PanORAMa: Oblivious RAM with Logarithmic Overhead

Sarvar Patel*, Giuseppe Persiano†, Mariana Raykova‡, Kevin Yeo*
*Google LLC, {sarvar, kwlyeo}@google.com
†Google LLC and Università di Salerno, giuper@gmail.com
‡Google LLC and Yale University, mariana.raykova@yale.edu

***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 $O\left(\log N + \mathsf{poly}(\log\log\lambda)\right)$ 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.

I. 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 [1], [2]. 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 [3], [4], [5].

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 [6], and it has turned ORAM into one of the classical cryptographic concepts. The seminal works of Goldreich and Ostrovsky [6], [7], [8] 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 [9]. A recent work of Larsen and Nielsen [10] 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 [8] 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 [9] 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 [10] proved the best ORAM lower bound which

2575-8454/18/$31.00 ©2018 IEEE
DOI 10.1109/FOCS.2018.00087

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 [11].

As we discussed above, the lower bound results apply to all algorithms that work for any block size. While there are constructions [12], [5] 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 [13], [14], [15], [16], [17] that follow the hierarchical blueprint introduced in the work of Goldreich and Ostrovsky [7], [8], and constructions [18], [19], [12], [20], [5], [21], [22] that follow the tree-based template with a recursive position map, which was introduced in the work of Shi et al. [18].

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 [8] used pseudorandom functions (PRFs) resulting in $O(\log^3 N)$ amortized communication overhead. Later, the work of Pinkas and Reinman [13] 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 [16] 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. [17] devised a balancing scheme that further improved the bandwidth to $O(\log^2 N/\log\log N)$. The work of Chan et al. [22] 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. [18] that pioneered the tree-based approach achieved $O(\log^3 N)$ communication. Gentry *et al.* [19] improved the overhead to $O(\frac{\log^3 N}{\log\log N})$. Currently the most efficient tree-based construction is the Path ORAM construction [12], 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 [5], 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 [11], [23] 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

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

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 [9] 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 [8]). 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 [8]. Our construction is also $O(\log\log N)$ away from the lower bound by Larsen and Nielsen [10] 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 [22] 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.
- *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 [8] (see Chan et al. [22] 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 [24], [25], [26], [27] 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 [28].

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 IV.

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 III.

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 IV.

Oblivious Hash Table. Our oblivious hash table construction is inspired by the two-tier hash scheme proposed by Chan et al. [29], 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. [29] 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 [30] we present two instantiations for the oblivious bin using a binary tree and 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.* [17], 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 II our new definitions of oblivious hash table and oblivious random multi-array shuffle. Section III describes our construction of an oblivious random multi-array shuffle. We describe our general oblivious hash table construction in Section IV. Finally, we describe our ORAM construction in Section V. In the full version [30], we provide security proofs, efficiency analysis and further discussion of our constructions.

II. DEFINITIONS

In this section we present only the main definitions for the new primitives that we introduce in our work. We refer the reader to the full version [30] for the standard definition of oblivious RAM as well as additional intuition for some of the new definitions.

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.

A. 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 1 (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 $(|\mathsf{A}_1|, \ldots, |\mathsf{A}_L|)$, such that for any PPT adversary

$\mathbf{Expt}_{\mathcal{A}}^{\mathsf{Real,OblMultArrShuff}}(\lambda)$	$\mathbf{Expt}_{\mathsf{Sim},\mathcal{A}}^{\mathsf{Ideal,OblMultArrShuff}}(\lambda)$
$(\mathsf{A}_1, \ldots, \mathsf{A}_L, \mathsf{st}_{\mathcal{A}}) \leftarrow \mathcal{A}_1(1^\lambda)$	$(\mathsf{A}_1, \ldots, \mathsf{A}_L, \mathsf{st}_{\mathcal{A}}) \leftarrow \mathcal{A}_1(1^\lambda)$
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 $\mathsf{OblMultArrShuff}$.

algorithm $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$:

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

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.

B. *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 2 (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 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 $n = \mathsf{poly}(\lambda)$,

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

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.

III. 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

$\mathbf{Expt}_{\mathcal{A}}^{\mathsf{Real,OblivHT}}(\lambda, n)$	$\mathbf{Expt}_{\mathsf{Sim},\mathcal{A}}^{\mathsf{Ideal,OblivHT}}(\lambda, n)$
$(D, \mathsf{st}_{\mathcal{A}}) \leftarrow \mathcal{A}_1(1^\lambda)$	$(D, \mathsf{st}_{\mathcal{A}}) \leftarrow \mathcal{A}_1(1^\lambda)$
$\chi_{\mathsf{Init}} \leftarrow \mathsf{Addrs}[(\tilde{D}, \mathsf{st}) \leftarrow \mathsf{OblivHT.Init}(D)]$	$\chi_{\mathsf{Init}} \leftarrow \mathsf{Addrs}[(\tilde{D}, \mathsf{st}_{\mathsf{Sim}}) \leftarrow \mathsf{Sim}_{\mathsf{Init}}(\|D\|)]$
$\chi_{\mathsf{Build}} \leftarrow \mathsf{Addrs}[(\tilde{H}, \mathsf{st}) \leftarrow \mathsf{OblivHT.Build}(D, \mathsf{st})]$	$\chi_{\mathsf{Build}} \leftarrow \mathsf{Addrs}[(\tilde{H}, \mathsf{st}_{\mathsf{Sim}}) \leftarrow \mathsf{Sim}_{\mathsf{Build}}(\|D\|)]$
$\chi_{\mathsf{Lookup}} \leftarrow \perp$, $\chi \leftarrow (\chi_{\mathsf{Init}}, \chi_{\mathsf{Build}})$	$\chi_{\mathsf{Lookup}} \leftarrow \perp$, $\chi \leftarrow (\chi_{\mathsf{Init}}, \chi_{\mathsf{Build}})$
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 $D \setminus \{k_i\}_{i=1}^n$ and the padding dummies in $\tilde{D}$	Let π' be a random permutation on N items
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 OblivHT.

improves on general oblivious sorting algorithms 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 $N_i \ge N_{i+1}$ for $i = 1, \ldots, L$, and there exists 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, $\mathsf{Bin}_1^{\mathsf{in}}, \ldots, \mathsf{Bin}_{\tilde{m}}^{\mathsf{in}}$, 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, $\mathsf{Bin}_1^{\mathsf{out}}, \ldots, \mathsf{Bin}_m^{\mathsf{out}}$, of expected polylog size but slightly smaller than the input bins $\mathsf{Bin}^{\mathsf{in}}$. 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 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 3: [Oblivious Random Multi-Array Shuffle] We define our oblivious random multi-array shuffle algorithm OblMultArrShuff, which will also use algorithms OblMultArrShuff.BinShuffle and OblMultArrShuff.Shuffle as building blocks.

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 $|\mathsf{A}_{\mathsf{cutoff}}| \ge \frac{N}{\log\lambda}$ 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 $\mathsf{Assign} : [N] \to [\mathsf{cutoff}]$ 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.

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 $\mathsf{Bin}^{\mathsf{in}}_1, \ldots, \mathsf{Bin}^{\mathsf{in}}_{\tilde{m}}$ 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 $\mathsf{Bin}^{\mathsf{in}}_1, \ldots, \mathsf{Bin}^{\mathsf{in}}_{\tilde{m}}$ assigning each block a bin at random and recording an encryption of the source array index i. The items of $\mathsf{Bin}^{\mathsf{in}}_1, \ldots, \mathsf{Bin}^{\mathsf{in}}_{\tilde{m}}$ are stored encrypted on the server. Note $\mathsf{Bin}^{\mathsf{in}}_1, \ldots, \mathsf{Bin}^{\mathsf{in}}_{\tilde{m}}$ 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 $\mathsf{Bin}^{\mathsf{out}}_1, \ldots, \mathsf{Bin}^{\mathsf{out}}_m$ with random subsets of pairs from E_ℓ by assigning each pair from E_ℓ to a randomly selected bin. The items of $\mathsf{Bin}^{\mathsf{out}}_1, \ldots, \mathsf{Bin}^{\mathsf{out}}_m$ are stored encrypted on the server. Note the sizes of $\mathsf{Bin}^{\mathsf{out}}_1, \ldots, \mathsf{Bin}^{\mathsf{out}}_m$ 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 $\mathsf{Bin}^{\mathsf{in}}_j$ in D according to the positions specified by Assign in the pairs in $\mathsf{Bin}^{\mathsf{out}}_j$ by running

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

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 $\mathsf{Bin}^{\mathsf{out}}_{\tilde{m}+1}, \ldots, \mathsf{Bin}^{\mathsf{out}}_m$, and set $\mathsf{E}_{\ell+1} := \mathsf{Bin}^{\mathsf{out}}_{\tilde{m}+1} \cup \ldots \cup \mathsf{Bin}^{\mathsf{out}}_m$.
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 OblMultArrShuff.BinShuffle executions above, by running $\mathsf{OblMultArrShuff.Shuffle}(\mathsf{A}_{\ell+1,1}, \ldots, \mathsf{A}_{\ell+1,L}, \mathsf{D}, \mathsf{E}_{\ell+1}, \ell + 1)$.

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

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

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 $\mathsf{Bin}^{\mathsf{in}}$.
2) Obliviously sort $\mathsf{Bin}^{\mathsf{in}}$ according to array index of the blocks placing real blocks before dummy blocks with the same array index.
3) Let k^{out}_i be the number of pairs $(b, \mathsf{Assign}(b)) \in \mathsf{Bin}^{\mathsf{out}}$ such that $\mathsf{Assign}(b) = i$ for all $i \in [L]$. We compute the values k^{out}_i privately in the following oblivious manner:

a) Initialize all $k^{\mathsf{out}}_1, \ldots, k^{\mathsf{out}}_L$ to 0 and store them encrypted on the server.

b) For each pair $(b, \mathsf{Assign}(b)) \in \mathsf{Bin}^{\mathsf{out}}$, scan all the ciphertexts of $k^{\mathsf{out}}_1, \ldots, k^{\mathsf{out}}_L$ and only increment $k^{\mathsf{out}}_{\mathsf{Assign}(b)}$.

4) Tag all items in $\mathsf{Bin}^{\mathsf{in}}$ with moving, if they are a real item that will be placed in D using the Assign mapping

from $\mathsf{Bin}^{\text{out}}$; 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 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 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.

We provide an efficiency analysis and proof of security for our multi-array shuffle scheme in the full version [30].

Theorem 4: Assuming the existence of a semantically secure encryption scheme and the existence of $\mathsf{cutoff} > 1$ 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 $i \in \{1, \ldots, \mathsf{cutoff} - 1\}$, Construction 3 is an Oblivious Random Multi-Array Shuffle with $O(N \log\log\lambda + \frac{N \log N}{\log\lambda})$ block communication.

IV. Oblivious Hash Table

In this section, we present our *oblivious hash table* (OHT) construction which achieves bandwidth overhead of $O(\log N + \log\log\lambda)$ blocks, amortized per query. Additionally, our OHT will allow efficient extraction of unqueried items in a random order unknown to the adversary. Our OHT 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 our overall efficiency requirements.

The OHT initialization algorithm adds as many dummy items as the number of input real items, mixes them together and outputs the resulted random permutation of dummy and real items in permuted order. We separate this initialization from the build for the OHT since in the context of our ORAM application, we will obtain the output of the initialization using the multi-array shuffle algorithm.

To build our OHT using the output of initialization of size R, we randomly split the input items into bins of expected polylog size using a PRF with output range $[\frac{R}{\log^{\mathsf{c}}\lambda}]$. For each bin j, a random new load R_j of real items is drawn from the binomial distribution on $(1-\epsilon)$ fraction of the real items. An oblivious sort is used to remove any excess items and pad the bin with dummies to size $\mathsf{thrsh} = (1-\epsilon)(\log^{\mathsf{c}}\lambda)$. This guarantees that the new load of real items R_j in each bin remains hidden from the server, which will be important for query security. The value of thrsh is chosen to guarantee that it is larger than any binomial load of real items R_j and that it is smaller than the total number of items assigned to each bin using the PRF. The resulting items, real and dummies, in each bin up to thrsh are used to build an oblivious bin.

After all oblivious bins have been built, only the excess items $\mathsf{D}_{\mathsf{over}}$ remain, which is a constant fraction ϵ of the original input items. We recursively assign these items into following smaller OHT levels. We make the size of the smallest level $O(\frac{N}{\log N})$ since for this size we can use a single single oblivious bin incurring only $O(N)$ cost.

To query for an item, we consider levels in reverse order that they were built in, from smallest to biggest. For each level, we either user the PRF evaluation on the searched item to determine one oblivious bin that will be queried, if the item has not been found. Once an item is found, we query random bins in the remaining levels.

To extract unqueried items from the OHT, it suffices to run the extraction algorithm on each oblivious bin and concatenate the output. The reason is that the unqueried items are distributed randomly among levels and among bins within each level using the PRFs and the oblivious binomial loads. Since each bin has logarithmic number of items we can use oblivious sort to implement the extraction for oblivious bins.

Next we present our formal OHT construction. We refer the reader to the full version of the paper [30] for our OblivBin constructions based on binary trees and Cuckoo hashing [16]. An OblivBin will be described by a hash table H and an additional stash S such that any item can be found with either an efficient lookup to H or scanning all of S.

Construction 5: [Oblivious Hash Table] Let F be a pseudorandom function and $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ be a symmetric key encryption. Let $\mathcal{U} = \mathcal{U}_{\mathsf{real}} \cup \mathcal{U}_{\mathsf{dummy}}$ where $\mathcal{U}_{\mathsf{real}}$ and $\mathcal{U}_{\mathsf{dummy}}$ are two non-intersecting sets representing the real and dummy key identifiers. We define an oblivious hash construction OblivHT = (OblivHT.Init, OblivHT.Build, OblivHT.Lookup, OblivHT.Extract) as follows:

OblivHT.Init.*:* This algorithm permutes the items of the input database and adds the necessary dummy items.

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

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

OblivHT.Build.*:* This algorithm builds an oblivious hash table from the output of OblivHT.Init.

$(\mathsf{st}, \tilde{H}, \tilde{S}) \leftarrow \mathsf{OblivHT.Build}\left(\tilde{D}, \mathsf{st}\right)$: the data $\tilde{D}$ contains at most half of it real items and the rest are dummy items.

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

OblivHT.BuildLevel.*:* This algorithm constructs one level in the OHT structure.

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

1) Let $R = R^{\mathsf{real}} + R^{\mathsf{dummy}}$.
2) If $R \leq \frac{N}{\log \lambda}$, set

$$(\mathsf{st}^{\mathsf{ctr}}, \tilde{H}^{\mathsf{ctr}}, \tilde{S}^{\mathsf{ctr}}) \leftarrow \mathsf{OblivBin.Build}(\tilde{D}).$$

Return $(\mathsf{st}^{\mathsf{ctr}}, \mathsf{ctr}, \mathsf{SK}), \tilde{H}^{\mathsf{ctr}}, \tilde{S}^{\mathsf{ctr}}$.

3) Otherwise, construct a level in the oblivious hash table as follows:
   a) Let F be a PRF with output range $[\frac{R}{\log^{\mathsf{c}} \lambda}]$. Generate a PRF key $\mathsf{K}_{\mathsf{ctr}}$.
   b) Initialize $\frac{R}{\log^{\mathsf{c}} \lambda}$ empty bins.
   c) For each $i \in [R]$ append $\mathsf{Enc}(\mathsf{SK}, (b_i, i, v_i))$ to bin $\mathsf{F}(\mathsf{K}_{\mathsf{ctr}}, i)$.
   d) Let $\mathsf{thrsh} = (1 - \epsilon)(\log^{\mathsf{c}} \lambda)$.
   e) Set $T = (1-\delta)R^{\mathsf{real}}$ and $t = \frac{R}{\log^{\mathsf{c}} \lambda}$. For $1 \leq j \leq \frac{R}{\log^{\mathsf{c}} \lambda}$ do as follows:
      i) Sample from the binomial distribution $R_j \leftarrow \mathsf{Binomial}(T, \frac{1}{t})$.[2]
      ii) If there are less than R_j real items B_j or the total number of items assigned to B_j were less than thrsh, abort.
      iii) Linearly scan the items assigned to B_j in Step 3c, leaving the tags $b_i = 0$ to the first R_j of the real items and setting the tag to rest to be $b_i = 2$; the dummy items stay with tag $b_i = 1$.
      iv) Obliviously shuffle 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 of the real items back to $b_i = 0$.
      v) Initialize a bin oblivious structure on the items left in $_j\mathsf{B}$:

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

      vi) Append the real items from $\tilde{S}^{(\mathsf{ctr},j)}$ to $\mathsf{D}_{\mathsf{over}}$.
      vii) Set $T = T - R_j$ and $t = t - 1$.
   f) Let $\tilde{H}^{(\mathsf{ctr})} \leftarrow \mathsf{Enc}\left(\mathsf{SK}, \{(\mathsf{st}^{(\mathsf{ctr},j)}, \tilde{H}^{(\mathsf{ctr},j)})\}_{j=1}^{R/\log^{\mathsf{c}} \lambda}\right)$.
   g) Call recursively the level building functionality on the remaining items in $\mathsf{D}_{\mathsf{over}}$: $(\mathsf{st}', \tilde{H}', \tilde{S}') \leftarrow \mathsf{OblivHT.BuildLevel}\left(\mathsf{D}_{\mathsf{over}}, \frac{|\mathsf{D}_{\mathsf{over}}|}{2}, \frac{|\mathsf{D}_{\mathsf{over}}|}{2}, \mathsf{ctr}+1, \mathsf{SK}\right)$.
   h) Return $(\mathsf{st}, \tilde{H}, \tilde{S})$, where $\mathsf{st} \leftarrow (\mathsf{SK}, \mathsf{ctr}, \mathsf{st}^{\mathsf{ctr}}, \mathsf{st}')$, $\tilde{H} \leftarrow \left(\tilde{H}', \tilde{H}^{(\mathsf{ctr})}\right)$, and $\tilde{S} \leftarrow \left(\tilde{S}'\right)$.

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) Let $\mathsf{st} = (\mathsf{st}^{(1)}, \mathsf{d}, \mathsf{SK})$ and $\tilde{H} = (\tilde{H}^{(1)}, \ldots, \tilde{H}^{(\mathsf{d})})$.
2) Set found $= 0$, for $\mathsf{ctr} = \mathsf{d}$ to 1, do the following:
   a) If $\mathsf{ctr} = \mathsf{d}$, then do

$$(v', \tilde{H}', \tilde{S}') \leftarrow \mathsf{OblivBin.Lookup}(k, \tilde{H}^{(\mathsf{ctr})}, \tilde{S}^{(\mathsf{ctr})}, \mathsf{st}^{(\mathsf{ctr})}),$$

   If $v' \neq \perp$, set $v \leftarrow v'$ and found $= 1$.
   b) If found $= 0$, set $j = \mathsf{F}(\mathsf{K}_{\mathsf{ctr}}, k)$, else choose j at random among the α_{ctr} bins at level ctr.
   c) Run

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

[2]This can be done in constant time using the Splitting algorithm for binomial random variates in Section 4.4., X4 [31].

If $v' \neq \perp$, set $v \leftarrow v'$ and 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 $\tilde{H} = (\tilde{H}^{(1)}, \dots, \tilde{H}^{(\mathsf{d})})$.
2) For $\mathsf{ctr} \in [\mathsf{d}]$ and $j \in [\alpha_{\mathsf{ctr}}]$ where $\alpha_{\mathsf{ctr}} = \frac{\epsilon^{i-1} N}{\log^c \lambda}$, let
$$(\tilde{D}_{\mathsf{ctr},j}, \mathsf{st}') \leftarrow \mathsf{OblivHT.Extract}(\tilde{H}^{(\mathsf{ctr},j)}, \tilde{S}, \mathsf{st}^{(\mathsf{ctr},j)}),$$
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 $\tilde{D}$.

We present the efficiency analysis and security proof for the OHT construction in the full version [30].

Theorem 6: Assuming the existence of a PRF, Construction 5 is an Oblivious Hash Table with block communication cost $O(N \log N)$ for Init, $O(N \log\log\lambda)$ for Build, $O(\log\log\lambda)$ for Lookup and $O(N \log\log\lambda)$ for Extract.

V. 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. We obtain the following result:

Theorem 7: Assuming the existence of a PRF, Construction 8 is an Oblivious RAM with $O(\log N \cdot \log\log N)$ amortized block communication cost per query.

Construction 8: Let OblivHT = (OblivHT.Init, OblivHT.Build, OblivHT.Lookup, OblivHT.Extract) be an oblivious hash table and 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 ORAM = (ORAM.Init, 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 $\mathsf{st}_{\mathsf{Build}}$.

$\mathsf{ORAM.Access}$.: 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 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 found $= 1$.
5) For $i = \alpha + 1$ to $\log N$, do the following:
   a) If found $= 1$, then sample a random query from $k \leftarrow \mathcal{U}_{\mathsf{query}}$ [3]. 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.

$\mathsf{ORAM.Shuffle}$.: 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) For $i = \alpha + 1$ to j. parse $\ell_i = (\mathsf{st}_i, \tilde{H}_i)$
$$(\tilde{A}_i, \mathsf{st}_i') \leftarrow \mathsf{OblivHT.Extract}(\tilde{H}_i, \mathsf{st}_i).$$
3) Run the multi-array shuffle algorithm on the data extracted from each level oblivious hash table
$$\tilde{D} \leftarrow \mathsf{OblMultArrShuff}\left(\{\tilde{A}_i\}_{i=\alpha}^{j}\right).$$
4) Construct an oblivious hash table using the output from the multi-array shuffle
$$(\mathsf{st}_j, \tilde{H}_j, \tilde{S}_j) \leftarrow \mathsf{OblivHT.Build}\left(\tilde{D}, \mathsf{st}\right).$$

[3]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.

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

The amortized bandwidth of a lookup in the above construction is $O(\log N \log\log\lambda)$ blocks. We present detailed security and efficiency analysis of the scheme in the full version [30].

References

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

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

[3] R. Ostrovsky and V. Shoup, "Private information storage (extended abstract)," in *STOC*, 1997.

[4] 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*, 2012, pp. 513–524.

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

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

[7] R. Ostrovsky, "Efficient computation on oblivious RAMs," in *STOC*, 1990, pp. 514–523.

[8] O. Goldreich and R. Ostrovsky, "Software Protection and Simulation on Oblivious RAMs," *J. ACM*, vol. 43, no. 3, 1996.

[9] E. Boyle and M. Naor, "Is There an Oblivious RAM Lower Bound?" in *ITCS*, 2016, pp. 357–368.

[10] 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.

[11] 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*, 2015, pp. 145–174.

[12] 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*, 2013, pp. 299–310.

[13] B. Pinkas and T. Reinman, "Oblivious RAM revisited," in *CRYPTO*, 2010, pp. 502–519.

[14] M. Ajtai, "Oblivious RAMs without cryptogrpahic assumptions," in *STOC*, 2010, pp. 181–190.

[15] I. Damgård, S. Meldgaard, and J. B. Nielsen, "Perfectly secure oblivious ram without random oracles," in *TCC*, 2011, pp. 144–163.

[16] M. T. Goodrich and M. Mitzenmacher, "Privacy-preserving access of outsourced data via oblivious ram simulation," in *ICALP*, 2011, pp. 576–587.

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

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

[19] 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*, 2013, pp. 1–18.

[20] K. Chung, Z. Liu, and R. Pass, "Statistically-secure ORAM with õ(log^2 n) overhead," in *ASIACRYPT 2014*, P. Sarkar and T. Iwata, Eds., 2014, pp. 62–81.

[21] 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*, 2015, pp. 415–430.

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

[23] T. Moataz, T. Mayberry, and E.-O. Blass, "Constant communication ORAM with small blocksize," in *CCS*, 2015, pp. 862–873.

[24] A. Waksman, "A permutation network," *Journal of the ACM (JACM)*, vol. 15, no. 1, pp. 159–163, 1968.

[25] M. T. Goodrich, "Randomized Shellsort: A simple oblivious sorting algorithm," in *SODA*, 2010, pp. 1262–1277.

[26] ——, "Zig-zag sort: A simple deterministic data-oblivious sorting algorithm running in O(n log n) time," in *STOC*, 2014, pp. 684–693.

[27] T. H. Chan, Y. Guo, W.-K. Lin, and E. Shi, "Cache-oblivious and data-oblivious sorting and applications," in *SODA*, 2018, pp. 2201–2220.

[28] S. Patel, G. Persiano, and K. Yeo, "CacheShuffle: A Family of Oblivious Shuffles," in *ICALP*, 2018, pp. 161:1–161:13.

[29] T.-H. H. Chan, K.-M. Chung, and E. Shi, "On the depth of oblivious parallel RAM," in *ASIACRYPT*, T. Takagi and T. Peyrin, Eds., 2017, pp. 567–597.

[30] 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.

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