf{A}_1, \ldots, \mathsf{A}_L)$ that takes as input L arrays $\mathsf{A}_1, \ldots, \mathsf{A}_L$ containing a total of N blocks and outputs a destination array D that contain all N blocks. Each of the L arrays are assumed to have been arranged according to a permutation chosen uniformly at random. The blocks in D should be arranged according to a permutation chosen uniformly at random.

A random multi-array shuffle algorithm $\mathsf{OblMultArrShuff}$ is *oblivious* if there exists a PPT simulator Sim, which takes as input $(\lvert\mathsf{A}_1\rvert, \ldots, \lvert\mathsf{A}_L\rvert)$, such that for any PPT adversary algorithm $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$:

$$\left|\mathsf{Pr}\left[b = 1 \mid b \leftarrow \mathbf{Expt}_{\mathcal{A}}^{\mathsf{Real,OblMultArrShuff}}(\lambda)\right] - \mathsf{Pr}\left[b' = 1 \mid b' \leftarrow \mathbf{Expt}_{\mathsf{Sim},\mathcal{A}}^{\mathsf{Ideal,OblMultArrShuff}}(\lambda)\right]\right| < \mathsf{negl}(\lambda)$$

where the real and ideal executions are defined in Figure 1.

Intuitively, the definition above captures the security with respect to an adversary that does not know the original order of the items in the input arrays (i.e., each array is randomly shuffled) but is allowed to pick the partition of the N blocks across the L levels. We require that the adversary does not obtain any information about the final permutation of the N blocks and we formally capture this requirement by providing the adversary with the actual induced permutation in the real execution and a completely random permutation in the ideal execution. Thus, if the adversary cannot distinguish the real and the ideal experiment, it follows that it has not learned anything about the permutation by observing the access pattern leaked from the shuffle algorithm.

2.5 Oblivious Hash Table

Next we define our oblivious hash table, which provides oblivious access for non-repeating queries. In addition, it has an extraction algorithm which allows to extract obliviously the unqueried items remaining in the OHT in a randomly permuted order.

Definition 4 (Oblivious Hash Table)**.** An oblivious hash table scheme $\mathsf{OblivHT} = (\mathsf{OblivHT.Init}, \mathsf{OblivHT.Build}, \mathsf{OblivHT.Lookup}, \mathsf{OblivHT.Extract})$ consists of the following algorithms:

- $(\tilde{D}, \mathsf{st}) \leftarrow \mathsf{OblivHT.Init}(D)$: an algorithm that takes as input an array of key-value pairs $D = \{(k_i, v_i)\}_{i=1}^{N}$ and outputs a processed version of D, which we denote $\tilde{D}$.
- $(\tilde{H}, \mathsf{st}') \leftarrow \mathsf{OblivHT.Build}(\tilde{D}, \mathsf{st})$: an algorithm that takes as input a processed database $\tilde{D}$ and a state and initializes the hash table $\tilde{H}$ and updates the state st.
- $(v, \tilde{H}', \mathsf{st}') \leftarrow \mathsf{OblivHT.Lookup}(k, \tilde{H}, \mathsf{st})$: an algorithm that takes as input the oblivious hash table, $\tilde{H}$, the state produced in the build stage, st, and a lookup key, k, and outputs the value v_i corresponding to the key k_i together with an updated hash table, $\tilde{H}'$, and updated state, st'. If the k is a dummy query, then $v := \perp$.
- $(\tilde{D}, \mathsf{st}') \leftarrow \mathsf{OblivHT.Extract}(\tilde{H}, \mathsf{st})$: an algorithm that takes the hash table, $\tilde{H}$, and the state after the execution of a number of queries, st, and outputs a database $\tilde{D}$, which contains only the unqueried items $(k_i, v_i) \in D$ and is padded to size N with dummy items, and the content of $\tilde{D}$ is randomly permuted.

The resulting hash scheme is oblivious if there exists a PPT simulator $\mathsf{Sim} = (\mathsf{Sim}_{\mathsf{Init}}, \mathsf{Sim}_{\mathsf{Build}}, \mathsf{Sim}_{\mathsf{Lookup}}, \mathsf{Sim}_{\mathsf{Extract}})$, which takes as input the size of the database $|D|$, such that for any PPT adversary algorithm $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2, \mathcal{A}_3)$ and for any $n = \mathsf{poly}(\lambda)$,

$$\left|\mathsf{Pr}\left[b = 1 \mid b \leftarrow \mathbf{Expt}_{\mathcal{A}}^{\mathsf{Real},\mathsf{OblivHT}}(\lambda, n)\right] - \mathsf{Pr}\left[b' = 1 \mid b' \leftarrow \mathbf{Expt}_{\mathsf{Sim},\mathcal{A}}^{\mathsf{Ideal},\mathsf{OblivHT}}(\lambda, n)\right]\right| < \mathsf{negl}(\lambda),$$

where the real and ideal executions are defined in Figure 2.

Although it may not be apparent at this stage why we have separate $\mathsf{OblivHT.Init}$ and $\mathsf{OblivHT.Build}$ algorithms, the reason is that in the context of our ORAM construction we will instantiate the $\mathsf{OblivHT.Init}$ algorithm with our oblivious random multi-array shuffle.

We want to guarantee that the output of $\mathsf{Extract}$ algorithm is randomly shuffled from the point of view of the adversary. We formalize this similarly to the oblivious random multi-array shuffle, by providing the adversary with the real induced permutation in the real execution and a random independent permutation in the ideal execution. If the adversary cannot distinguish which is the real permutation, it means that it did not learn anything from the access patterns it observed.

3 Oblivious Random Multi-Array Shuffle

In this section, we present a novel oblivious random multi-array random shuffling algorithm that realizes the idea of leveraging entropy in the input. In particular, we assume that each input array has been previously shuffled in an order that is not known by the adversary. This assumption allows us to achieve better efficiency. Our algorithm improves on general oblivious sorting algorithms

$\mathbf{Expt}_{\mathcal{A}}^{\mathsf{Real,OblivHT}}(\lambda, n)$	$\mathbf{Expt}_{\mathsf{Sim},\mathcal{A}}^{\mathsf{Ideal,OblivHT}}(\lambda, n)$
$(D, \mathsf{st}_{\mathcal{A}}) \leftarrow \mathcal{A}_1(1^\lambda)$	$(D, \mathsf{st}_{\mathcal{A}}) \leftarrow \mathcal{A}_1(1^\lambda)$
$\chi_{\mathsf{Init}} \leftarrow \mathsf{Addrs}[(\tilde{D}, \mathsf{st}) \leftarrow \mathsf{OblivHT.Init}(D)]$	$\chi_{\mathsf{Init}} \leftarrow \mathsf{Addrs}[(\tilde{D}, \mathsf{st}_{\mathsf{Sim}}) \leftarrow \mathsf{Sim}_{\mathsf{Init}}(\lvert D\rvert)]$
$\chi_{\mathsf{Build}} \leftarrow \mathsf{Addrs}[(\tilde{H}, \mathsf{st}) \leftarrow \mathsf{OblivHT.Build}(D, \mathsf{st})]$	$\chi_{\mathsf{Build}} \leftarrow \mathsf{Addrs}[(\tilde{H}, \mathsf{st}_{\mathsf{Sim}}) \leftarrow \mathsf{Sim}_{\mathsf{Build}}(\lvert D\rvert)]$
$\chi_{\mathsf{Lookup}} \leftarrow \perp$, $\chi \leftarrow (\chi_{\mathsf{Init}}, \chi_{\mathsf{Build}})$	$\chi_{\mathsf{Lookup}} \leftarrow \perp$, $\chi \leftarrow (\chi_{\mathsf{Init}}, \chi_{\mathsf{Build}})$
for $i = 1$ to n	for $i = 1$ to n
$(k_i, \mathsf{st}_{\mathcal{A}} \mid \{k_i \neq k_j\}_{1 \le j < i}) \leftarrow \mathcal{A}_2(\tilde{H}, \mathsf{st}_{\mathcal{A}}, \chi, \chi_{\mathsf{Lookup}})$	$(k_i, \mathsf{st}_{\mathcal{A}} \mid \{k_i \neq k_j\}_{1 \le j < i}) \leftarrow \mathcal{A}_2(\tilde{H}, \mathsf{st}_{\mathcal{A}}, \chi, \chi_{\mathsf{Lookup}})$
$\chi_{\mathsf{Lookup}} \leftarrow \mathsf{Addrs}[(\tilde{H}, v_i, \mathsf{st}) \leftarrow \mathsf{OblivHT.Lookup}(k_i, \tilde{H}, \mathsf{st})]$	$\chi_{\mathsf{Lookup}} \leftarrow \mathsf{Addrs}[(\tilde{H}, \mathsf{st}_{\mathsf{Sim}}) \leftarrow \mathsf{Sim}_{\mathsf{Lookup}}(\tilde{H}, \mathsf{st}_{\mathsf{Sim}})]$
$\chi_{\mathsf{Extract}} \leftarrow \mathsf{Addrs}[(\tilde{D}, \mathsf{st}) \leftarrow \mathsf{OblivHT.Extract}(\tilde{H}, \mathsf{st})]$	$\chi_{\mathsf{Extract}} \leftarrow \mathsf{Addrs}[\tilde{H} \leftarrow \mathsf{Sim}_{\mathsf{Extract}}(\tilde{H}, \mathsf{st}_{\mathsf{Sim}})]$
Let π be the permutation induced on the items $D \setminus \{k_i\}_{i=1}^n$	Let π' be a random permutation on N items
and the padding dummies in $\tilde{D}$	
Output $b \leftarrow \mathcal{A}_3(\tilde{D}, \pi, \mathsf{st}_{\mathcal{A}}, \chi_{\mathsf{Extract}})$	Output $b \leftarrow \mathcal{A}_3(\tilde{D}, \pi', \mathsf{st}_{\mathcal{A}}, \chi_{\mathsf{Extract}})$

Figure 2: Real and ideal executions for $\mathsf{OblivHT}$.

achieving bandwidth of $O(N \log\log\lambda + \frac{N \log N}{\log \lambda})$ blocks for N data blocks. Our entropy requirement for the input comes in the following form: the N input blocks are divided in L input arrays, $\mathsf{A}_1, \ldots, \mathsf{A}_L$, each of which is randomly permuted in a manner unknown to the server storing the arrays. The arrays have sizes $N_1, \ldots, N_L$ where $N_i \geq N_{i+1}$ for $i = 1, \ldots, L$, and there exists $\mathsf{cutoff} = O(\log\log\lambda)$ such that $|\mathsf{A}_{\mathsf{cutoff}}| + \ldots + |\mathsf{A}_L| = O(\frac{N \log\log\lambda}{\log N})$ and $|\mathsf{A}_i| = \Omega(\frac{N}{\log\lambda})$ for all $i \in \{1, \ldots, \mathsf{cutoff} - 1\}$, which is the case for geometrically decreasing input array sizes that arise in the context of the ORAM shuffles.

As all the input arrays are randomly ordered, it suffices to distribute items based only on the array indices. To ensure the expectations are tightly concentrated, we require input arrays to be at least $\Omega(\frac{N}{\log\lambda})$ size. Thus, as a first step all small input arrays must be shuffled together in a single input array using an oblivious sort. Next we randomly sample an assignment, Assign, that specifies an array index, $i \in [L]$ for each location of the output array D with the intention that if $\mathsf{Assign}(j) = i$ for some output array index, $j \in [N]$, then our algorithm should distribute an item from input array A_i to the j-th location of the output array, $\mathsf{D}[j]$. To distribute items, our algorithm randomly partitions each of the input arrays $\mathsf{A}_1, \ldots, \mathsf{A}_L$ into $\tilde{m}$ input bins, $\mathsf{Bin}_1^{\mathrm{in}}, \ldots, \mathsf{Bin}_{\tilde{m}}^{\mathrm{in}}$, of expected polylog size, i.e., each input bin contains elements from all input arrays. The output array D is also partitioned into m output bins, $\mathsf{Bin}_1^{\mathrm{out}}, \ldots, \mathsf{Bin}_m^{\mathrm{out}}$, of expected polylog size but slightly smaller than the input bins $\mathsf{Bin}^{\mathrm{in}}$. To ensure output bins are smaller, m is chosen to be larger than $\tilde{m}$. We pair up input bins and output bins until we run out of output bins. As long as the input arrays are large enough, it can be shown that any input bin will have sufficient number of blocks from each input arrays to fill in the output array locations of the corresponding output bin according to Assign. Using oblivious sorts on both the input and output bin, the blocks in the input bin can be obliviously placed into the corresponding output array locations. All unused blocks of the input bin are padded to hide sizes and placed back into leftover bins $\mathsf{LeftoverBin}$ which are separated according to their original input arrays.

Once all input and output bin pairs are processed, a fraction of the input items have been placed into an appropriate output array locations. Our algorithm will recursively apply the algorithm using the leftover bins and unoccupied output array locations as input. Note that revealing which output bins are initialized in the recursive steps does not have any additional leakage as long as we are hiding which items from each input array are placed in these output bins. This is achieved by the oblivious manner of initializing the leftover bins. Each iteration reduces the input items by a constant fraction. After $O(\log\log N)$ recursive iterations, there are $O(\frac{N}{\log N})$ remaining items, and we use an oblivious sort to complete the algorithm with only $O(N)$ overhead.

Construction 5. [Oblivious Random Multi-Array Shuffle] We define our oblivious random multi-array shuffle algorithm $\mathsf{OblMultArrShuff}$, which will also use algorithms $\mathsf{OblMultArrShuff.BinShuffle}$ and $\mathsf{OblMultArrShuff.Shuffle}$ as building blocks.

$\mathsf{OblMultArrShuff}$. This algorithm takes as input L shuffled arrays and outputs array D, which contains a random permutation of the input array elements.

$\mathsf{D} \leftarrow \mathsf{OblMultArrShuff}(\mathsf{A}_1, \ldots, \mathsf{A}_L)$:

1. Initialize the output array D to be an empty array of size $N := |A_1| + \ldots + |A_L|$ blocks.
2. Choose the largest cutoff such that $|\mathsf{A}_{\mathsf{cutoff}}| \geq \frac{N}{\log \lambda}$ and then randomly permute the entries of the arrays $\mathsf{A}_{\mathsf{cutoff}}, \ldots, \mathsf{A}_L$ into $\mathsf{A}_{0,\mathsf{cutoff}}$ using an oblivious random shuffle.
3. Initialize $\mathsf{A}_{0,1}, \ldots, \mathsf{A}_{0,\mathsf{cutoff}-1}$ with the content of $\mathsf{A}_1, \ldots, \mathsf{A}_{\mathsf{cutoff}-1}$ respectively.
4. Sample a random assignment function $\mathsf{Assign} : [N] \rightarrow [\mathsf{cutoff}]$ such that $|\{b \in [N] : \mathsf{Assign}(b) = i\}| = |\mathsf{A}_{0,i}|$ for every $i \in [\mathsf{cutoff}]$. Since we assume only constant local memory, which does not fit the description of Assign, we use the following oblivious algorithm for sampling Assign at random:
   (a) Store an encryption of $\mathsf{cnt}_i := |\mathsf{A}_{0,i}|$, for $i \in [\mathsf{cutoff}]$ and an encryption of $\mathsf{cnt} = N$ on the server.
   (b) For each block $b \in [N]$, set $\mathsf{Assign}(b) = i$ with probability $\frac{\mathsf{cnt}_i}{\mathsf{cnt}}$, for $i \in [\mathsf{cutoff}]$. We do this obliviously as follows: choose a random value $r_b \in [\mathsf{cnt}]$ and set $\mathsf{Assign}(b)$ to be equal to the minimal s such that $\mathsf{cnt}_1 + \ldots + \mathsf{cnt}_s \geq r_b$ and s is computed by scanning the encrypted integers $\mathsf{cnt}_1, \ldots, \mathsf{cnt}_{\mathsf{cutoff}}$. During the scanning cnt_s and cnt are each decreased by 1. Store $(b, \mathsf{Enc}(\mathsf{Assign}(b))$ at the server.
5. Let E_0 be the array $\{(b, \mathsf{Enc}(\mathsf{Assign}(b)))\}_{b\in[N]}$ describing Assign computed and stored at the server in the previous step. Note, the server knows the first indices of each pair in E_0 as they are unencrypted.
6. Set $L := \mathsf{cutoff}$.
7. Run $\mathsf{OblMultArrShuff.Shuffle}(\mathsf{A}_{0,1}, \ldots, \mathsf{A}_{0,L}, \mathsf{D}, \mathsf{E}_0, N, 0)$.
8. Return D.

$\mathsf{OblMultArrShuff.Shuffle}$. This algorithm takes as input L randomly shuffled arrays, an output array D, which might be partially filled, the set of empty indices b_i in D together with their corresponding encrypted $\mathsf{Assign}(b_i)$ values stored in E_ℓ, and an index ℓ corresponding the current level of recursion. The algorithm fills D through several recursive steps using the encrypted values of Assign in E_ℓ and the input arrays $\mathsf{A}_{\ell,1}, \ldots, \mathsf{A}_{\ell,L}$.

$\mathsf{OblMultArrShuff.Shuffle}(\mathsf{A}_{\ell,1}, \ldots, \mathsf{A}_{\ell,L}, \mathsf{D}, \mathsf{E}_\ell, \ell)$:

1. If $|\mathsf{A}_{\ell,1}|+\ldots+|\mathsf{A}_{\ell,L}| \leq \frac{N}{\log\lambda}$, assign the items in $\mathsf{A}_{\ell,1},\ldots,\mathsf{A}_{\ell,L}$ to the remaining open positions in D using the Assign mappings stored in E_ℓ by running:

$$\mathsf{OblMultArrShuff.BinShuffle}(\mathsf{A}_{\ell,1}\cup\ldots\cup\mathsf{A}_{\ell,L},\mathsf{E}_\ell,\mathsf{D}).$$

2. Set $m := (2\epsilon)^\ell \frac{N}{\log^3\lambda}$ and $\tilde{m} := (1-2\epsilon)m$.

3. Initialize $\tilde{m}$ input bins $\mathsf{Bin}^{\text{in}}_1,\ldots,\mathsf{Bin}^{\text{in}}_{\tilde{m}}$ with random subsets of blocks from the inputs arrays as follows. For each input array $\mathsf{A}_{\ell,i}$, $i\in[L]$ distribute its blocks across $\mathsf{Bin}^{\text{in}}_1,\ldots,\mathsf{Bin}^{\text{in}}_{\tilde{m}}$ assigning each block a bin at random and recording an encryption of the source array index i. The items of $\mathsf{Bin}^{\text{in}}_1,\ldots,\mathsf{Bin}^{\text{in}}_{\tilde{m}}$ are stored encrypted on the server. Note $\mathsf{Bin}^{\text{in}}_1,\ldots,\mathsf{Bin}^{\text{in}}_{\tilde{m}}$ will have different sizes. Furthermore, the above is done in a non-oblivious manner and the server knows the distribution of the blocks from each $\mathsf{A}_{\ell,i}$ across the input bins.

4. Initialize m output bins $\mathsf{Bin}^{\text{out}}_1,\ldots,\mathsf{Bin}^{\text{out}}_m$ with random subsets of pairs from E_ℓ by assigning each pair from E_ℓ to a randomly selected bin. The items of $\mathsf{Bin}^{\text{out}}_1,\ldots,\mathsf{Bin}^{\text{out}}_m$ are stored encrypted on the server. Note the sizes of $\mathsf{Bin}^{\text{out}}_1,\ldots,\mathsf{Bin}^{\text{out}}_m$ will be different. Furthermore, the above is done in a non-oblivious manner and the server knows the distribution of the blocks from each E_ℓ across the output bins. Therefore, the server knows the subset of positions from D assigned to each output bin - these are the b values in each pair $(b,\mathsf{Enc}(\mathsf{Assign}(b)))$ of E_ℓ.

5. Initialize $\mathsf{A}_{\ell+1,1},\ldots,\mathsf{A}_{\ell+1,L}$ to be empty block arrays.

6. For $j=1,\ldots,\tilde{m}$:

   (a) Distribute the blocks from $\mathsf{Bin}^{\text{in}}_j$ in D according to the positions specified by Assign in the pairs in $\mathsf{Bin}^{\text{out}}_j$ by running

$$(\mathsf{LeftoverBin}_1,\ldots,\mathsf{LeftoverBin}_L)\leftarrow \\ \mathsf{OblMultArrShuff.BinShuffle}(\mathsf{Bin}^{\text{in}}_j,\mathsf{Bin}^{\text{out}}_j,\mathsf{D}).$$

   (b) Append $\mathsf{LeftoverBin}_i$ to $\mathsf{A}_{\ell+1,i}$ for all $i\in[L]$.

7. Collect all uninitialized indices in D together with their corresponding mappings under Assign, which have been distributed in output bins $\mathsf{Bin}^{\text{out}}_{\tilde{m}+1},\ldots,\mathsf{Bin}^{\text{out}}_m$, and set $\mathsf{E}_{\ell+1} := \mathsf{Bin}^{\text{out}}_{\tilde{m}+1}\cup\ldots\cup \mathsf{Bin}^{\text{out}}_m$.

8. Execute recursively the shuffling functionality on the remainders of the input arrays, which have not been placed in D so far but were returned as leftovers from the $\mathsf{OblMultArrShuff.BinShuffle}$ executions above, by running $\mathsf{OblMultArrShuff.Shuffle}(\ \mathsf{A}_{\ell+1,1},\ldots,\mathsf{A}_{\ell+1,L},\mathsf{D},\mathsf{E}_{\ell+1},\ell+1)$.

$\mathsf{BinShuffle}$. This is an algorithm that takes as input a bin Bin^{in} that contains items, a bin $\mathsf{Bin}^{\text{out}}$ that contains mappings under Assign of a subset of indices in the input D. $\mathsf{BinShuffle}$ distributes all but an 2ϵ fraction of the items in Bin^{in} into D according to the mappings and positions of D specified in $\mathsf{Bin}^{\text{out}}$. The items of Bin^{in} that are not placed in D are returned in leftover bins separated according to their input arrays.

$(\mathsf{LeftoverBin}_1,\ldots,\mathsf{LeftoverBin}_L)\leftarrow \mathsf{OblMultArrShuff.BinShuffle}(\mathsf{Bin}^{\text{in}},\mathsf{Bin}^{\text{out}},\mathsf{D})$

1. For $i \in [L]$, create $\mathsf{NumLeftover}_i := (4\epsilon)\frac{N_i}{N}\log^3 \lambda$ dummy blocks tagged with an array index i and append them encrypted to Bin^{in}.

2. Obliviously sort Bin^{in} according to array index of the blocks placing real blocks before dummy blocks with the same array index.

3. Let k_i^{out} be the number of pairs $(b, \mathsf{Assign}(b)) \in \mathsf{Bin}^{\text{out}}$ such that $\mathsf{Assign}(b) = i$ for all $i \in [L]$. We compute the values k_i^{out} privately in the following oblivious manner:

   (a) Initialize all $k_1^{\text{out}}, \ldots, k_L^{\text{out}}$ to 0 and store them encrypted on the server.

   (b) For each pair $(b, \mathsf{Assign}(b)) \in \mathsf{Bin}^{\text{out}}$, scan all the ciphertexts of $k_1^{\text{out}}, \ldots, k_L^{\text{out}}$ and only increment $k_{\mathsf{Assign}(b)}^{\text{out}}$.

4. Tag all items in Bin^{in} with moving, if they are a real item that will be placed in D using the Assign mapping from $\mathsf{Bin}^{\text{out}}$; $\mathsf{leftover}$ if they are real or dummy items that will be returned as leftover; or unused if they are dummy items that will be discarded. We obliviously tag items as follows: for each block j in Bin^{in}:

   (a) Let i be the input array index of the j-th block in Bin^{in}.

   (b) For $t \in [L]$, download the counter k_t^{out}. If $t \neq i$, reencrypt and upload back the counter. If $t = i$, upload an encryption of a decremented counter $\mathsf{Enc}(k_t^{\text{out}} - 1)$.

   (c) Tag the j-th block as follows: if $k_{\ell,i}^{\text{out}} > 0$, then the block is marked as real. If $-\mathsf{NumLeftover}_i < k_{\ell,i}^{\text{out}} \leq 0$, then the block is marked as $\mathsf{leftover}$. Otherwise, the block is marked as unused.

5. Obliviously sort Bin^{in} according to the tags computed in the previous step in a manner where all blocks with tag moving precede all blocks with tag $\mathsf{leftover}$ and both of these precede blocks with tags unused. All blocks with the same tag are sorted according to their input array index.

6. The blocks in Bin^{in} are separated in the following way:

   (a) Blocks that will be placed in D - these are the first $|\mathsf{Bin}^{\text{out}}|$ blocks in the sorted Bin^{in}, which are moved to TempD;

   (b) Blocks that will be returned as leftover blocks: these are the next L groups of blocks of sizes $\mathsf{NumLeftover}_1, \ldots, \mathsf{NumLeftover}_L$ blocks, which are placed in $\mathsf{LeftoverBin}_1, \ldots, \mathsf{LeftoverBin}_L$ respectively.

7. Obliviously sort the pairs $(b, \mathsf{Assign}(b))$ in $\mathsf{Bin}^{\text{out}}$ according to input array index $\mathsf{Assign}(b)$. Before sorting, encrypt the b value of each pair of $\mathsf{Bin}^{\text{out}}$ to ensure obliviousness. Recall the $\mathsf{Assign}(b)$ value is already encrypted. Note that TempD and $\mathsf{Bin}^{\text{out}}$ now contain the same numbers of items tagged with each of the input array indices, which are also sorted according to these indices. Thus, for each position $i \in [|\mathsf{Bin}^{\text{out}}|]$, the block in position i in TempD has input array index equal to $\mathsf{Assign}(b_i)$, where $(b_i, \mathsf{Assign}(b_i))$ is the i-th pair in the sorted $\mathsf{Bin}^{\text{out}}$.

8. Assign to each block in TempD its corresponding location in D as follows: for $i \in [|\mathsf{Bin}^{\text{out}}|]$, tag the block in position i in TempD with an encryption of b_i, where $(b_i, \mathsf{Assign}(b_i))$ is the i-th pair in $\mathsf{Bin}^{\text{out}}$.

9. Obliviously sort TempD according to tags computed in the previous step and copy the content of TempD in the positions denoted by their tags in D. Note these positions of D are public since the positions from D assigned to $\mathsf{Bin}^{\text{out}}$ are known to the server and only encrypted in Step 7.

How the input shrinks over calls. To get an understanding of $\mathsf{OblMultArrShuff}$, we will analyze the sizes of the inputs to recursive calls of $\mathsf{OblMultArrShuff.Shuffle}$. Let us start with the very first execution of $\mathsf{OblMultArrShuff.Shuffle}$. Initially, $\mathsf{A}_{0,1}, \ldots, \mathsf{A}_{0,L}$ collectively contain exactly N real blocks and E_0 contains N pairs (since all indices of D are still free). The indices of E_0 are uniformly and independently assigned at random to $m := \frac{N}{\log^3 \lambda}$ output bins and each is expected to have $\log^3 \lambda$ indices. The blocks of $\mathsf{A}_{0,1}, \ldots, \mathsf{A}_{0,L}$ are uniformly and independently distributed into $\tilde{m} := (1-2\epsilon)m$ input bins and each is expected to have $\log^3 \lambda/(1-2\epsilon) \approx (1+2\epsilon)\log^3 \lambda$ blocks. We note that E_1 consists of only indices of the last $2\epsilon \cdot m$ groups (see Step 7). Therefore, E_1 will contain $2\epsilon N$ indices in expectation.

Each execution of $\mathsf{OblMultArrShuff.BinShuffle}$ outputs L leftover bins of sizes $\{(4\epsilon)\frac{N_i}{N}\log^3 \lambda\}_{i=1,\ldots,L}$. So, after m executions of $\mathsf{OblMultArrShuff.BinShuffle}$, a total of $4\epsilon N_i$ will be placed into each $\mathsf{A}_{1,i}$, for $i = 1, \ldots, L$. As a result of $\mathsf{OblMultArrShuff.Shuffle}$, E_1 is only a 2ϵ fraction of the size of E_0. Similarly, each $\mathsf{A}_{1,i}$ is only a 4ϵ fraction of the size of $\mathsf{A}_{0,i}$. This reduction continues as $\mathsf{OblMultArrShuff.Shuffle}$ is executed more times. In particular, $\mathsf{A}_{\ell,i}$ will contain exactly $(4\epsilon)^\ell N_i$ blocks and E_ℓ contains $(2\epsilon)^\ell N$ indices in expectation.

The above analysis considers counting both real and dummy blocks. We turn our attention to strictly real blocks. At each level of $\mathsf{OblMultArrShuff.Shuffle}$, an 2ϵ fraction of the blocks are unassigned and will be dealt with by the later executions of $\mathsf{OblMultArrShuff.Shuffle}$. As we have seen, the unassigned blocks are kept partitioned by array index and, in order to hide the actual number of blocks that are still to be assigned from each array, dummy blocks are introduced. The next lemma shows that the number of real blocks in $\mathsf{A}_{\ell,i}$, $N_{\ell,i}$, is binomially distributed.

Lemma 6. The random variable $N_{\ell,i}$ is distributed according to $\mathsf{Binomial}[N_i, (2\epsilon)^\ell]$.

Proof. We will use the observation that for any $i \in [L]$,

$$N_{\ell,i} = |\{(b, \mathsf{Assign}(b)) \in \mathsf{E}_\ell \ : \ \mathsf{Assign}(b) = i\}|.$$

The lemma follows by the fact that the events that $(b, \mathsf{Assign}(b)) \in \mathsf{E}_\ell$ are independent and have probability $(2\epsilon)^\ell$, for every $\ell \geq 0$.

For $\ell = 0$, this is trivially true. By inductive hypothesis, assume $\Pr[(b,i) \in \mathsf{E}_\ell] = (2\epsilon)^\ell$, for some $\ell \geq 0$. All pairs of E_ℓ are distributed uniformly and independently at random into m bins in Step 4 of $\mathsf{OblMultArrShuff.Shuffle}$. A single index $(b, \mathsf{Assign}(b)) \in \mathsf{E}_\ell$ appears in $\mathsf{E}_{\ell+1}$ if and only if b is assigned to one of $\mathsf{Bin}^{\text{out}}_{\tilde{m}+1}, \ldots, \mathsf{Bin}^{\text{out}}_m$. Therefore,

$$\Pr[(b,i) \in \mathsf{E}_{\ell+1} \mid (b,i) \in \mathsf{E}_\ell] = \frac{m - \tilde{m}}{m} = 2\epsilon.$$

Since the assignment of E_ℓ is done independently of all previous events, we see that

$$\Pr[(b,i) \in \mathsf{E}_{\ell+1}] = \Pr[(b,i) \in \mathsf{E}_\ell] \cdot 2\epsilon = (2\epsilon)^{\ell+1}.$$

□

In the next lemma, we bound the number of blocks, $k^{\text{in}}_{\ell,i,j}$ of array of index i, A_i, that are assigned to the j-th input bin $\mathsf{Bin}^{\text{in}}_j$ of level ℓ of the algorithm.

Lemma 7. For every i and j and for every level ℓ,

$$\Pr\left[(1+\epsilon)\cdot\frac{N_i}{N}\log^3\lambda \le k^{\text{in}}_{\ell,i,j} \le (1+3\epsilon)\cdot\frac{N_i}{N}\log^3\lambda\right] \ge 1-\mathsf{negl}(\lambda).$$

Proof. By Lemma 6, we know that $N_{\ell,i}$ is distributed according to $\mathsf{Binomial}[N_i,(2\epsilon)^\ell]$. Moreover, each block in $\mathsf{A}_{\ell,i}$ is independently and uniformly assigned to one input bin at Step 3 and thus $k^{\text{in}}_{\ell,i,j}$ is distributed according to

$$\mathsf{Binomial}\left[N_i,\frac{(2\epsilon)^\ell}{\tilde{m}}\right] = \mathsf{Binomial}\left[N_i,\frac{\log^3\lambda}{(1-2\epsilon)N}\right].$$

Therefore

$$\mu_{\ell,i,j} := \mathbb{E}[k^{\text{in}}_{\ell,i,j}] = (1+2\epsilon)\cdot\frac{N_i}{N}\cdot\log^3\lambda$$

where we used the approximation $(1+2\epsilon)\approx\frac{1}{1-2\epsilon}$. By Chernoff Bounds, we get the following:

$$\begin{aligned}\Pr\left[(1+\epsilon)\cdot\frac{N_i}{N}\cdot\log^3\lambda \le k^{\text{in}}_{\ell,i,j} \le (1+3\epsilon)\cdot\frac{N_i}{N}\cdot\log^3\lambda\right] &= \Pr\left[(1-\epsilon)\mu_{\ell,i,j} \le k^{\text{in}}_{\ell,i,j} \le (1+\epsilon)\mu_{\ell,i,j}\right]\\ &\ge 1-2^{-\Omega(\mu_{\ell,i,j})};\end{aligned}$$

where we used the approximations $(1+2\epsilon)\cdot(1+\epsilon)\approx(1+3\epsilon)$ and $(1+2\epsilon)\cdot(1-\epsilon)\approx(1+\epsilon)$. The lemma follows by observing that, by Step 2 of $\mathsf{OblMultArrShuff}$, we have $\frac{N}{N_i}<\log\lambda$ and therefore, $\mu_{\ell,i,j}=\Omega(\log^2\lambda)$. □

A similar lemma holds for the number of blocks, $k^{\text{out}}_{\ell,i,j}$, from A_i that are assigned to a location of array D that belongs to output bin $\mathsf{Bin}^{\text{out}}_j$.

Lemma 8. For every i and j and for every level ℓ,

$$\Pr\left[(1-\epsilon)\cdot\frac{N_i}{N}\cdot\log^3\lambda \le k^{\text{out}}_{\ell,i,j} \le (1+\epsilon)\cdot\frac{N_i}{N}\cdot\log^3\lambda\right].$$

Proof. Each pair of E_ℓ is independent and uniformly assigned to one output bin at Step 4 of $\mathsf{OblMultArrShuff.Shuffle}$. So, $k^{\text{out}}_{\ell,i,j}$ is distributed according to

$$\mathsf{Binomial}\left[N_i,\frac{(2\epsilon)^\ell}{m}\right] = \mathsf{Binomial}\left[N_i,\frac{\log^3\lambda}{N}\right]$$

and $\mu_{\ell,i,j} := \mathbb{E}[k^{\text{out}}_{\ell,i,j}] = \frac{N_i}{N}\cdot\log^3\lambda$. By Chernoff bounds, we obtain that

$$\begin{aligned}\Pr\left[(1-\epsilon)\cdot\frac{N_i}{N}\log^3\lambda \le k^{\text{out}}_{\ell,i,j} \le (1+\epsilon)\cdot\frac{N_i}{N}\log^3\lambda\right] &= \Pr\left[(1-\epsilon)\mu_{\ell,i,j} \le k^{\text{out}}_{\ell,i,j} \le (1+\epsilon)\mu_{\ell,i,j}\right]\\ &\ge 1-2^{-\Omega(\mu_{\ell,i,j})}.\end{aligned}$$

The lemma follows by observing that, by Step 2 of $\mathsf{OblMultArrShuff}$, we have $\frac{N}{N_i}<\log\lambda$ and therefore, $\mu_{\ell,i,j}=\Omega(\log^2\lambda)$. □

Lemma 9. The construction $\mathsf{OblMultArrShuff}$ aborts with probability negligible in λ.

Proof. $\mathsf{OblMultArrShuff}$ aborts only if at Step 4 of $\mathsf{OblMultArrShuff.BinShuffle}$ one of the following two events occurs, for some i, j, ℓ,

1. The j-th input bin contains too few real blocks from array $A_{\ell,i}$ and thus a dummy block is marked moving. This happens if $k^{\text{in}}_{\ell,i,j} < k^{\text{out}}_{\ell,i,j}$. But by Lemma 7 and Lemma 8 we have that

$$k^{\text{in}}_{\ell,i,j} \geq (1+\epsilon) \cdot \frac{N_i}{N} \cdot \log^3 \lambda \geq k^{\text{out}}_{\ell,i,j}$$

except with negligible probability.

2. The j-th input bin contains too many real blocks from array $A_{\ell,i}$ and thus a real block is marked unused. This happens if $k^{\text{in}}_{\ell,i,j} > k^{\text{out}}_{\ell,i,j} + \mathsf{NumLeftover}_i = k^{\text{out}}_{\ell,i,j} + 4\epsilon\frac{N_i}{N} \cdot \log^3 \lambda$. Again by Lemma 7 and Lemma 8 we have that

$$k^{\text{in}}_{\ell,i,j} \leq (1+3\epsilon) \cdot \frac{N_i}{N} \log^3 \lambda \leq 4\epsilon \cdot \frac{N_i}{N} \log^3 \lambda + k^{\text{out}}_{\ell,i,j}$$

except with negligible probability.

□

Theorem 10. The construction $\mathsf{OblMultArrShuff}$ is an oblivious random multi-array shuffle according to Definition 3.

Proof. We construct a simulation Sim that outputs D as follows. Sim will fill each A_i with encryptions of random values. Sim executes the honest $\mathsf{OblMultArrShuff}$ algorithm, which will construct D filled with the encryptions of random values. The output array generated in the real and simulated execution are indistinguishable since they contain encryptions of the same number of items.

The access pattern while running $\mathsf{OblMultArrShuff}$ involve either linear scans, oblivious sorts and movements of blocks determined by random coin flips (see Step 3 and 4 of $\mathsf{OblMultArrShuff.Shuffle}$). However, the random coin flips are revealed publicly and indistinguishable from any other truly random coin flip sequence. Therefore, the access patterns from the real and simulated execution are indistinguishable.

Finally, we have to show that final permutation given to the adversary is indistinguishable in conjunction with the access pattern. In the real experiment, each A_i is permuted according to a τ_i hidden from the adversary. $\mathsf{OblMultArrShuff}$ applies an Assign function chosen uniformly at random. We show in Lemma 20 the result is a uniformly random permutation. Since Assign is chosen before revealing any random values in the access pattern, Assign is independent of access patterns in the real execution. Therefore, the final permutations in the real and simulated execution are indistinguishable completing the proof. □

Efficiency. For efficiency, we focus on the situation when there exists $\mathsf{cutoff} = O(\log\log\lambda)$ such that $|\mathsf{A}_{\mathsf{cutoff}}| + \ldots + |\mathsf{A}_L| = O(\frac{N \log\log\lambda}{\log N})$. In this case, we show that $\mathsf{OblMultArrShuff}$ uses $O(BN\log\log\lambda + B\frac{N\log N}{\log\lambda})$ bandwidth except with probability negligible in λ.

Note, $\mathsf{OblMultArrShuff}$ performs an oblivious sort on $|\mathsf{A}_{\mathsf{cutoff}}| + \ldots + |\mathsf{A}_L|$ blocks in Step 2 of $\mathsf{OblMultArrShuff}$. Since $|\mathsf{A}_{\mathsf{cutoff}}| + \ldots + |\mathsf{A}_L| = O(\frac{N\log\log\lambda}{\log N})$, a total of $O(BN\log\log\lambda)$ bandwidth is required. Constructing Assign in Step 4 requires $O(BN \cdot \mathsf{cutoff}) = O(BN \log\log\lambda)$. The remaining steps of $\mathsf{OblMultArrShuff}$ require $O(N)$ bandwidth.

Let us now focus on the bandwidth of $\mathsf{OblMultArrShuff.BinShuffle}$. An oblivious sort is applied to at most $|\mathsf{Bin}^{\mathrm{in}}| + 4\log^3\lambda$ blocks in Steps 2 and 5. An oblivious sort is applied to at most $|\mathsf{Bin}^{\mathrm{out}}|$ blocks in Steps 7 and 9. Step 3 requires $O(|\mathsf{Bin}^{\mathrm{out}}| \cdot \mathsf{cutoff}) = O(|\mathsf{Bin}^{\mathrm{out}}| \cdot \log\log\lambda)$ blocks of bandwidth. Step 4 requires $O(|\mathsf{Bin}^{\mathrm{in}}| \cdot \mathsf{cutoff}) = O(|\mathsf{Bin}^{\mathrm{in}}| \cdot \log\log\lambda)$ blocks of bandwidth. All other steps of $\mathsf{OblMultArrShuff.BinShuffle}$ require $O(|\mathsf{Bin}^{\mathrm{out}}|+|\mathsf{Bin}^{\mathrm{in}}|)$ blocks of bandwidth. Altogether, the total required bandwidth of $\mathsf{OblMultArrShuff.BinShuffle}$ is $O((|\mathsf{Bin}^{\mathrm{in}}|+\log^3\lambda)(\log|\mathsf{Bin}^{\mathrm{in}}|+\log\log\lambda)+|\mathsf{Bin}^{\mathrm{out}}|(\log|\mathsf{Bin}^{\mathrm{out}}| + \log\log\lambda))$.

Finally, we consider $\mathsf{OblMultArrShuff.Shuffle}$ now. If $|\mathsf{A}_{\ell,1}|+\ldots+|\mathsf{A}_{\ell,L}| \leq \frac{N}{\log\lambda}$, then $\mathsf{OblMultArrShuff.Shuffle}$ executes $\mathsf{OblMultArrShuff.BinShuffle}$ at Step 5 requiring $O\left(\frac{N(\log N+\log\log\lambda)}{\log\lambda}\right)$ blocks of bandwidth. In the other case, $\mathsf{OblMultArrShuff.BinShuffle}$ is executed $\tilde{m}$ times. By Lemma 7, we know that, for all $j \in [\tilde{m}]$, $|\mathsf{Bin}^{\mathrm{in}}_j| = \sum_{i=1}^{L} k^{\mathrm{in}}_{\ell,i,j} \leq (1+3\epsilon)\log^3\lambda$. By Lemma 8, we know that, for all $j \in [m]$, $|\mathsf{Bin}^{\mathrm{out}}_j| = \sum_{i=1}^{L} k^{\mathrm{out}}_{\ell,i,j} \leq (1+\epsilon)\log^3\lambda$. Therefore, each execution of $\mathsf{OblMultArrShuff.BinShuffle}$ requires $O(\log^3\lambda\log\log\lambda)$ blocks of bandwidth. All executions require $O((2\epsilon)^\ell N\log\log\lambda)$ blocks of bandwidth. The cost of all executions of $\mathsf{OblMultArrShuff.Shuffle}$ is

$$\sum_{\ell\geq 0} O((2\epsilon)^\ell N\log\log\lambda)) = O(N\log\log\lambda)$$

blocks of bandwidth, for large enough N, when $\epsilon < 1/4$ thus completing the proof.

4 Oblivious Hash Table

In this section we present our *oblivious hash table* construction which achieves bandwidth overhead of $O(\log N + \log\log\lambda)$ blocks, amortized per query. It uses as a building block the notion of an *oblivious bin*, which provides the same security properties as a general OHT but the main difference is that the oblivious bin structure will be used only for small inputs, which can be obliviously shuffled without violating out overall efficiency requirements. The largest bins will be of size $\frac{N}{\log^c\lambda}$ for $c > 1$. We start with our oblivious bin constructions.

4.1 Oblivious Cuckoo Hash Bin

We now present a construction of oblivious bins using cuckoo hashing. In particular, the $\mathsf{Init}, \mathsf{Build}$ and Lookup will be identical to the cuckoo hashing hash table presented by Goodrich and Mitzenmacher [22]. Our modification will be the introduction of an $\mathsf{Extract}$ functionality which is integral to our Oblivious RAM construction. Our oblivious cuckoo hash bin only works on input arrays with at least $\log^7\lambda$ items.

Construction 11 (Oblivious Cuckoo Hash Bin)**.** Let F be a pseudorandom function and $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ be a symmetric key encryption. We define an oblivious hash construction $\mathsf{CuckooBin} = (\mathsf{CuckooBin.Init}, \mathsf{CuckooBin.Build}, \mathsf{CuckooBin.Lookup}, \mathsf{CuckooBin.Extract})$ as follows

- $(\mathsf{st}, \tilde{D}) \leftarrow \mathsf{CuckooBin.Init}\left(D = (k_i, v_i)_{i=1}^N\right)$:

1. Generate N dummy items $(k'_i, \bot)_{i=N+1}^{2N}$ where $k'_i \in \mathcal{U}_{\mathsf{dummy}}$ and append them to D.
2. Generate an encryption key SK and set $\mathsf{st} \leftarrow \mathsf{SK}$.
3. Set $\tilde{D} = \{\mathsf{ct}_i = \mathsf{Enc}(\mathsf{SK}, (b_i, k_i, v_i))\}_{i=1}^{2N}$ where $b_i = 0$ if k_i is real and $b_i = 1$ if k_i is dummy. The real and dummy items are in any, not necessarily random, order.

- $(\mathsf{st}, \tilde{H}_{\mathsf{cuckoo}}, \mathsf{S}) \leftarrow \mathsf{CuckooBin.Build}\left(\mathsf{SK}, \tilde{D} = \{\mathsf{Enc}(\mathsf{SK}, (b_i, k_i, v_i))\}_{i=1}^{2N}\right)$:

  1. Generate PRF keys $\mathsf{K}_1, \mathsf{K}_2$ and encryption key and set $\mathsf{st} \leftarrow (\mathsf{K}_1, \mathsf{K}_2, \mathsf{SK})$.
  2. Run an oblivious algorithm to construct a Cuckoo hash table that consists of two tables $\mathsf{T}_1, \mathsf{T}_2$ and a stash S with parameters $\mathsf{K}_1, \mathsf{K}_2$, which contain the items of $\tilde{D}$ using SK to decrypt entries of $\tilde{D}$ when necessary. Set $\tilde{H}_{\mathsf{cuckoo}} \leftarrow (\mathsf{T}_1, \mathsf{T}_2, \mathsf{S})$.

- $(v, \tilde{H}'_{\mathsf{cuckoo}}) \leftarrow \mathsf{CuckooBin.Lookup}\left(k_i, \tilde{H}_{\mathsf{cuckoo}} = (\mathsf{T}_1, \mathsf{T}_2), \mathsf{S}, \mathsf{st} = (\mathsf{K}_1, \mathsf{K}_2, \mathsf{SK})\right)$:

  1. Access all items in S as well as items $\mathsf{T}_1[\mathsf{F}_{K_1}(k_i)]$ and $\mathsf{T}_2[\mathsf{F}_{K_2}(k_i)]$, decrypting them using SK and checking whether the stored items matches the search index k_i. If the item with index k_i is found, set v to be the corresponding data, and otherwise set $v \leftarrow \bot$.
  2. If either $\mathsf{T}_1[\mathsf{F}_{K_1}(i)]$ or $\mathsf{T}_2[\mathsf{F}_{K_2}(i)]$ is of the form (b_i, k_i, v_i) write back an encryption of the value $(b_i, k'_i, \bot)$ where k'_i is a dummy key.

- $(\tilde{D}, \mathsf{st}') \leftarrow \mathsf{CuckooBin.Extract}(\tilde{H}_{\mathsf{cuckoo}}, \mathsf{S}, \mathsf{st})$:

  1. Use an oblivious sort together all items in T_1, T_2 and S according their flag bit b_i to obtain an array $\tilde{H}$.
  2. Set $\tilde{D}$ to be the first N items of $\tilde{H}$.
  3. Use an oblivious sort to permute $\tilde{D}$ at random.

Theorem 12. The oblivious Cuckoo bin construction $\mathsf{CuckooBin}$ presented in Construction 11 is an oblivious hash table according to Definition 4 assuming that the number of input blocks is $\Omega(\log^7 \lambda)$.

Proof. The simulator will initiate the Cuckoo hash table using random input as data and our $\mathsf{Sim}_{\mathsf{Init}}$ outputs $2N$ ciphertexts encrypting random values. We will use the result of Goodrich and Mitzenmacher [22], which was later given a complete proof in the work of Chan et al. [7], which says that there existing a simulator that can simulate the initialization of a Cuckoo table on $n = \Omega(\log^7 \lambda)$ elements given just the number of elements with a stash of size $O(\log \lambda)$. We will use this simulator as our $\mathsf{Sim}_{\mathsf{Build}}$. The $\mathsf{Sim}_{\mathsf{Lookup}}$ will just access two random locations in T_1 and T_2. The indistinguishability of this simulated access from real lookups follows again from the result of [22]. Finally, $\mathsf{Sim}_{\mathsf{Extract}}$ just runs the regular extraction algorithm on the simulated oblivious Cuckoo table. The indistinguishability of the real world, where the adversary gets the real permutation, and the ideal world, where he obtains a random independent permutation, follows from the fact that the only leakage is the access pattern from the oblivious sort, which does not reveal any information about the permutation. At the end of $\mathsf{CuckooBin.Extract}$, $\tilde{D}$ is randomly permuted using an oblivious sort. The random permutation is chosen independently and unifomly at random. Therefore, the final permutation of $\tilde{D}$ is indistinguishable from a uniformly at random chosen permutation. $\square$

Efficiency. The initialization, building and extraction algorithms for Cuckoo bin have bandwidth of $O(n \log n)$ blocks. The lookup time has instead bandwidth of $O(\log n + \log\log \lambda)$. While at first sight the Cuckoo bin does not seem to bring any efficiency advantage and it also needs much larger minimal size of the blocks in order to use Cuckoo hash tables, the majority of the lookup time $O(\log n)$ is spent on reading the stash. When we use several levels of Cuckoo hashes in our general oblivious hash table construction, we will have a more efficient way to deal with the stashes on multiple Cuckoo hash bins. In particular, all the stashes can be combined into a single $O(\log n)$ size stash (see [22] for a proof).

Construction 13. [Oblivious Hash Table] Let F be a pseudorandom function and $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ be a symmetric key encryption. Moreover, let $\mathsf{OblivBin} = (\mathsf{OblivBin.Init}, \mathsf{OblivBin.Build}, \mathsf{OblivBin.Lookup}, \mathsf{OblivBin.Extract})$ be an oblivious bin. We define oblivious hash $\mathsf{OblivHT} = (\mathsf{OblivHT.Init}, \mathsf{OblivHT.Build}, \mathsf{OblivHT.Lookup}, \mathsf{OblivHT.Extract})$ as follows:

$\mathsf{OblivHT.Init}$**.** This algorithm receives as input a database D consisting of N key/value pairs and constructs N *real* items from the N pairs of D and N additional *dummy* items. The $2N$ items are randomly shuffled and then returned.

$(\mathsf{st}, \tilde{D}) \leftarrow \mathsf{OblivHT.Init}(D = \{(k_i, v_i)\}_{i=1}^{N})$:

1. Generate key $\mathsf{SK} \leftarrow \mathsf{Gen}(1^\lambda)$ and set $\mathsf{st} \leftarrow \mathsf{SK}$.
2. Generate real items of the form $\{\mathsf{Enc}\left(\mathsf{SK}, (0, k_i, v_i)\right)\}_{i=1}^{N}$ and an additional N dummy items of the form $\{\mathsf{Enc}\left(\mathsf{SK}, (1, k_i, \perp)\right)\}_{i=N+1}^{2N}$.
3. Compute an oblivious shuffle on the above $2N$ real and dummy items and set $\tilde{D}$ to be the result.

$\mathsf{OblivHT.Build}$**.** This algorithm builds an oblivious hash table from the output of $\mathsf{OblivHT.Init}$.

$(\mathsf{st}, \tilde{H}, \tilde{S}) \leftarrow \mathsf{OblivHT.Build}\left(\tilde{D}, \mathsf{st}\right)$: the data $\tilde{D}$ contains exactly half items with tag 0 and exactly half with tag 1.

1. Return $(\mathsf{st}, \tilde{H}, \tilde{S}) \leftarrow \mathsf{OblivHT.BuildLevel}\left(\tilde{D}, \emptyset, N, 1, \mathsf{st}\right)$.

$\mathsf{OblivHT.BuildLevel}$**.** This algorithm constructs the levels of the OHT structure. It takes as input an array $\tilde{D}$ of $2 \cdot R^{\mathsf{real}}$ encrypted items (b_i, k_i, v_i) of which R^{real} have $b_i = 0$ (the real items) and R^{real} have $b_i = 1$ (the dummy items). In addition the algorithm takes the items $\tilde{S}$ that were assigned to the stash by the previous levels, the counter ctr that keeps track of the current level and encryption key SK used to encrypt the items. The algorithm returns the sequence $\left(\tilde{H}^{(\mathsf{ctr})}, \tilde{H}^{(\mathsf{ctr}+1)}, \ldots, \tilde{H}^{(\mathsf{d})}\right)$ of levels of the OHT structure from ctr to the maximum level d, the stashes $\tilde{S}^{(\mathsf{d}-1)}$ and $\tilde{S}^{(\mathsf{d})}$ and state information st.

The algorithm distinguishes two cases. If the combined size of $\tilde{D}$ and $\tilde{S}$ is not too large (see Step 1 below) then it constructs an Oblivious Bin $\tilde{H}^{(\mathsf{d}-1)}$ for $\tilde{D}$ and an Oblivious Bin $\tilde{H}^{(\mathsf{d})}$ for $\tilde{S}$ that constitute, respectively, levels $\mathsf{d} - 1$ and d of the OHT. Each level gives a stash, $\tilde{S}^{(\mathsf{d}-1)}$ and $\tilde{S}^{(\mathsf{d})}$, and a state, $\mathsf{st}^{\mathsf{d}-1}$ and st^{d}.

Otherwise, the algorithm proceeds as follows. It performs two balls and bins processes. The first process considers R^{real} 0-balls and R^{real} 1-balls for a total of $n = 2 \cdot R^{\mathsf{real}}$ balls and $m := n/(2\log^{\mathsf{c}} \lambda)$ bins. We denote by X_j^{real} the random variable of the number of the 0-balls that are assigned to the j-th bin and X_j^{dummy} to be the random variable of the number of 1-balls that are assigned to the j-th bin. It is easy to see that $\mathbb{E}[X_j^{\mathsf{real}}] = \mathbb{E}[X_j^{\mathsf{dummy}}] = \log^{\mathsf{c}} \lambda$. This first process is implemented by taking each item (b_i, k_i, v_i) and by assigning it to bin $j = \mathsf{F}(\mathsf{K}_{\mathsf{ctr}}, k_i)$, where F is a pseudorandom function and $\mathsf{K}_{\mathsf{ctr}}$ is a randomly chosen seed. The second process considers the same number of m bins and a smaller number of $n' := (1-\delta)n/2$ balls with only 0-tags. This process is only simulated as the algorithm only needs to sample the random variables Y_j^{real} describing the number of balls in each bin. It is easy to see that $\mathbb{E}[Y_j^{\mathsf{real}}] = (1-\delta)\log^{\mathsf{c}} \lambda$.

The algorithm aborts if there exists a bin j such that the number of 0-balls placed in the second process exceeds the number of 0-balls placed in the first process (that is, $Y_j^{\mathsf{real}} > X_j^{\mathsf{real}}$) or if the number of 1-balls placed in the first process is less than $\mathsf{thrsh} - Y_j^{\mathsf{real}}$ where $\mathsf{thrsh} := 2(1-\delta)\log^{\mathsf{c}} \lambda$. We will later show that the abort probability is negligible by choosing δ appropriately.

Next the algorithm constructs an oblivious bin for each bin j by selecting exactly Y_j^{real} 0-balls and $\mathsf{thrsh} - Y_j^{\mathsf{real}}$ 1-balls. Note that each oblivious bin will consider exactly thrsh items and will produce a new stash $\tilde{S}^{(\mathsf{ctr},j)}$ that is added to the $\tilde{S}$ received from previous levels. The selection of the items of bin j that are considered for the j-th oblivious bin is performed obliviously. The m oblivious bins consume $\sum_j Y_j^{\mathsf{real}} = (1-\delta)n/2$ 0-balls and $\sum_j(\mathsf{thrsh} - Y_j^{\mathsf{real}}) = (1-\delta)n/2$ 1-balls. Therefore the algorithm is left with the same number, $\delta n/2$, of leftover 0 and 1-balls that are collected in array $\mathsf{D}_{\mathsf{over}}$ (see Step 2(e)v below) and used to recursively construct the next levels. We next formally describe algorithm $\mathsf{OblivHT.BuildLevel}$.

$(\mathsf{st}, \tilde{H}, \tilde{S}) \leftarrow \mathsf{OblivHT.BuildLevel}\left(\tilde{D}, \tilde{S}, R^{\mathsf{real}}, \mathsf{ctr}, \mathsf{SK}\right)$:

1. If $|\tilde{D}| + |\tilde{S}| = O\left(\frac{N}{\log \lambda}\right)$:
   (a) Set $(\tilde{\mathsf{st}}^{\mathsf{ctr}}, \tilde{H}^{\mathsf{ctr}}, \tilde{S}^{\mathsf{ctr}}) \leftarrow \mathsf{OblivBin.Build}(\tilde{D})$.
   (b) Set $(\tilde{\mathsf{st}}^{\mathsf{ctr}+1}, \tilde{H}^{\mathsf{ctr}+1}, \tilde{S}^{\mathsf{ctr}+1}) \leftarrow \mathsf{OblivBin.Build}(\tilde{S})$.
   (c) Return $(\mathsf{st}^{\mathsf{ctr}} = \tilde{\mathsf{st}}^{\mathsf{ctr}}, \mathsf{st}^{\mathsf{ctr}+1} = (\tilde{\mathsf{st}}^{\mathsf{ctr}+1}, \mathsf{SK})), (\tilde{H}^{\mathsf{ctr}}, \tilde{H}^{\mathsf{ctr}+1}), (\tilde{S}^{\mathsf{ctr}}, \tilde{S}^{\mathsf{ctr}+1})$.
2. Otherwise, construct a level in the oblivious hash table as follows:
   (a) Let F be a PRF with output range $[\frac{R^{\mathsf{real}}}{\log^{\mathsf{c}} \lambda}]$. Randomly select PRF key $\mathsf{K}_{\mathsf{ctr}}$ of length λ.
   (b) Initialize $\frac{R^{\mathsf{real}}}{\log^{\mathsf{c}} \lambda}$ empty bins.
   (c) For each item in $\tilde{D}$, decrypt to get the values (b_i, k_i, v_i) and append $\mathsf{Enc}(\mathsf{SK}, (b_i, k_i, v_i))$ to bin $\mathsf{F}(\mathsf{K}_{\mathsf{ctr}}, k_i)$.
   (d) Let $\mathsf{thrsh} = 2(1-\delta)\log^{c} \lambda$.
   (e) Set $T = (1-\delta)R^{\mathsf{real}}$ and $t = \frac{R^{\mathsf{real}}}{\log^{\mathsf{c}} \lambda}$. For $1 \leq j \leq \frac{R^{\mathsf{real}}}{\log^{\mathsf{c}} \lambda}$ do as follows:
      i. Sample from the binomial distribution $R_j \leftarrow \mathsf{Binomial}(T, \frac{1}{t})$.
      ii. Set $T = T - R_j$ and $t = t - 1$.
      iii. If there are less than R_j items with tag value 0 or the total number of items with tag value 1 is less than $\mathsf{thrsh} - R_j$, then abort.

iv. Linearly scan the items assigned to B_j in Step 2c, leaving the tags $b_i = 0$ to the first R_j real items from $\tilde{D}$ and setting the tag to of the remaining real items to be $b_i = 2$; the added dummy items stay with tag $b_i = 1$.

v. Obliviously sort the items assigned to B_j according to their assigned tag. Move all items at the end of array starting from position $\mathsf{thrsh} + 1$ to array $\mathsf{D}_{\mathsf{over}}$ changing the tag $b_i = 2$ of the real items back to $b_i = 0$.

vi. Initialize an oblivious bin structure on the items left in B_j:

$$(\mathsf{st}^{(\mathsf{ctr},j)}, \tilde{H}^{(\mathsf{ctr},j)}, \tilde{S}^{(\mathsf{ctr},j)}) \leftarrow \mathsf{OblivBin.Build}(\mathsf{SK}, \mathsf{B}_j).$$

vii. For each item in $\tilde{S}^{(\mathsf{ctr},j)}$, append the encrypted tag $\mathsf{Enc}(\mathsf{SK}, (\mathsf{ctr}, j))$ and add to the set $\tilde{S}$.

(f) Let $\tilde{H}^{(\mathsf{ctr})} \leftarrow \mathsf{Enc}\left(\mathsf{SK}, \{(\mathsf{st}^{(\mathsf{ctr},j)}, \tilde{H}^{(\mathsf{ctr},j)})\}_{j=1}^{R_{\mathsf{real}}/\log^{\mathsf{c}}\lambda}, \mathsf{K}_{\mathsf{ctr}}\right)$.

(g) Call recursively $\mathsf{OblivHT.BuildLevel}$ on the leftover items in $\mathsf{D}_{\mathsf{over}}$:

$$(\mathsf{st}', \tilde{H}', \tilde{S}') \leftarrow \mathsf{OblivHT.BuildLevel}\left(\mathsf{D}_{\mathsf{over}}, \tilde{S}, \delta R^{\mathsf{real}}, \mathsf{ctr}+1, \mathsf{SK}\right).$$

(h) Parse $\tilde{H}'$ as $\tilde{H}' = (\tilde{H}^{(\mathsf{ctr}+1)}, \ldots, \tilde{H}^{(\mathsf{d})})$ and st' as $\mathsf{st}' = (\mathsf{st}^{\mathsf{ctr}+1}, \ldots, \mathsf{st}^{\mathsf{d}})$.
Return $(\mathsf{st}, \tilde{H}, \tilde{S})$, where $\mathsf{st} \leftarrow \left(\mathsf{st}^{\mathsf{ctr}}, \mathsf{st}^{\mathsf{ctr}+1}, \ldots, \mathsf{st}^{\mathsf{d}}\right)$, $\tilde{H} \leftarrow \left(\tilde{H}^{(\mathsf{ctr})}, \tilde{H}^{(\mathsf{ctr}+1)}, \ldots, \tilde{H}^{(\mathsf{d})}\right)$, and $\tilde{S} \leftarrow \tilde{S}'$.

$\mathsf{OblivHT.Lookup}$. This algorithm retrieves an item stored in the OHT table.

$(v, \tilde{H}', \tilde{S}', \mathsf{st}') \leftarrow \mathsf{OblivHT.Lookup}(k, \tilde{H}, \tilde{S}, \mathsf{st})$:

1. Parse st as $\mathsf{st} = \left(\mathsf{st}^1, \ldots, \mathsf{st}^{\mathsf{d}}\right)$ and obtain SK from st^{d}.
   Parse $\tilde{H}$ as $\tilde{H} = (\tilde{H}^{(1)}, \ldots, \tilde{H}^{(\mathsf{d})})$.
   Set $\mathsf{found} = 0$.

2. For $\mathsf{ctr} = \mathsf{d}$ to 1, do the following:

   (a) If $\mathsf{found} = 0$, set $j = \mathsf{F}(\mathsf{K}_{\mathsf{ctr}}, k)$, else choose j at random among the α_{ctr} bins at level ctr.

   (b) If $\mathsf{ctr} \geq \mathsf{d} - 1$, then execute $(v', \tilde{H}', \tilde{S}') \leftarrow \mathsf{OblivBin.Lookup}(k, \tilde{H}^{(\mathsf{ctr})}, \tilde{S}^{(\mathsf{ctr})}, \mathsf{st}^{(\mathsf{ctr})})$. If $v' \neq \perp$, set $v \leftarrow v'$ and $\mathsf{found} = 1$.

   (c) Otherwise when $\mathsf{ctr} < \mathsf{d} - 1$, then execute the following two steps

$$\{(\mathsf{st}^{(\mathsf{ctr},k)}, \tilde{H}^{(\mathsf{ctr},k)})\}_{k=1}^{\alpha_{\mathsf{ctr}}} \leftarrow \mathsf{Dec}(\mathsf{SK}, \tilde{H}^{(\mathsf{ctr})}),$$
$$(v', \tilde{H}', \tilde{S}') \leftarrow \mathsf{OblivBin.Lookup}(k, \tilde{H}^{(\mathsf{ctr},j)}, \perp, \mathsf{st}^{(\mathsf{ctr},j)}).$$

   If $v' \neq \perp$, set $v \leftarrow v'$ and $\mathsf{found} = 1$.

$\mathsf{OblivHT.Extract}$. This algorithm returns a fixed size data array that contains only the unqueried items in the OHT padded with dummy items.

$(\tilde{D}, \mathsf{st}') \leftarrow \mathsf{OblivHT.Extract}(\tilde{H}, \tilde{S}, \mathsf{st})$:

1. Let $\mathsf{st} = (\mathsf{st}^{(1)}, \mathsf{d}, \mathsf{SK})$ and $\tilde{H} = (\tilde{H}^{(1)}, \ldots, \tilde{H}^{(\mathsf{d})})$.
2. Execute $(\tilde{D}^{\mathsf{d}}, \mathsf{st}^{\mathsf{d}}) \leftarrow \mathsf{OblivBin.Extract}(\tilde{H}^{\mathsf{d}}, \tilde{S}^{\mathsf{d}}, \mathsf{st}^{\mathsf{d}})$.
3. Obliviously sort the items in $\tilde{D}^{\mathsf{d}}$ according to their appended encrypted tags $\mathsf{Enc}(\mathsf{SK}, (\mathsf{ctr}, j))$ which denotes that the item comes from the j-th $\mathsf{OblivBin}$ in the ctr-th level of the $\mathsf{OblivHT}$. As a result, we get the stashes $\tilde{S}^{(\mathsf{ctr},j)}$ for all $\mathsf{OblivBin}$ built at every level.
4. For $\mathsf{ctr} \in [\mathsf{d}-1]$ and $j \in [\alpha_{\mathsf{ctr}}]$ where $\alpha_{\mathsf{ctr}} = \frac{\delta^{i-1}N}{\log^{\mathsf{c}} \lambda}$, let
$$(\tilde{D}_{\mathsf{ctr},j}, \mathsf{st}') \leftarrow \mathsf{OblivBin.Extract}(\tilde{H}^{(\mathsf{ctr},j)}, \tilde{S}^{(\mathsf{ctr},j)}, \mathsf{st}^{(\mathsf{ctr},j)}),$$
for each bin B_j in level ctr, where $\tilde{S} = \perp$ for $\mathsf{ctr} < \mathsf{d}$ and $\{(\mathsf{st}^{(\mathsf{ctr},j)}, \tilde{H}^{(\mathsf{ctr},j)})\}_{j=1}^{\alpha_{\mathsf{ctr}}} \leftarrow \mathsf{Dec}(\mathsf{SK}, \tilde{H}^{(\mathsf{ctr})})$. Append $\tilde{D}_{i,j}$ to $\tilde{D}$.

Generating binomial variates. This can be done using the Splitting algorithm for binomial random variates in Section X.4.4 of [15]. The original Splitting algorithm executes in expected constant time. To make it oblivious, we repeat the coin toss of the rejection method $\omega(\log \lambda)$ times at which point the algorithm will terminate except with negligible probability. Furthermore, the algorithm requires performing operations over real numbers. By using $\Theta(\log n \cdot \mathsf{poly}(\log\log\lambda))$ bits which is $\Theta(\mathsf{poly}(\log\log\lambda))$ blocks to represent real numbers, we can ensure truncation of real numbers does not affect our result. Therefore, the total running time of generating a single binomial variate for each bin is operating over $\omega(\log\lambda \cdot \mathsf{poly}(\log\log\lambda))$ blocks. However, we still need to perform operations over $\Theta(\log^{\mathsf{c}} \lambda)$ blocks for each bin for some constant $\mathsf{c} > 1$. Therefore, the running time of our $\mathsf{OblivHT.Build}$ remains the same as the bandwidth overhead. We note that the running time of generating binomial variates only affects the running time of our ORAM construction and not the bandwidth overhead.

Efficiency. The initialization for the oblivious hash table is an oblivious sort, which we can do with bandwidth $O(N \log N)$. The building of the oblivious hash table from a shuffled array will be proportional to the the cost of running the build algorithm for each bin at each level plus on oblivious shuffle per bin. The size of each bin in each level, except the last two, is $O(\log^{\mathsf{c}} \lambda)$. The total number of bins of this size is
$$\sum_{i=1}^{\log\log\lambda - 1} \frac{1}{\log^{\mathsf{c}} \lambda}(2\delta)^{i-1}N = \frac{1}{\log^{\mathsf{c}} \lambda}(1-(2\delta)^{\log\log\lambda})N = O\left(\frac{N}{\log^{\mathsf{c}}\lambda}\right).$$

Each stash is size $O(\log N)$ and there are $O(N/\log^{\mathsf{c}} \lambda)$ stashes. Therefore, the total size of all the stashes is at most $O(\log N \cdot N/\log^{\mathsf{c}} \lambda)$ which is $O(N/\log\lambda)$ whenever $\lambda = \mathsf{poly}(N)$ and $\mathsf{c} \geq 2$. In our case, we have the restriction that $\mathsf{c} \geq 7$ in order to satisfy the input size required by the oblivious algorithm to correctly construct a Cuckoo hash table (see Theorem 12). Therefore, the build for the oblivious hash table will take:

$$O\left(\frac{N}{\log^{\mathsf{c}}\lambda}(\log^{\mathsf{c}}\lambda)(\log\log^{\mathsf{c}}\lambda)+\frac{N}{\log\lambda}\log N\right)=O\left(N\log\log\lambda+\frac{N}{\log\lambda}\log N\right).$$

A lookup in the oblivious hash table consists of $O(\log\log\lambda)$ lookups on bins of size $\log^{\mathsf{c}}\lambda$ plus the lookup on the single bin in the smallest level which is of size $N/(\log^{\mathsf{c}}\lambda)$.

The cost of a lookup when we instantiate the bins with Cuckoo bins changes since a lookup in a Cuckoo bin without a stash has $O(\log\log\lambda)$ cost. The cost for the lookup on the two bins in the smallest level is $O(\log N+\log\log\lambda)$ since this level has logarithmic stash. Thus, the total lookup cost is $O(\log N+\log\log\lambda)$. Thus the OHT lookup can be performed at cost $O(\log\log\lambda)$.

Security. Before stating the main lemma proving that the above construction is an oblivious hash table, we prove some lemmas that we will consequently use. First we argue what is the number of levels in out OHT construction.

Lemma 14. The depth of the oblivious hash table is $\mathsf{d}=O(\log\log\lambda)$.

Proof. The size the input array, $\tilde{D}$, of level i is $(2\delta)^{i-1}N$ and we recurse until we reach level of size $\frac{N}{\log\lambda}$. Therefore it will take $O(\log\log\lambda)$ steps to reach this size. On the other hand, note that the size of the accumulated stash $\tilde{S}$ is $O(\log N\cdot R^{\mathsf{real}}/\log^{\mathsf{c}}\lambda)$ which is $O(N/\log^{\mathsf{c}-1}\lambda)$ when $\lambda=\mathsf{poly}(N)$. Therefore, $\mathsf{d}=O(\log\log\lambda)$ for our $\mathsf{OblivHT}$ construction. □

Next we analyze the probability that the algorithm constructing the oblivious hash table fails and aborts.

Lemma 15. For constants $0<\delta<1$ and $\mathsf{c}>1$, algorithm $\mathsf{OblivHT.Build}$ in the above construction aborts with negligible probability in Step 2(e)iii.

Proof. First, we estimate the probability that X_j^{real}, the random variable of the number R_j sampled at Step 2(e)i, is larger than Y_j^{real}, the number of real items assigned to bin B_j during Step 2c. We have $\mathsf{E}[X_j^{\mathsf{real}}]=\log^{\mathsf{c}}\lambda$ and $\mathsf{E}[Y_j^{\mathsf{real}}]=(1-\delta)\log^{\mathsf{c}}\lambda$ and we remind the reader that R^{real} is the number of real items among the total $2R^{\mathsf{real}}$ items of δD received in input by $\mathsf{OblivHT.BuildLevel}$.

Using Chernoff bounds and the bound on the fraction of real items in each level, we estimate the following probabilities:

$$\mathsf{Pr}\left[X_j^{\mathsf{real}}\le(1-\alpha)\log^{\mathsf{c}}\lambda\right]\le e^{-\frac{\alpha^2\log^{\mathsf{c}}\lambda}{2}}=e^{-\omega(\log\lambda)}=\mathsf{negl}(\lambda).$$
$$\mathsf{Pr}\left[Y_j^{\mathsf{real}}\ge(1+\alpha)(1-\delta)\log^{\mathsf{c}}\lambda\right]\le e^{-\frac{\alpha^2(1-\delta)\log^{\mathsf{c}}\lambda}{2}}=e^{-\omega(\log\lambda)}=\mathsf{negl}(\lambda).$$

If $\delta>\frac{2\alpha}{1+\alpha}$ then $(1-\alpha)\log^{\mathsf{c}}\lambda>(1+\alpha)(1-\delta)\log^{\mathsf{c}}\lambda$, hence $\mathsf{Pr}[X_j^{\mathsf{real}}<Y_j^{\mathsf{real}}]=\mathsf{negl}(N)$. Since $0<\frac{2\alpha}{1+\alpha}<1$ for $0<\alpha<1$, we do not have any additional restrictions on the value of δ.

Next, we estimate the number of dummy items that are placed into bin B_j. Let X_j^{dummy} be the random variable denoting the number of dummy items assigned to bin B_j in Step 2c. Let Y_j^{dummy} be the random variable denoting the number of dummy items assigned by the private binomial sampling of Step 2(e)i, which is equal to $\mathsf{thrsh}-R_j$ where $\mathsf{thrsh}=2(1-\delta)\log^{\mathsf{c}}\lambda$. Once again,

$\mathbb{E}[X_j^{\mathsf{dummy}}] = \log^{\mathsf{c}} \lambda$ and $\mathbb{E}[Y_j^{\mathsf{dummy}}] = \mathsf{thrsh} - \mathbb{E}[R_j] = (1-\delta)\log^{\mathsf{c}} \lambda$. Using the same analysis as above, it can be shown that the number of dummy items required is always satisfied except with probability negligible in N. $\square$

With the above lemmas, we are ready to state our main security theorem about the oblivious hash table construction.

Theorem 16. Construction 13 is an oblivious hash table according to Definition 4.

Proof. We construct a simulator $\mathsf{Sim} = (\mathsf{Sim}_{\mathsf{Build}}(N), \mathsf{Sim}_{\mathsf{Lookup}}(), \mathsf{Sim}_{\mathsf{Extract}}(N))$ for the above construction as follows:

- $\mathsf{Sim}_{\mathsf{Build}}(N)$ runs the $\mathsf{OblivHT.Build}$ algorithm using random elements as real elements for the database, and using the simulator for the construction of the oblivious bins.
- $\mathsf{Sim}_{\mathsf{Lookup}}()$ chooses a random bin at each level for the oblivious hash table and runs $\mathsf{Sim}^{\mathsf{B}}_{\mathsf{Lookup}}()$ for that bin.
- $\mathsf{Sim}_{\mathsf{Extract}}(N)$ run the extraction simulator for each bin with input the corresponding thrsh value for that level.

The access pattern in the above simulated distribution is indistinguishable from the real execution for the following reasons:

- The distributions of real items vs random values across bins is indistinguishable since the distributions of the corresponding PRF values are indistinguishable. Also the sampled distributions of real items are indistinguishable when generated from the same number real and dummy items.
- The simulated and the real lookups are indistinguishable as long as the protocol does not abort, which happens with all but negligible probability as we proved in Lemma 15. If we build protocol has completed without abort, the distributions of the real items in bin across each level is independent from the initial distributions of all items across all bins using the PRF and this distributions remains hidden after the completion of $\mathsf{OblivHT.Build}$ since we ensure that each bin contains fewer items than the threshold and we don not reveal how many fewer. Also this distribution of real items across bins is indistinguishable from the distributions where the real items were assigned at random among the bins. The former is the distribution revealed from the real execution lookups, and the later is the distribution revealed by the simulated queries.
- The indistinguishability of the real and the ideal extraction follows from the indistinguishability of the bin extractors, which are executed on public values dependent only on the size of the database. We note that the extract procedure of the OHT returns in total N elements: the extract procedure in each level returns total of $2(1-\delta)R^{\mathsf{real}}$ elements from the extractions across all bins, and the size of the $\mathsf{D}_{\mathsf{over}} = 2\delta R^{\mathsf{real}}$ for the next recursive call. If the bin in the final level is initialized with R' items, then the extract algorithm for that bin will also return R' items. Thus, the total size of the data returned from extract is N elements. Furthermore, exactly $R'/2$ items will be with tag 0 and the remaining $R'/2$ items will be with tag 1 denoting real and dummy items.

- We will argue the items in the output of the extraction algorithm are shuffled under a permutation chosen uniformly at random. We can consider the decomposition of this permutation as defined in Lemma 21 of Appendix A. The unqueried items are distributed according to a PRF into different bins in $\mathsf{OblivHT.BuildLevel}$. Since $\mathsf{OblivHT.Extract}$ only deals with unqueried items, the distribution of unqueried items corresponds to the unrevealed outputs of PRF evaluations with a secret key. Therefore, the distribution of unqueried items is indistinguishable from sampling a random Assign function. Finally, the bin extraction methods will apply the local permutations to unqueried items within each bin. Therefore, by Lemma 21, the final permutation of unqueried and dummy items after $\mathsf{OblivHT.Extract}$ is indistinguishable from a permutation chosen uniformly at random.

□

5 Oblivious RAM Construction

In this section we present our ORAM construction, which follows the hierarchical blueprint. It uses our OHT to store the data assigned to each level as well as our oblivious random multi-array shuffle to merge the content of consecutive levels when moving data from smaller to bigger levels in the ORAM. Note that our $\mathsf{OblivHT}$ construction from above only works for inputs with at least $O(\log^7 n)$ items. In our ORAM construction, whenever we instantiate an $\mathsf{OblivHT}$ with less than $O(\log^7 n)$ items, we will use the original $\mathsf{OblivHT}$ constructions from the work of Goodrich and Mitzenmacher [22] which uses simple hashing as well as oblivious sorting over the entire input array. The extract functionality for these $\mathsf{OblivHT}$ constructions are implemented by using an oblivious sort over the hash table. Using these two $\mathsf{OblivHT}$ constructions, we obtain the following result:

Theorem 17. Assuming the existence of a PRF, Construction 18 is an Oblivious RAM with $O(\log N \cdot \log\log N)$ amortized block communication cost per query.

Construction 18. Let $\mathsf{OblivHT} = (\mathsf{OblivHT.Init}, \mathsf{OblivHT.Build}, \mathsf{OblivHT.Lookup}, \mathsf{OblivHT.Extract})$ be an oblivious hash table and $\mathsf{OblMultArrShuff}$ be an oblivious random multi-array shuffle. Let $\mathcal{U} = \mathcal{U}_{\mathsf{real}} \cup \mathcal{U}_{\mathsf{dummy}} \cup \mathcal{U}_{\mathsf{query}}$, where the items of the database come from $\mathcal{U}_{\mathsf{real}}$, items used for padding in the construction come from $\mathcal{U}_{\mathsf{dummy}}$ and dummy queries use values from $\mathcal{U}_{\mathsf{query}}$. We construct an oblivious RAM scheme $\mathsf{ORAM} = (\mathsf{ORAM.Init}, \mathsf{ORAM.Access})$ as follows:

$\mathsf{ORAM.Init}$. This algorithm initializes an ORAM memory structure using an input database.

$(\tilde{D}, \mathsf{st}) \leftarrow \mathsf{ORAM.Init}(1^\lambda, D)$:

1. Create a hierarchy of $\log N$ levels $\{\ell_i\}_{i=\alpha}^{\log N}$ of size 2^i where N is the size of D and $2^\alpha = O(\log N)$.
2. Initialize an oblivious hash table that contains the whole database by running

$$(\mathsf{st}_{\mathsf{Init}}, \tilde{D}') \leftarrow \mathsf{OblivHT.Init}(D)$$
$$(\mathsf{st}_{\mathsf{Build}}, \tilde{H}, \tilde{S}) \leftarrow \mathsf{OblivHT.Build}\left(\tilde{D}, \mathsf{st}_{\mathsf{Init}}\right).$$

3. Set the first level to be the array representing the stash for the oblivious hash table $\ell_\alpha \leftarrow \tilde{S}$; the last level to be the new oblivious hash table $\ell_{\log N} \leftarrow (\mathsf{st}_{\mathsf{Build}}, \tilde{H})$; and all other levels to be empty $\{\ell_i \leftarrow (\perp, \perp)\}_{i=\alpha+1}^{\log N-1}$.
4. Set $\tilde{D} \leftarrow \{\ell_i\}_{i=\alpha}^{\log N}$. Output $(\tilde{D}, \mathsf{st})$ where st contain the secret encryption key in $\mathsf{st}_{\mathsf{Build}}$.

ORAM.Access. This algorithm takes as input a database access instruction and executes it in an oblivious way having access to an ORAM memory structure.

$(v, \mathsf{st}) \leftarrow \mathsf{ORAM.Access}(\mathsf{st}, \tilde{D}, \mathsf{I}, \mathsf{cnt})$:

1. If $\mathsf{cnt} = 2^j$ for some j, invoke $\mathsf{ORAM.Shuffle}(\{\ell_i\}_{i=\alpha}^{j})$ defined below.
2. Parse $\tilde{D} \leftarrow \{\ell_i\}_{i=k}^{\log N}$ where $\ell_i = (\mathsf{st}_i, \tilde{H}_i)$ and $\mathsf{I} = (\mathsf{op}, \mathsf{addr}, \mathsf{data})$.
3. Set flag $\mathsf{found} = 0$.
4. Do a linear scan on the items (k_j, v_j) stored in the array stored in the smallest level ℓ_α looking for item with $k = \mathsf{addr}$. If such an item is found, set $\mathsf{found} = 1$.
5. For $i = \alpha + 1$ to $\log N$, do the following:
   (a) If $\mathsf{found} = 1$, then sample a random query from $k \leftarrow \mathcal{U}_{\mathsf{query}}$ [2]. Otherwise, set $k \leftarrow \mathsf{addr}$.
   (b) Do a lookup in the oblivious hash table at that level:
   $$(v, \tilde{H}_i', \perp, \mathsf{st}_i') \leftarrow \mathsf{OblivHT.Lookup}(k, \tilde{H}_i, \perp, \mathsf{st}_i).$$
   (c) Update $\ell_i \leftarrow (\mathsf{st}_i', \tilde{H}_i')$
   (d) If $\mathsf{op} = \mathsf{read}$, the set $d = \mathsf{Enc}(\mathsf{SK}, (\mathsf{addr}, v))$, else if $\mathsf{op} = \mathsf{write}$, set $d = \mathsf{Enc}(\mathsf{SK}, (\mathsf{addr}, \mathsf{data}))$. Append d to the array stored in ℓ_α where SK is stored in st.

ORAM.Shuffle. This algorithm shuffles together the data from a number of consecutive levels in the ORAM hierarchical memory structure.

$\tilde{D} \leftarrow \mathsf{ORAM.Shuffle}(\{\ell_i, \mathsf{st}\}_{i=\alpha}^{j})$:

1. Append to the smallest level ℓ_α, $2^\alpha + 1$ dummy items from $\mathcal{U}_{\mathsf{dummy}}$ and then apply an oblivious shuffle on the resulting set of real and dummy items and set $\tilde{A}_\alpha$ to be the output of the oblivious shuffle.
2. If $j = \log N$, then we are shuffling all levels.
   (a) Use an oblivious sort on all $\Theta(N)$ input items in $\ell_\alpha, \ldots, \ell_{\log N}$. This oblivious sort will remove any extra dummy items inserted by Oblivious Bin lookups and only maintain the required N dummy items for $\mathsf{OblivHT.Build}$. After oblivious sorting, we get an array $\tilde{A}_{\log N}$ which contains $2N$ items with the N real items and N dummy items shuffled randomly.

[2]We assume that dummy queries do not repeat. We can enforce this either by setting the universe $\mathcal{U}_{\mathsf{query}}$ to be large enough or by keeping a counter for the dummy queries.

(b) Execute $(\mathsf{st}_{\log N}, \tilde{H}_{\log N}, \tilde{S}_{\log N}) \leftarrow \mathsf{OblivHT.Build}(\tilde{D}, \mathsf{st})$.

(c) Set $\ell_{\log N} \leftarrow (\mathsf{st}_{\log N}, \tilde{H}_{\log N})$.

(d) Set $\ell_\alpha \leftarrow \tilde{S}_{\log N}$ and all intermediate levels $\{\ell_i \leftarrow \perp\}_{i=\alpha+1}^{\log N}$ to be empty.

(e) Return $\tilde{D} \leftarrow \{\ell_i\}_{i=\alpha}^{\log N}$.

3. Otherwise when $j < \log N$:

(a) For $i = \alpha + 1$ to j: parse $\ell_i = (\mathsf{st}_i, \tilde{H}_i)$

$$(\tilde{A}_i, \mathsf{st}'_i) \leftarrow \mathsf{OblivHT.Extract}(\tilde{H}_i, \mathsf{st}_i).$$

(b) Run the multi-array shuffle algorithm on the data extracted from each level oblivious hash table

$$\tilde{D} \leftarrow \mathsf{OblMultArrShuff}\left(\{\tilde{A}_i\}_{i=\alpha}^{j}\right).$$

(c) Construct an oblivious hash table using the output from the multi-array shuffle

$$(\mathsf{st}_j, \tilde{H}_j, \tilde{S}_j) \leftarrow \mathsf{OblivHT.Build}\left(\tilde{D}, \mathsf{st}\right).$$

(d) Set $\ell_j \leftarrow (\mathsf{st}_j, \tilde{H}_j)$, $\ell_\alpha \leftarrow \tilde{S}_j$ and all intermediate levels to be empty $\{\ell_i \leftarrow \perp\}_{i=\alpha+1}^{j-1}$.

(e) Return $\tilde{D} \leftarrow \{\ell_i\}_{i=\alpha}^{\log N}$.

Correctness. The $\mathsf{OblivHT.Build}$ algorithm expects input that has equal numbers of real and dummy items. Thus, we need to argue that the input array to the multi-array shuffle contain equal numbers of real and dummy value. The $\mathsf{OblivHT.Extract}$ functionality returns an array of size equal to the capacity for real items of the oblivious hash table. The size of of $\tilde{A}_\alpha$ is $2^{\alpha+1} + 1$ and the size of each $\tilde{A}_{\alpha+i}$ for $1 \leq i \leq t - 1$ is $2^{\alpha+i}$. Thus, total size of $\sum_{i=1}^{t} |\tilde{A}_{\alpha+i}|$ will be $2^{\alpha+1} + \sum_{i=\alpha+1}^{\alpha+t} 2^i = \sum_{i=1}^{\alpha+t} 2^i - \sum_{i=1}^{\alpha} 2^i = 2^{\alpha+1} + 2^{\alpha+t+1} - 2^{\alpha+1} = 2^{\alpha+t+1}$. At the same time the total number of real items across the first t levels is at most $2^{\alpha+t}$. By adding a number of real items on the order of the smallest level, we can ensure the number of real items is exactly half of the size of the array $\tilde{D}$ from the multi-array shuffle, which is correct input for $\mathsf{OblivHT.Build}$.

Theorem 19. Construction 18 is an access pattern hiding oblivious RAM scheme according to Definition 1.

Proof. We construct a simulator for the ORAM scheme as follows. For the initialization phase, $\mathsf{Sim}_{\mathsf{Init}}$ just invokes the OHT simulators $\mathsf{Sim}_{\mathsf{Init}}$ and $\mathsf{Sim}_{\mathsf{Build}}$. The query simulator $\mathsf{Sim}_{\mathsf{Access}}$ uses the query simulators $\mathsf{Sim}_{\mathsf{Lookup}}$ for each oblivious hash table in each level of the ORAM. To simulate the shuffling procedures $\mathsf{Sim}_{\mathsf{Access}}$ uses the OHT $\mathsf{Sim}_{\mathsf{Extract}}$ and then the multi-array shuffle simulator $\mathsf{Sim}_{\mathsf{MultArrShuff}}$. Note that the inputs for both the extraction and the shuffle simulators are deterministically depending only on the size of the database. The indistinguishability of the simulation from the real execution follows from the simulators that we used and the fact that their inputs are completely defined by the size of the database. □

Efficiency. We evaluate the amortized query complexity over N ORAM accesses on database of size N, assuming that $N = \mathsf{poly}(\lambda)$. First, we deal with the case when we shuffle all levels as it is a special case. When shuffling all levels, we employ an oblivious sort on $\Theta(N)$ items resulting in $\Theta(N \log N)$ bandwidth overhead. However, this happens ever $\Theta(N)$ operations meaning an amortized cost of $\Theta(\log N)$. For all smaller levers, after each 2^i accesses we reshuffle the top i levels using the oblivious random multi-array shuffle. Thus, we have the following total cost of the shuffle for all levels with at least $\log^8 N$ items:

$$O\left(N' \log\log\lambda + \frac{N' \log N'}{\log \lambda}\right) = O(N \log\log\lambda), \text{where } N' = \sum_{i=\log\alpha}^{\log N} 2^i,$$

which results in amortized cost of $O(\log\log\lambda)$ blocks per access from the shuffle. The cost for a look up across all levels consists of the scan on the smallest level which takes $O(\log N)$ plus the cost of the OHT without stash access for each level, which is $O\left(\log N \log\log\lambda\right)$ when we use Cuckoo bins in the OHT. For all levels with less than $\log^8 N$ items, we use the $\mathsf{OblivHT}$ built using simple hashing and oblivious sorts over entire input arrays, which costs $\log N$ blocks for each lookup as well as $O(\log N \log\log N)$ blocks during shuffling. As there are only $O(\log\log N)$ levels with less than $\log^8 N$ items, the increased cost of these smaller levels do not affect the total complexity of our ORAM. Therefore, the amortized complexity of a lookup is $O\left(\log N \log\log N\right)$ when $\lambda = \mathsf{poly}(N)$.

5.1 Optimization Discussion

In our hierarchical ORAM construction each level has capacity to hold all preceding smaller levels and each level stores all blocks using a single oblivious hash table. Kushilevitz *et al.* [26] presented an optimization to hierarchical ORAM constructions where each level has capacity to hold K times the joint capacity of all smaller levels. Each level is implemented using K disjoint oblivious hash tables. When a level gets full, then the level's contents are moved to an empty oblivious hash table of the next larger level of the hierarchical ORAM. This optimization decreases the amortized shuffling cost by a factor of $O(\log K)$ at the cost of increasing the online cost of ORAM queries by a factor of $O(\frac{K}{\log K})$ and it is effective only when the amortized shuffling cost is larger than the online query costs (like for example in [23]).

For our construction, the two effects resulting from this modification are: first, an ORAM query must perform K oblivious hash table accesses per level and, secondly, the number of levels in the hierarchy decreases from $O(\log N)$ to $O(\frac{\log N}{\log K})$. The online bandwidth of an ORAM query becomes $O(\frac{K \log N \log\log\lambda}{\log K})$ or $O(\frac{K \log N (\log\log\lambda)^2}{\log K})$ blocks depending on the underlying oblivious bin scheme used. The amortized bandwidth of shuffling becomes $O(\frac{\log N \log\log\lambda}{\log K})$. We observe that the bandwidth is minimized when $K = O(1)$, which results in the same asymptotic overhead as the construction in Section 5. Therefore, our ORAM construction does not benefit from this optimization.

Acknowledgements

The authors would like to thank Enoch Pesorico for pointing out a bug when generating binomial variates.

References

[1] M. Ajtai. Oblivious RAMs without cryptogrpahic assumptions. In *STOC*, pages 181–190, 2010.

[2] M. Ajtai, J. Komlós, and E. Szemerédi. An 0 (n log n) sorting network. In *Proceedings of the fifteenth annual ACM symposium on Theory of computing*, pages 1–9. ACM, 1983.

[3] K. E. Batcher. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, spring joint computer conference*, pages 307–314. ACM, 1968.

[4] E. Boyle, K.-M. Chung, and R. Pass. Oblivious parallel ram and applications. In E. Kushilevitz and T. Malkin, editors, *Theory of Cryptography*, pages 175–204, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[5] E. Boyle and M. Naor. Is There an Oblivious RAM Lower Bound? In *ITCS*, pages 357–368, 2016.

[6] D. Cash, P. Grubbs, J. Perry, and T. Ristenpart. Leakage-abuse attacks against searchable encryption. In *CCS*, 2015.

[7] T. H. Chan, Y. Guo, W. Lin, and E. Shi. Oblivious hashing revisited, and applications to asymptotically efficient ORAM and OPRAM. In *ASIACRYPT*, pages 660–690, 2017.

[8] T. H. Chan, Y. Guo, W.-K. Lin, and E. Shi. Cache-oblivious and data-oblivious sorting and applications. In *SODA*, pages 2201–2220, 2018.

[9] T.-H. H. Chan, K.-M. Chung, and E. Shi. On the depth of oblivious parallel RAM. In T. Takagi and T. Peyrin, editors, *ASIACRYPT*, pages 567–597, 2017.

[10] T.-H. H. Chan and E. Shi. Circuit OPRAM: Unifying statistically and computationally secure orams and oprams. In *Theory of Cryptography Conference*, pages 72–107. Springer, 2017.

[11] B. Chen, H. Lin, and S. Tessaro. Oblivious parallel ram: Improved efficiency and generic constructions. In E. Kushilevitz and T. Malkin, editors, *Theory of Cryptography*, pages 205–234, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[12] K. Chung, Z. Liu, and R. Pass. Statistically-secure ORAM with õ($\log^2$ n) overhead. In P. Sarkar and T. Iwata, editors, *ASIACRYPT 2014*, pages 62–81, 2014.

[13] I. Damgård, S. Meldgaard, and J. B. Nielsen. Perfectly secure oblivious ram without random oracles. In *TCC*, pages 144–163, 2011.

[14] S. Devadas, M. van Dijk, C. W. Fletcher, L. Ren, E. Shi, and D. Wichs. Onion ORAM: A constant bandwidth blowup oblivious RAM. In *TCC*, pages 145–174, 2015.

[15] L. Devroye. *Non-Uniform Random Variate Generation.* Springer-Verlag, 1986.

[16] J. Doerner and A. Shelat. Scaling oram for secure computation. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 523–535, New York, NY, USA, 2017. ACM.

[17] C. Gentry, K. A. Goldman, S. Halevi, C. Julta, M. Raykova, and D. Wichs. Optimizing ORAM and using it efficiently for secure computation. In *PoPETS*, pages 1–18, 2013.

[18] O. Goldreich. Towards a Theory of Software Protection and Simulation by Oblivious RAMs. In *STOC*, 1987.

[19] O. Goldreich and R. Ostrovsky. Software Protection and Simulation on Oblivious RAMs. *J. ACM*, 43(3), 1996.

[20] M. T. Goodrich. Randomized Shellsort: A simple oblivious sorting algorithm. In *SODA*, pages 1262–1277, 2010.

[21] M. T. Goodrich. Zig-zag sort: A simple deterministic data-oblivious sorting algorithm running in O(n log n) time. In *STOC*, pages 684–693, 2014.

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

[23] M. T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia. Oblivious RAM Simulation with Efficient Worst-case Access Overhead. In *CCSW*, pages 95–100, 2011.

[24] S. D. Gordon, J. Katz, V. Kolesnikov, F. Krell, T. Malkin, M. Raykova, and Y. Vahlis. Secure two-party computation in sublinear (amortized) time. In *CCS*, pages 513–524, 2012.

[25] M. S. Islam, M. Kuzu, and M. Kantarcioglu. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *NDSS*, 2012.

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

[27] K. G. Larsen and J. B. Nielsen. Yes, there is an oblivious RAM lower bound! Cryptology ePrint Archive, Report 2018/423, 2018. https://eprint.iacr.org/2018/423.

[28] T. Moataz, T. Mayberry, and E.-O. Blass. Constant communication ORAM with small blocksize. In *CCS*, pages 862–873, 2015.

[29] O. Ohrimenko, M. T. Goodrich, R. Tamassia, and E. Upfal. The melbourne shuffle: Improving oblivious storage in the cloud. In *International Colloquium on Automata, Languages, and Programming*, pages 556–567. Springer, 2014.

[30] R. Ostrovsky. Efficient computation on oblivious RAMs. In *STOC*, pages 514–523, 1990.

[31] R. Ostrovsky and V. Shoup. Private information storage (extended abstract). In *STOC*, 1997.

[32] S. Patel, G. Persiano, M. Raykova, and K. Yeo. PanORAMa: Oblivious RAM with logarithmic overhead. Cryptology ePrint Archive, Report 2018/373, 2018. https://eprint.iacr.org/2018/373.

[33] S. Patel, G. Persiano, and K. Yeo. Cacheshuffle: An oblivious shuffle algorithm using caches. *CoRR*, abs/1705.07069, 2017.

[34] S. Patel, G. Persiano, and K. Yeo. CacheShuffle: A Family of Oblivious Shuffles. In *ICALP*, pages 161:1–161:13, 2018.

[35] B. Pinkas and T. Reinman. Oblivious RAM revisited. In *CRYPTO*, pages 502–519, 2010.

[36] L. Ren, C. Fletcher, A. Kwon, E. Stefanov, E. Shi, M. van Dijk, and S. Devadas. Constants count: Practical improvements to oblivious RAM. In *USENIX Security*, pages 415–430, 2015.

[37] E. Shi, T. H. H. Chan, E. Stefanov, and M. Li. Oblivious RAM with $O(\log^3 n)$ worst-case cost. In D. H. Lee and X. Wang, editors, *ASIACRYPT 2011*, pages 197–214, 2011.

[38] E. Stefanov, M. van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas. Path ORAM: An Extremely Simple Oblivious RAM Protocol. In *CCS '13*, pages 299–310, 2013.

[39] A. Waksman. A permutation network. *Journal of the ACM (JACM)*, 15(1):159–163, 1968.

[40] X. Wang, H. Chan, and E. Shi. Circuit ORAM: On Tightness of the Goldreich-Ostrovsky Lower Bound. In *CCS*, pages 850–861, 2015.

[41] X. S. Wang, Y. Huang, T.-H. H. Chan, A. Shelat, and E. Shi. Scoram: Oblivious ram for secure computation. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 191–202, New York, NY, USA, 2014. ACM.

[42] S. Zahur, X. S. Wang, M. Raykova, A. Gascón, J. Doerner, D. Evans, and J. Katz. Revisiting square-root ORAM: efficient random access in multi-party computation. In *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*, pages 218–234, 2016.

A Permutation Decomposition

Oblivious shuffle algorithms are a central building block in many ORAM constructions. The oblivious properties of these algorithms are very strong and they guarantee that the output is shuffled under a random permutation that the adversary server does not learn anything about even when he knows in the input in the clear. However, in many cases, oblivious shuffles are used as intermediate steps of ORAM constructions when the server does not know the input for the shuffles in the clear. Often, parts of the input are still in a shuffled state from a previous step in the ORAM algorithm. This means that the input has more entropy to the adversary compared to the setting where the input is given in the clear by the server. In our constructions, we will leverage this input entropy in order to achieve oblivious properties for a random shuffle with better efficiency.

In the first decomposition, we assume that we are given a fixed partition of $[N]$ into $A_1, \ldots, A_L$. To generate a random permutation, one can permute each of $A_1, \ldots, A_L$ according to randomly generated $\tau_1, \ldots, \tau_L$ and then randomly mix the elements of L into a single array. If the permutations of the partitions and mixing are performed obliviously, the result permutation is indistinguishable from a random permutation by any PPT adversary even if the adversary knows the initial partition $A_1, \ldots, A_L$. This decomposition is used by our multi-array shuffle algorithm, which presents an oblivious shuffle algorithm on input that consists of several smaller arrays each of which are shuffled according to a random permutation hidden from the adversary. We can think of this input as an intermediate state of the shuffle where the τ_i permutations have been already applied and the algorithm completes the second part of the permutation by obliviously mixing the different arrays.

Our second decomposition instead simply receives a set of integers $N_1, \ldots, N_L$ that sum up to N. One can generate a random permutation by randomly partitioning $[N]$ into the sets $A_1, \ldots, A_L$ each with size $N_1, \ldots, N_L$ respectively, each of the sets are permuted using randomly chosen permutations $\tau_1, \ldots, \tau_L$ and the permuted arrays are, simply, concatenated. Our oblivious hash table constructions will mirror this decomposition. In the initialization of oblivious hash tables, items are distributed according to a random partitioning. Items remain in this state while the hash table is processing queries. Unqueried items remain randomly partitioned independent of the access patterns seen during the initialization and queries execution, and therefore remains hidden from the adversary. This intermediate state of the shuffle is given as input to the extract function, which applies $\tau_1, \ldots, \tau_L$ to each set completing the random permutation for unqueried items.

First Decomposition. Fix any $N_1, \ldots, N_L$ such that $N = N_1 + \ldots + N_L$ and any partition of $[N]$ into sets $A_1 \cup \ldots \cup A_L = [N]$. We consider the following decomposition of a permutation π into functions $(\mathsf{Assign}, \{\tau_i\}_{i\in[L]})$, where $\mathsf{Assign} : [N] \to [L]$ and $|\mathsf{Assign}^{-1}(j)| = N_j$ for all $j \in [L]$ and $\tau_j : [N_j] \to [N_j]$ for $j \in [L]$. Assign is a random mixing of the L arrays where Assign explicitly states that $\pi(i) \in A_{\mathsf{Assign}(i)}$. Let $\mathsf{Assign}^{-1}(i)$ be ordered in an increasing order and let $\mathsf{Assign}^{-1}(i)[j]$ be the j-th largest item. We define π as follows:

$$\pi(i) = \mathsf{Assign}^{-1}(l)[\tau_l(k)] \text{ where } i \in A_l \text{ and } k = |\{j \;:\; j < i \text{ and } j \in A_l\}|.$$

In the above decomposition, Assign allocates exactly N_i locations for items of A_i, while τ_i describes the arrangement of the items of A_i within the N_i locations. In words, π assigns the k-th largest item of A_l to the $\tau_l(k)$-th largest index of $\mathsf{Assign}^{-1}(l)$. This process is well defined since $|\mathsf{Assign}^{-1}(l)| = |A_i|$.

We argue in the next lemma that we can sample a uniform permutation π by sampling uniform assignment function, Assign and uniform permutations $\{\tau_i\}_{i\in[L]}$.

Lemma 20. For all $L > 0$, $N_1, \ldots, N_L = N$ and a partitions $A_1 \cup \ldots \cup A_L = [N]$ such that $|A_i| = N_i$ for all $i \in [L]$, the permutation output by the process above is uniformly distributed over the set of permutation over N elements.

Proof. Our proof will proceed in two steps. First, we show that the process has $N!$ possible choices of $(\mathsf{Assign}, \tau_1, \ldots, \tau_L)$ and any two such choices give different permutations as output.

The number of possible choices $(\mathsf{Assign}, \tau_1, \ldots, \tau_L)$ is easily seen to be

$$\binom{N}{N_1, \ldots, N_L} \cdot \prod_{i\in[L]} N_i! = N!.$$

Let $(\mathsf{Assign}, \tau_1, \ldots, \tau_L) \neq (\mathsf{Assign}', \tau_1', \ldots, \tau_L')$ and let π and π' be the associated permutations. Suppose $\mathsf{Assign}(i) \neq \mathsf{Assign}'(i)$ for some $i \in [N]$. Then $\pi(i) \in A_{\mathsf{Assign}(i)}$ while $\pi'(i) \in A_{\mathsf{Assign}'(i)}$, which implies that π and π' are different. Suppose instead Assign and Assign' coincide on all inputs. Then, there exists $j \in [L]$ and $k \in [N_j]$ such that $\tau_j(k) \neq \tau_j'(k)$. In π, the k-th largest item in A_j will be placed into $\mathsf{Assign}^{-1}(l)[\tau_j(k)]$ while the same item will be placed into $\mathsf{Assign}^{-1}(l)[\tau_j'(k)]$ meaning π and π' are different. ☐

Second Decomposition. We can also decompose permutations in another way. Fix any $N_1, \ldots, N_L$ such that $N_1 + \ldots + N_L = N$. First, choose a random partitioning $\mathsf{Distribute} : [N] \to [L]$ of the integers in $[N]$ into sets $A_1, \ldots, A_L$ of sizes $N_1, \ldots, N_L$ respectively. In other words, $i \in A_j$ if and only if $\mathsf{Distribute}(i) = j$. Each set A_j is permuted according to a randomly selected permutation τ_j over $[N_j]$. Finally, the permutation π is obtained by concatenating of permuted $A_1, \ldots, A_L$. Formally, π is defined as

$$\pi(i) = \sum_{1 \leq j < \mathsf{Distribute}(i)} N_j + \tau_{\mathsf{Distribute}(i)}(k) \text{ where } k = |\{j : j < i \text{ and } \mathsf{Distribute}(i) = \mathsf{Distribute}(j)\}|.$$

The next lemma proves that π is uniformly distributed over the set of permutation of $[N]$.

Lemma 21. For all $L > 0$, the permutation output by the above process is uniformly distributed over the set of permutations over N elements.

Proof. Again, the number of possible $(\mathsf{Distribute}, \tau_1, \ldots, \tau_j)$ is easily seen to be $N!$ following the same process as Lemma 20.

To show injectivity, pick $(\mathsf{Distribute}, \tau_1, \ldots, \tau_L) \neq (\mathsf{Distribute}', \tau'_1, \ldots, \tau'_L)$ and let π and π' be the associated permutations. Suppose that $\mathsf{Distribute}(i) < \mathsf{Distribute}'(i)$ without loss of generality. We know that $\pi(i) \leq N_1 + \ldots + N_{\mathsf{Distribute}(i)}$ while $\pi'(i) > N_1 + \ldots + N_{\mathsf{Distribute}'(i)}$ meaning that π and π' differ. On the other hand, suppose $\mathsf{Distribute} = \mathsf{Distribute}'$ and $\tau_j(k) \neq \tau'_j(k)$ for some $j \in [L]$ and $k \in [N_j]$. Denote $A_1, \ldots, A_L$ as the sets induced by $\mathsf{Distribute}$ as well as $\mathsf{Distribute}'$. In this case, the k-th largest item of A_j is placed into different locations meaning π and π' differ. $\square$

B Related Work

Since its inception in [18], the concept of an Oblivious RAM has been object of intense study and several constructions with different properties have been proposed. As we have already pointed out above, our construction is in the most general model in which no assumption is made on the block size and the server is considered a storage device and not required to perform any computation on the blocks. The constructions of [19, 35, 22, 26] are all in the same general model as ours. Weakening any of these assumptions leads to more efficient, albeit less general, constructions. Specifically, Circuit ORAM [40] has overhead $O(\alpha(N) \cdot \log N)$ for blocks of size N^ϵ, for any $\alpha(N) = \omega(1)$ and constant ϵ. If the server is also considered to have some computational ability then the overhead can be reduced even further. Onion Oblivious RAM of [14] only requires a constant overhead provided that the blocks have size $\Omega(\log^6 N)$. This requirement was reduced to $\Omega(\log^4 N)$ by [28]

The logarithmic lower bound of Goldreich and Ostrovsky [19] only applies to statistically secure construction in balls and bins model, as pointed out by Boyle and Naor [5]. Specifically, this model only allows data to be shuffled around while not allowing sophisticated data encodings. Furthermore, Boyle and Naor show that without obtaining currently unknown superlinear lower bounds in sorting circuit sizes, there is no hope of getting lower bounds for Oblivious RAM in the general setting.

The client of a hierarchical Oblivious RAM typically uses a Pseudo-Random Function to remember where each block is stored and this is the main reason why they manage to guarantee only computational security. In the original construction of Goldreich and Ostrovsky [19], the need for a PRF can be removed, thus obtaining statistical security in the balls and bins model where blocks are modeled as indistingushable blobs of data, provided that the client has access to a private random function. Damgård *et al.* [13] and Ajtai [1] gave statistically secure constructions in the standard model with a overhead of $O(\sqrt{N} \cdot \log^2 N)$. The tree-based approach, pioneered by Shi et al. [37] instead does not need to remember where each block is stored; for example, Path ORAM [38] stores the position map recursively in smaller Oblivious RAMs. Thus, they offer statistical security with polylogarithmic overhead. Gentry *et al.* [17] improved the overhead to $O(\frac{\log^3 N}{\log \log N})$. Currently, the most efficient construction with statistical security is Path ORAM with $O(\log^2 N)$ overhead. The construction of Chung *et al.* [12] has a slightly worse overhead of $O(\log^2 N \cdot \log \log N)$ but a much simpler proof of security and performance. Further work by Ren *et al.* [36] improve the hidden constants of tree-based constructions.

Boyle *et al.* [4] introduced the notion of Oblivious Parallel RAM, which modeled the scenario of multiple processes accessing shared memory in parallel. Further works [11, 10, 7] have improved the overhead of Oblivious Parallel RAMs. Chan *et al.* [9] present lower bounds on the depth of Oblivious Parallel RAMs.

In recent years, the problem of using Oblivious RAM in multiparty computation has been heavily studied. Gordon *et al.* [24] showed that Oblivious RAM could be used to perform two-party computation in sublinear amortized time. Many works improving the asymptotics of Oblivious RAM for multiparty computation were presented [41, 40]. Further works have shown that considering ORAM constructions with worse asymptotics but smaller hidden constants can lead to better practical efficiency [42, 16].

Data-oblivious sorting is an important building block of many previous Oblivious RAM constructions. Batcher [3] presented an $O(N \log^2 N)$-sized sorting circuit. Ajtai *et al.* [2] presented an $O(N \log N)$-sized sorting circuit, however with very large constants. Goodrich [20] presented an $O(N \log N)$ randomized sorting algorithm with smaller constants, but larger depth. Afterwards, Goodrich [21] presented a deterministic $O(N \log N)$ sorting algorithm with similarly smaller constants, but large depth. Chan *et al.* [8] present a cache-oblivious sorting algorithm. Data-oblivious shuffling is another primitive regularly used in Oblivious RAMs. Waksman [39] presented an $O(N \log N)$ permutation network. Ohrimenko *et al.* [29] present an oblivious shuffle using $O(N \log_C N)$ bandwidth when the client has C blocks of storage available. Patel *et al.* [33] improve the hidden constants of the oblivious shuffle. In the same work, the notion of K-oblivious shuffling is introduced, which generalizes the knowledge of the adversary with respect to the input. In particular it assumes that out of the N blocks that are input for the shuffle the adversary knows the identities of only K. Leveraging the entropy of the remaining $N - K$ blocks the K-oblivious shuffle $2N + O(K \log_C K)$ bandwidth for any client with C blocks of storage available. Our multi-array shuffle uses ideas from the K-oblivious shuffle as starting point but achieves better asymptotic complexity and handles much more complex input entropy, which is crucial for its application to ORAM that achieves the new best asymptotic communication complexity.