CacheShuffle: A Family of Oblivious Shuffles

Sarvar Patel
Google LLC, Mountain View, USA
sarvar@google.com

Giuseppe Persiano
Google LLC, Mountain View, USA and Università di Salerno, Salerno, Italy
giuper@gmail.com

Kevin Yeo
Google LLC, Mountain View, USA
kwlyeo@google.com

Abstract

We consider *oblivious* two-party protocols where a *client* outsources N blocks of private data to a *server*. The client wishes to access the data to perform operations in such a way that the access pattern does not leak information about the data and the operations. In this context, we consider *oblivious shuffling* with a focus on bandwidth efficient protocols for clients with small local memory. In the shuffling problem, the N outsourced blocks, $B_1, \ldots, B_N$, are stored on the server according to an initial permutation π. The client wishes to reshuffle the blocks according to permutation σ. Oblivious shuffling is a building block in several applications that hide patterns of data access. In this paper, we introduce a generalization of the oblivious shuffling problem, the K-*oblivious shuffling* problem, and provide bandwidth efficient algorithms for a wide range of client storage requirements. The task of a K-oblivious shuffling algorithm is to shuffle N encrypted blocks that were previously randomly allocated on the server in such a way that an adversarial server learns nothing about either the new allocation of blocks or the block contents. The security guarantee must hold when an adversary has partial information on the initial placement of a subset of $K \leq N$ *revealed* blocks. The notion of oblivious shuffling is obtained for $K = N$.

We first study the N-oblivious shuffling problem and start by presenting CacheShuffleRoot, that is tailored for clients with $O(\sqrt{N})$ blocks of memory and uses approximately $4N$ blocks of bandwidth. CacheShuffleRoot is a 4x improvement over the previous best known N-oblivious shuffle for practical sizes of N. We then generalize CacheShuffleRoot to CacheShuffle that can be instantiated for any client memory size S and requires $O(N \log_S N)$ blocks of bandwidth. Next, we present K-oblivious shuffling algorithms that require $2N + f(K, S)$ blocks of bandwidth for all K and a wide range of S. Any extra bandwidth above the $2N$ lower bound depends solely on K and S. Specifically, for clients with $O(K)$ blocks of memory, we present KCacheShuffleBasic that uses exactly $2N$ blocks of bandwidth. For clients with memory $S \leq K$, we present KCacheShuffle, that requires $2N + O(K \log_S K)$ blocks of bandwidth. Finally, motivated by applications to ORAMs, we consider the case where the server stores D dummy blocks whose contents are irrelevant in addition to the N real blocks. For this case, we design algorithm KCacheShuffleDummy that shuffles $N + D$ blocks with K revealed blocks using $O(K)$ blocks of client storage and approximately $D + 2N$ blocks of bandwidth.

2012 ACM Subject Classification Security and privacy → Management and querying of encrypted data, Security and privacy → Privacy-preserving protocols, Information systems → Data encryption

Keywords and phrases Shuffling, Data-Oblivious Algorithms

Digital Object Identifier 10.4230/LIPIcs.ICALP.2018.161

© Sarvar Patel, Giuseppe Persiano, and Kevin Yeo;
licensed under Creative Commons License CC-BY

Related Version A full version of the paper can be found at [11], `https://arxiv.org/abs/1705.07069`.

1 Introduction

In recent years, cloud storage has increased in popularity due to the great benefits available to users. Outsourcing files to the cloud allows users to share documents conveniently. Users are able to access documents from many different machines without transferring data. The burden of data replication for recovery is placed on the storage provider. For many corporations, cloud storage is cost efficient compared to maintaining their own internal storage systems. With the widespread use of cloud storage, providing privacy for outsourced data becomes crucial. Unfortunately, encrypting outsourced data is not sufficient. Previous works [8, 9] show that learning the patterns of data access may leak information about the contents of encrypted data. This scenario provides motivation towards the study of *oblivious algorithms*.

In this paper, we focus on oblivious algorithms for *shuffling* data stored on a server. The ability to obliviously move blocks of encrypted data is an important primitive for privacy-conscious users of cloud storage. We consider the scenario where a client has outsourced the encryptions of N identically-sized blocks, $B_1, \ldots, B_N$, to a server. The N blocks are stored by the server on an array Source according to a permutation π. That is, an encryption of B_i is stored as $\mathsf{Source}[\pi(i)]$, for all $i = 1, \ldots, N$. The client wishes to shuffle the blocks into a server-stored destination array, Dest, according to a permutation σ.

An *oblivious shuffle* is an algorithm whose pattern of block movement and operations involving the server does not leak information about either σ or the contents of $B_1, \ldots, B_N$. If the client has N blocks of client memory available, oblivious shuffling is trivial. All N blocks are downloaded, decrypted, re-encrypted and re-uploaded to their correct location in Dest according to σ. Similarly, if bandwidth is unlimited, then oblivious shuffling is also trivial. The client streams all N blocks and keeps block $B_{\sigma^{-1}(1)}$ to be be placed into $\mathsf{Dest}[1]$ after all blocks have been streamed. The client repeats this algorithm for all $i = 2, \ldots, N$, which costs N^2 blocks of bandwidth. Our work focuses on oblivious shuffles that minimize bandwidth while the client has only a sublinear number of blocks of memory available.

Our Contributions. Our contribution is two-fold. We propose an obliviousness notion that abstracts the initial knowledge of π given to the adversarial server. Then, we present bandwidth efficient oblivious shuffles for any possible initial knowledge of π by the adversary and a wide range of client memory requirements.

Our work generalizes the notion of obliviousness studied in previous work by presenting the notion of K-*oblivious shuffling* that takes into account the knowledge of the adversary on the initial positioning of the blocks (that is, π). In the K-oblivious shuffling problem, the adversary fixes the position of $K \leq N$ blocks, which we denote as the *revealed* blocks. The client wishes to shuffle the N blocks into the server-stored destination array, Dest, without revealing information about σ or the contents of $B_1, \ldots, B_N$ to the adversarial server. The parameter K describes the difficulty of the problem. Intuitively, as K decreases, K-obliviously shuffling should be more efficient. In our work, we present algorithms whose bandwidth above inherent lower bounds depend only on K. For $K = N$, the notion of K-oblivious shuffling coincides with the original notion of a oblivious shuffling by Ohrimenko *et al.* [10], which assumes that the adversary has initial knowledge about all N input blocks.

The notion of K-oblivious shuffling is suitable in the context of Oblivious RAMs. Oblivious RAM (or ORAM) is a storage primitive introduced by Goldreich [4] that allows random access to N encrypted blocks that are stored by an adversarial server. Many ORAMs use oblivious shuffling as a building block to move blocks without revealing information about the final positions of blocks as well as the contents of the blocks. In the majority of ORAM constructions, the adversary does not learn the position of all blocks that are going to be shuffled. By incorporating the initial amount of knowledge available to the adversary about the N input blocks, we can improve the bandwidth efficiency of ORAM constructions such as the Square Root ORAM [4, 5].

Before designing efficient K-oblivious shuffling algorithms, we revisit the original oblivious shuffling (that is, N-oblivious shuffling) problem and present new oblivious shuffling algorithms with improved bandwidth efficiency. Our algorithms use a client-stored *cache* to store blocks. The main technical difficulty in our algorithms is showing that the size of the cache remains small. We first apply this design principle in Section 3 by presenting an oblivious shuffling algorithm, $\mathsf{CacheShuffleRoot}$, that uses approximately $4N$ blocks of bandwidth and $O(\sqrt{N})$ block of client storage. For similar client memory usage, the previous, state-of-the-art algorithm, the Melbourne Shuffle [10], uses about 4 times more bandwidth. We present a generalization of $\mathsf{CacheShuffleRoot}$, $\mathsf{CacheShuffle}_S$, in Section 4 when the client has $S = \omega(\log N)$ blocks of available client storage. $\mathsf{CacheShuffle}_S$ uses $O(N \log_S N)$ blocks of bandwidth.

Next, we focus on designing K-oblivious shuffling algorithms when $K < N$. All previous oblivious shuffling algorithms have always considered the most difficult scenario when $K = N$. To our knowledge, our work is the first to separate the two problems. In Section 5, we present a simple K-oblivious shuffling algorithm, $\mathsf{KCacheShuffleBasic}$, when the client has $O(K)$ blocks of available client storage. $\mathsf{KCacheShuffleBasic}$ uses exactly $2N$ blocks of bandwidth. In Section 6, we present $\mathsf{KCacheShuffle}_S$ for clients with only $O(S)$ blocks of client storage available. $\mathsf{KCacheShuffle}$ uses $2N + O(K \log_S K)$ blocks of bandwidth. For the case of $S = \sqrt{K}$, we present $\mathsf{KCacheShuffleRoot}$, which uses approximately $2N + 4K$ blocks of bandwidth. In general, any K-oblivious shuffling algorithm must upload and download each block at least once meaning a lower bound of $2N$ blocks of bandwidth when $K > 0$. The amount of bandwidth used by all our K-oblivious shuffling algorithms beyond the lower bound only depends on K.

In many ORAM schemes, encryptions of dummy blocks are also outsourced to the server. The contents of a dummy block is irrelevant. However, it is important that an adversarial server cannot learn whether a block is dummy or not. In the full version [11], we consider a scenario where the client has outsourced the encryptions of D *dummy* blocks and N *real* blocks inspired by the work of Stefanov *et al.* [12]. Any K-oblivious shuffling algorithm could be used to perform shuffling by treating dummy blocks as real blocks. By using the fact that the contents of dummy blocks are irrelevant, we present $\mathsf{KCacheShuffleDummy}$ that uses approximately $D + 2N$ blocks of bandwidth when the client has $O(K)$ blocks of available client storage. Applying directly $\mathsf{KCacheShuffleBasic}$ would require $2(N + D)$ blocks of bandwidth. Therefore, $\mathsf{KCacheShuffleDummy}$ saves D blocks of bandwidth. The bandwidth savings come at the cost of a small amount of server computation.

We complement our theoretical analysis with experiments to show that our algorithms are of practical interest in the full version.

Previous works. The early approach to oblivious shuffling was based on oblivious sorting algorithms which could be immediately derived from any sorting circuit. To evaluate a compare-exchange gate of a sorting circuit, the client downloads the two input encrypted blocks, decrypts and re-encrypts both blocks and uploads them in the correct order. Batcher's sort [2] is considered the most practical sorting circuit even though it has asymptotic cost of $O(N \log^2 N)$. Sorting networks such as AKS [1] and Zig-Zag [7] have $O(N \log N)$ size but large hidden constants. Randomized Shellsort [6] is another $O(N \log N)$ oblivious sort with smaller hidden constants but larger depth. Waksman [13] presents a circuit for oblivious shuffling of size $O(N \log N)$. Oblivious shuffling based on sorting circuits is interesting because the client only needs to store $O(1)$ blocks at any point. However, sorting circuits incur a large $\Omega(N \log N)$ blocks of bandwidth cost. The first oblivious shuffling algorithm not based on sorting circuits, the *Melbourne Shuffle*, was introduced by Ohrimenko *et al.* [10]. The Melbourne Shuffle uses $O(N)$ bandwidth while only requiring $O(\sqrt{N})$ blocks to be stored on the client at any time.

In the table below, we compare our algorithms with the Melbourne Shuffle [10].

Table 1 N denotes the number of blocks. Algorithm KCacheShuffleDummy receives D additional dummy blocks, for a total of $N+D$ blocks. Algorithm KCacheShuffleRoot is obtained from algorithm KCacheShuffle by setting $S = \sqrt{N}$. For all algorithms, server storage is cN, for a small constant c.

		Client Storage	Bandwidth
$K = N$	Melbourne Shuffle [10]	$O(\sqrt{N})$	$\approx 18N$
	CacheShuffleRoot	$O(\sqrt{N})$	$(4+\epsilon)N$
	CacheShuffle	$O(S)$	$O(N \log_S N)$
General K	KCacheShuffleBasic	$O(K)$	$2N$
	KCacheShuffleRoot	$O(\sqrt{K})$	$2N + (4+\epsilon)K$
	KCacheShuffle	$O(S)$	$2N + O(K \log_S K)$
	KCacheShuffleDummy	$O(K)$	$D + (2+\epsilon)N$

An algorithm similar to CacheShuffleRoot was developed in parallel and independent work in [3] for the context of privacy-preserving software monitoring.

2 Definitions

In this section, we give formal definition for *shuffling algorithms* and *oblivious shuffling algorithms*. Our reference scenario is a cloud storage model where a *client* outsources the storage of N identically-sized data blocks to a *server* with the capacity to store $M \geq N$ blocks.

We assume the N data blocks are uploaded by the Setup algorithm. As input, Setup receives N data blocks, $\mathbb{B} = (B_1, \ldots, B_N)$, each of size B and a *permutation* $\pi : [N] \to [N]$. Setup randomly selects an encryption key, `key`, whose length is determined by the security parameter λ, for a symmetric encryption scheme and uploads an encryption of each of the N data blocks to the server according to π. Formally, an encryption of B_i under `key` will be stored at server location $\pi(i)$, for all $i \in [N]$. Once Setup has uploaded all N blocks, an adversary $\mathcal{A}$ is allowed to learn the initial position of a subset of the data blocks, which we denote Revealed $\subseteq [N]$. For each index $i \in$ Revealed, $\pi(i)$ is revealed to $\mathcal{A}$. We call the data blocks in Revealed, the *revealed* data blocks.

The shuffling algorithm takes as input the encryption key `key`, the permutation map π, the set of revealed data blocks Revealed, and a new permutation σ. The task of the shuffling algorithm is to re-permute the N data blocks stored on the server according to the new permutation map σ. In particular, we are interested in *oblivious shuffling* algorithms. Roughly speaking, oblivious shuffles hide information about both the contents of $B_1, \ldots, B_N$

and σ even when the adversary has partial information on π (restricted to the input set $\mathsf{Revealed}$) and observes the blocks movements performed by the shuffling algorithm.

For convenience, we will abuse the notation of array indexing and function evaluation throughout our work. For any array A and index i, $A[i]$ refers to the element stored at location i in A. For a set of indices S, we define $A[S] := \{A[s] : s \in S\}$. Similarly for any function f and input set S, we define $f(S) := \{f(s) : s \in S\}$.

2.1 Mechanics of the Shuffling Algorithm

A shuffling algorithm receives as input the initial permutation π, the final permutation σ and the set $\mathsf{Revealed}$. A shuffling algorithm proceeds in steps. The state after the q-th step is described by a *server allocation map* $\rho_q : [M] \to [N] \cup \{\perp\}$ and by a *client allocation map* $L_q : [S] \to [N] \cup \{\perp\}$. Each allocation map specifies the block currently occupying each of the M server locations and S client locations, respectively. More precisely, $\rho_q(j) = i$ means that, after the q-th step is performed, the j-th server location contains an encryption of the B_i. If instead $\rho_q(j) = \perp$, then an encryption of a dummy block is stored at location j. Similar statements are true for the client allocation map, L_q.

When a shuffling algorithm starts, the server allocation map ρ_0 coincides with permutation map π on the first N storage location of the M server memory locations. That is, $\rho_0(i) = \pi^{-1}(i)$ for all $i = 1, \ldots, N$. The remaining $N - M$ locations contain encryptions of dummy blocks. All S client block locations initially contain dummy blocks. That is, $L_0(i) = \perp$ for all $i = 1, \ldots, S$. During each step, a shuffling algorithm can perform either a *move* operation or a *server computation* operation. A move operation can be either a *download* or an *upload* move and they modify the state as follows. All download and upload operations are associated with a *source* and a *destination*. Suppose the q-th operation is a download with source s_q and destination d_q. Then, an encryption of block $B_{\rho_{q-1}(s_q)}$ stored at server location s_q is copied to location d_q of client storage. Before storing in client storage, the block is decrypted and re-encrypted. As a consequence, the ρ_q is identical to ρ_{q-1}. On the other hand, L_q is identical to L_{q-1} except that $L_q(d_q) = \rho_{q-1}(s_q)$. If the q-th operation was an upload with source s_q and destination d_q, then the encryption of block $B_{\mathrm{L}_{q-1}(s_q)}$ stored at client location s_q is copied to location d_q of server storage. In this case, L_q is identical to L_{q-1}. However, S_q is obtained by modifying S_{q-1} such that $S_q(d_q) = L_{q-1}(s_q)$. Shuffle algorithms may perform upload moves with the source as $\perp$. In this case, an encryption of a dummy block is uploaded to the destination location of server storage.

A server computation operation is specified by the server performing a circuit that uses a subset of the blocks as input and copies the circuit's output to a subset of server storage locations. In our shuffling algorithms, server computation operations consist of homomorphic operations on block ciphertexts, which reduce bandwidth by using small amounts of server computation. The circuit description sent by the client must be considered as bandwidth.

2.2 Efficiency Measures

In our work, we consider three measures of efficiency for a shuffling algorithm: bandwidth, client memory and server memory. Our work focuses on minimizing bandwidth for a given amount of client memory, which is typically sublinear. While we do not prioritize optimizing server storage, all our shuffling algorithms use server storage that is linear with small constants in the number N of blocks. For shuffling algorithms with small client storage, we assume that the input permutations π and σ are space-efficient pseudorandom permutations.

Throughout this work, we consider blocks as our unit of measure. For example, if a shuffling algorithm uses T bandwidth, it means the shuffling algorithm uses T blocks of bandwidth.

2.3 Obliviousness

We define a *transcript* produced an execution of a shuffling algorithm Sh as the information seen by the adversarial server. The transcript consists of the initial encryptions of the data blocks as stored in server memory, the ordered list of the sources of all download moves, the ordered list of the destinations of all upload moves as well as the encryptions of the uploaded block and the list of circuits uploaded by the client. We stress that a transcript only contains the server locations that are involved in each move. That is, the transcript only contains the source for downloads and the destination for uploads. The transcript does not contain the involved client locations in each upload and download. The transcript models that an adversarial server $\mathcal{A}$ cannot observe information about the client's storage such as the destination of a download and the source of an upload.

Using the definition of a transcript, we now formally define an oblivious shuffling algorithm. For every sequence of N blocks $\mathbb{B} = (B_1, \ldots, B_N)$, every subset $\mathsf{Revealed}$ of revealed blocks, and every pair of permutations (π, σ), a shuffling algorithm Sh naturally induces a probability distribution $\mathcal{T}_{\mathsf{Sh}}(\mathbb{B}, \pi, \sigma, \mathsf{Revealed})$ over all possible transcripts. We capture the notion of a K*-oblivious shuffling* algorithm by the following game $\mathsf{OSGame}^{\mathcal{A}}_{\mathsf{Sh}}$ for a shuffling algorithm Sh between an adversary $\mathcal{A}$ and a challenger $\mathcal{C}$. In the formalization of our security notion, the adversary $\mathcal{A}$ receives partial information on the starting permutation map π to reflect the fact that the shuffling algorithm Sh might be part of a larger protocol whose execution leaks information about π. More precisely, $\mathcal{A}$ chooses the initial server locations of a subset of the N data blocks, $\mathsf{Revealed}$. We parametrize the security notion by the cardinality of the set $\mathsf{Revealed}$, which we denote as K. We say that an adversary $\mathcal{A}$ is K*-restricted* if it specifies the location of at most K blocks. That is, $|\mathsf{Revealed}| \leq K$. $\mathcal{C}$ fills in the remaining $N - |\mathsf{Revealed}|$ locations randomly under the constraint that each of the N blocks appears in exactly one location on the server. Then, $\mathcal{A}$ proposes two sequences of N blocks, $\mathbb{B}_0$ and $\mathbb{B}_1$, and two permutations, σ_0 and σ_1. $\mathcal{C}$ randomly picks $b \in \{0, 1\}$ and samples a transcript $\mathtt{trans}$ according to $\mathcal{T}_{\mathsf{Sh}}(\mathbb{B}_b, \pi, \sigma_b, \mathsf{Revealed})$. On input $\mathtt{trans}$, $\mathcal{A}$ outputs its guess b' for b. We present the formal definition below.

▶ **Definition 1.** For a shuffle algorithm Sh, we define the game $\mathsf{OSGame}^{\mathcal{A}}_{\mathsf{Sh}}(N, \lambda)$ between an adversary $\mathcal{A}$ and a challenger $\mathcal{C}$ for N data blocks and security parameter λ as follows:

1. $\mathcal{A}$ chooses a subset $\mathsf{Revealed} \subseteq [N]$, specifies $\pi(i)$ for each $i \in \mathsf{Revealed}$, and sends $(\mathsf{Revealed}, \pi(\mathsf{Revealed})$ to $\mathcal{C}$;
2. $\mathcal{A}$ chooses two pairs $(\mathbb{B}_0, \sigma_0)$ and $(\mathbb{B}_1, \sigma_1)$ and sends them to $\mathcal{C}$;
3. $\mathcal{C}$ completes the permutation π by randomly choosing the values at the point left unspecified by $\mathcal{A}$;
4. $\mathcal{C}$ randomly selects $b \leftarrow \{0, 1\}$ and sends $\mathcal{A}$ transcript $\mathtt{trans}$ drawn according to $\mathcal{T}_{\mathsf{Sh}}(\mathbb{B}_b, \pi, \sigma_b, \mathsf{Revealed})$;
5. $\mathcal{A}$ on input $\mathtt{trans}$ outputs b';

The game outputs 1 iff $b = b'$.

▶ **Definition 2** (K**-oblivious shuffling).** We say that shuffling algorithm Sh is a K*-oblivious shuffling* algorithm if for all K-restricted probabilistically polynomial time adversaries $\mathcal{A}$, and for all $N = \text{poly}(\lambda)$

$$\Pr[\mathsf{OSGame}^{\mathcal{A}}_{\mathsf{Sh}}(N, \lambda) = 1] \leq \frac{1}{2} + \text{negl}(\lambda).$$

We refer to N-oblivious shuffling algorithms as just *oblivious shuffling* algorithms.

2.4 Move-Based shuffling Algorithms

Move-based algorithms are shuffling algorithm that only perform move operations between client and server storage; that is, the server does not perform any computation on the stored encrypted blocks. To prove K-obliviousness for move-based algorithms, it suffices to show that for every random π and for every subset $\mathsf{Revealed} \subseteq [N]$ containing at most K indices, the probability distribution consisting of both the sources of downloads and the destinations of uploads are independent of σ conditioned on $\mathsf{Revealed}$ and $\pi(\mathsf{Revealed})$. More precisely, we define $\mathcal{M}_{\mathsf{Sh}}(\mathbb{B}, \pi, \sigma, \mathsf{Revealed})$ as the distribution of the move transcript $\mathtt{Mtrans}$ obtained from a transcript $\mathtt{trans}$ drawn from $\mathcal{T}_{\mathsf{Sh}}(\mathbb{B}, \pi, \sigma, \mathsf{Revealed})$ by removing the encryption of server-stored blocks and the encrypted blocks associated with upload moves. This is equivalent to considering the blocks as opaque indistinguishable balls. It is straightforward to prove that if the encryption scheme is IND-CPA and $\mathcal{T}_{\mathsf{Sh}}(\mathbb{B}, \pi, \sigma, \mathsf{Revealed})$ is independent of σ conditioned on $\mathsf{Revealed}$ and $\pi(\mathsf{Revealed})$, then Sh is a K-oblivious shuffling algorithm.

In the full version, we show that move-based K-oblivious shuffles have an inherent lower bound of $2N$ blocks of bandwidth when $K \geq 1$.

3 Oblivious Shuffling with $O(\sqrt{N})$ Client Memory

In this section, we describe $\mathsf{CacheShuffleRoot}$, which is an oblivious shuffling algorithm that uses $O(\sqrt{N})$ blocks of client storage except with negligible probability. For every $\epsilon > 0$, we describe an algorithm $\mathsf{CacheShuffleRoot}_\epsilon$ that uses $(3 + \epsilon/2)N$ blocks of server storage, $(4 + \epsilon)N$ blocks of bandwidth and $\delta_\epsilon\sqrt{N}$ blocks of client storage except with negligible probability in N. The value δ_ϵ is a constant that depends solely on ϵ and not from N. Whenever ϵ is clear from the context, we will just call the algorithm $\mathsf{CacheShuffleRoot}$.

3.1 Intuition

We start by describing a simple algorithm, which is insufficient to perform oblivious shuffling. However, the algorithm provides a general idea of our techniques.

Let us recall the inputs to oblivious shuffling. The client is provided permutations π and σ. Note that in our security model, σ is provided privately to the client and hidden from the adversarial server. On the server, N block ciphertexts are stored in the array Source. An encryption of block B_i is stored at $\mathsf{Source}[\pi(i)]$, for all $i = 1, \ldots, N$. At the termination of an oblivious shuffling, an encryption of block B_i should appear at $\mathsf{Dest}[\sigma(i)]$, for all $i = 1, \ldots, N$.

We now describe a simple, but incorrect, algorithm to provide intuition of our techniques. The N indices of Dest are randomly partitioned into $q := \sqrt{N}$ *destination buckets*, $\mathsf{destInd}_1, \ldots, \mathsf{destInd}_q$. Each $i \in [N]$ is assigned to a uniformly and independently chosen destination bucket. The indices of Source are partitioned into $s := \sqrt{N}$ *source groups*, $\mathsf{srcInd}_1, \ldots, \mathsf{srcInd}_s$, of exactly $N/s = \sqrt{N}$ blocks. The j-th source group consists of blocks located in $\mathsf{Source}[(j-1)N/s + 1, \ldots, j \cdot N/s]$, for all $j = 1, \ldots, s$. Finally, there will be q temporary server-stored arrays, $\mathsf{temp}_1, \ldots, \mathsf{temp}_q$, each of size s and initially empty.

On average, each destination bucket will contain $N/q = \sqrt{N}$ indices. Furthermore, each destination bucket will receive one block from each of the s source groups according to σ

in expectation. For now, let us suppose that each destination bucket receives exactly one block from each of the s source groups. In this case, we show that oblivious shuffling can be easily performed. Our algorithm would process each of the s source groups one at a time. When processing srcInd_j, the algorithm downloads all $\sqrt{N}$ blocks of $\mathsf{Source}[\mathsf{srcInd}_j]$. Then, exactly one block is uploaded to each of $\mathsf{temp}_1, \ldots, \mathsf{temp}_q$ from srcInd_j. In particular, the block to temp_k is the only block from srcInd_j that will be placed to $\mathsf{Dest}[\mathsf{destInd}_k]$ according to σ. After all s source groups have been processed, each temp_k contains all the blocks that are to be placed in a location of $\mathsf{Dest}[\mathsf{destInd}_k]$ but in the incorrect order. The algorithm performs another q phases to process each of $\mathsf{temp}_1, \ldots, \mathsf{temp}_q$. When processing temp_k, all s blocks of temp_k are downloaded and re-uploaded to their correct locations in $\mathsf{Dest}[\mathsf{destInd}_k]$ in any arbitrary order (such as as increasing order of $\mathsf{destInd}_k$).

This algorithm is easily seen to be oblivious but, unfortunately, it is unlikely that each destination bucket receives exactly the expected number blocks from each group. We thus present algorithm CacheShuffleRoot, which modifies the above algorithm to handle variances in expectation. CacheShuffleRoot does not expect each source group to contain exactly one block that should be uploaded to each destination bucket. Any time more than one block from a source group should be uploaded to a destination bucket, the extra blocks will be stored in a cache in the client's private storage. To ensure that the client's cache does not grow too large, we slightly increase the number of destination buckets from $\sqrt{N}$ to $(1 + \epsilon/2)\sqrt{N}$, for any constant $\epsilon > 0$. Let us now proceed more formally.

3.2 CacheShuffleRoot Description

For any constant $\epsilon > 0$, we describe algorithm CacheShuffleRoot. As input, the client receives permutations π and σ. On the server, the N blocks are stored in the *source array* Source according to π. That is, a ciphertext of block B_i appears in $\mathsf{Source}[\pi(i)]$, for all $i = 1, \ldots, N$. CacheShuffleRoot will output a server-stored *destination array*, Dest, such that an encryption of block B_i is stored as $\mathsf{Dest}[\sigma(l)]$, for all $l = i, \ldots, N$.

CacheShuffleRoot proceeds by partitioning the indices of Source into $s := \sqrt{N}$ *source groups* $\mathsf{srcInd}_1, \ldots, \mathsf{srcInd}_s$. For all $j = 1, \ldots, s$, srcInd_j consists of blocks stored at locations $\mathsf{Source}[(j-1)s + 1, \ldots, js]$. The N indices of the destination array Dest are randomly partitioned into $q := (1 + \epsilon/2)\sqrt{N}$ *destination buckets*, $\mathsf{destInd}_1, \ldots, \mathsf{destInd}_q$. Formally, index $i \in [N]$ is assigned to uniformly and independently chosen destination bucket. CacheShuffleRoot also initializes q server-stored *temporary arrays*, $\mathsf{temp}_1, \ldots, \mathsf{temp}_q$ and q client-stored *caches*, $\mathsf{Q}_1, \ldots, Q_q$. Each temporary array initially contains s empty block locations and each cache is initially empty.

CacheShuffleRoot consists of two phases: Spray and Recalibrate. The Spray phase consists of s rounds, one for each of the s source groups. In the j-th Spray round, the algorithm downloads all blocks in $\mathsf{Source}[\mathsf{srcInd}_j]$. Each downloaded ciphertext is decrypted and re-encrypted with fresh randomness. Each downloaded block is placed into one of the q caches according to their placement by σ. If block B_i is downloaded, then the re-encryption of B_i is placed into Q_k such that $\sigma(i) \in \mathsf{destInd}_k$. After all blocks of $\mathsf{Source}[\mathsf{srcInd}_j]$ are placed into their respective queues, exactly one block from each of $\mathsf{Q}_1, \ldots, Q_q$ is uploaded to the server. In particular, one block from Q_k is uploaded to temp_k. If any Q_k is empty, a *dummy* block containing an encryption of any arbitrary block is uploaded instead. After all s rounds of the Spray phase are completed, each of $\mathsf{temp}_1, \ldots, \mathsf{temp}_q$ contains exactly s blocks. Furthermore, there might be some blocks remaining in $\mathsf{Q}_1, \ldots, \mathsf{Q}_q$. Every block that is assigned to a location of $\mathsf{Dest}[\mathsf{destInd}_k]$ according to σ appears in either Q_k or temp_k.

The Recalibrate phase will simply rearrange all non-dummy blocks of Q_k and temp_k into the correct locations of $\mathsf{Dest}[\mathsf{destInd}_k]$. Formally, Recalibrate operates in q rounds, one round for each pair of $(\mathsf{Q}_1, \mathsf{temp}_1), \ldots, (\mathsf{Q}_q, \mathsf{temp}_q)$. In the k-th round, all s blocks of temp_k are downloaded. All non-dummy blocks of temp_k are decrypted and re-encrypted before being placed into Q_k. At this point, all blocks assigned to $\mathsf{Dest}[\mathsf{destInd}_k]$ according to σ appear in Q_k. All blocks are simply uploaded to $\mathsf{Dest}[\mathsf{destInd}_k]$ in any arbitrary order (such as increasing order of $\mathsf{destInd}_k$). After all q rounds of Recalibrate, CacheShuffleRoot finishes executing.

The proof of the following theorem can be found in the full version.

▶ **Theorem 3.** CacheShuffleRoot *is an oblivious shuffle algorithm that uses* $(4+\epsilon)N$ *blocks of bandwidth,* $(3+\epsilon/2)N$ *blocks of server memory and* $O(\sqrt{N})$ *blocks of client memory except with probability negligible in* N*.*

4 Oblivious Shuffling with Smaller Client Memory

In this section, we generalize algorithm CacheShuffleRoot from Section 3. For any $S = \omega(\log N)$, we provide an oblivious shuffling algorithm that uses $O(S)$ blocks of client memory and $O(N \log_S N)$ blocks of bandwidth.

Let us take another look at CacheShuffleRoot. Once CacheShuffleRoot completes the Spray phase, all the blocks that should be placed into $\mathsf{Dest}[\mathsf{destInd}_k]$ according to σ are stored either in temp_k or Q_k. Afterwards the k-th round of Recalibrate arranges all non-dummy blocks in temp_k and Q_k into their correct location in $\mathsf{Dest}[\mathsf{destInd}_k]$. Each round of Recalibrate requires $|\mathsf{temp}_k| + |\mathsf{Q}_k|$ blocks of client memory. Therefore, the key to a oblivious shuffling using less blocks of client memory requires a modification to the Spray phase so that less blocks are placed into each temp_k and Q_k. We present RSpray, a modification of Spray, to achieve smaller server-stored temporary arrays and client-stored caches. Additionally, the output of RSpray is structured such that RSpray may be recursively applied.

4.1 RSpray Description

First, we abstract the input to RSpray to handle recursive applications. As input, RSpray receives a server-stored *source array*, RSource, of n block ciphertexts. In addition, the client privately receives, $\mathsf{destInd} \subseteq [N]$, of d *destination indices.* For every i such that $\sigma(i) \in \mathsf{destInd}$, an encryption of block B_i appears in RSource. The remaining $n - d$ ciphertexts of RSource are dummy blocks.

RSpray is parameterized by the number of blocks of client storage available, which we denote by S. As output, RSpray outputs S *temporary arrays*, $\mathsf{temp}_1, \ldots, \mathsf{temp}_S$, which contain block ciphertexts as well as a partition of destInd into S *destination buckets*, $\mathsf{destInd}_1, \ldots, \mathsf{destInd}_S \subset \mathsf{destInd}$. Furthermore, RSpray guarantees that if $\sigma(i) \in \mathsf{destInd}_k$, then an encryption of B_i will appear in temp_k. To keep the same notation as Section 3, we set $q = S$.

We now formally describe RSpray. RSpray partitions destInd into q destination buckets, $\mathsf{destInd}_1, \ldots, \mathsf{destInd}_q$. Each index $d \in \mathsf{destInd}$ is assigned to one of the q subsets uniformly and independently at random. RSpray will initialize q server-stored temporary arrays, $\mathsf{temp}_1, \ldots, \mathsf{temp}_q$. Each temp_k will contain $s := n/q$ empty block locations. RSpray also initializes q client-stored caches, $\mathsf{Q}_1, \ldots, \mathsf{Q}_q$. All q caches are initially empty. Finally, RSpray partitions RSource into s server-stored *source buckets*, $\mathsf{srcInd}_1, \ldots, \mathsf{srcInd}_s$. Each block ciphertext in RSource is assigned to one of the s source buckets uniformly and independently at random. Unlike Spray, RSpray initializes the s source buckets randomly.

RSpray performs s rounds, one for each of $\mathsf{srcInd}_1, \ldots, \mathsf{srcInd}_s$. In the j-th round, RSpray downloads all blocks of srcInd_j. All non-dummy blocks are decrypted and re-encrypted.

If block B_i is downloaded, then the new encryption of B_i is placed into Q_k such that $\sigma(i) \in \mathsf{destInd}_k$. After all blocks in srcInd_j are placed into their corresponding cache, exactly one block ciphertext is uploaded from each $\mathsf{Q}_1, \ldots, \mathsf{Q}_q$ to the server. In particular, a block from Q_k is uploaded to temp_k, for all $k = 1, \ldots, q$. If Q_k is empty, a dummy block is uploaded instead.

After all s rounds complete, each temp_k contains exactly s block ciphertexts. Furthermore, RSpray guarantees that if $\sigma(i) \in \mathsf{destInd}_k$, then an encryption of B_i appears in either temp_k or Q_k. RSpray performs another q *adjustment rounds* to move all client-stored blocks in Q_k to the server-stored temp_k. In the k-th adjustment round, RSpray downloads all s blocks of temp_k. All dummy blocks of temp_k are discarded. The non-dummy blocks of temp_k are decrypted and re-encrypted. Now, all blocks of Q_k are combined with the non-dummy downloaded blocks of temp_k. If more than s non-dummy blocks remain, then RSpray fails. If there are less than s non-dummy blocks, dummy blocks are added until exactly s blocks remain. Finally, all s blocks are uploaded back to temp_k. RSpray terminates upon completion of the s rounds.

4.2 CacheShuffle Description

We now describe CacheShuffle, which uses RSpray and Spray as subroutines. CacheShuffle starts by running the Spray algorithm of CacheShuffleRoot with parameters $s := N/S$ and $q := (1+\epsilon)S$. Under these parameters, Spray uses $O(S)$ client memory and outputs the following:

1. q caches $\mathsf{Q}_1, \ldots, \mathsf{Q}_q$ on the client;
2. q temporary arrays $\mathsf{temp}_1, \ldots, \mathsf{temp}_q$ on the server;
3. q destination buckets $\mathsf{destInd}_1, \ldots, \mathsf{destInd}_q$ on the client such that if $\sigma(i) \in \mathsf{destInd}_k$ then Q_k or temp_k contain an encryption of B_i;

For $j = 1, \ldots, q$, we perform the *adjustment round* of RSpray for each pair of $(\mathsf{Q}_k, \mathsf{temp}_k)$ outputted by Spray. In particular, all blocks of Q_k are placed into temp_k. After all q adjustment rounds, we have the property that if $\sigma(i) \in \mathsf{destInd}_k$, then temp_k contains an encryption of B_i.

Next, CacheShuffle recursively calls RSpray on each $\mathsf{temp}_1, \ldots, \mathsf{temp}_q$ exactly $l = O(\log_S N)$ times. Formally, each application of RSpray will produce q pairs of temporary arrays and destination buckets. RSpray will be applied to each of the q pairs independently. After l recursive applications of RSpray, we will receive q^l temporary arrays and destination buckets, $(\mathsf{temp}_{l,1}, \mathsf{destInd}_{l,1})$,…,$(\mathsf{temp}_{l,q^l}, \mathsf{destInd}_{l,q^l})$. Furthermore, all temporary arrays and destination buckets will contain less than S^2 elements. Each $\mathsf{temp}_{l,k}$ is obliviously shuffled into $\mathsf{Dest}[\mathsf{destInd}_{l,k}]$ using CacheShuffleRoot, for all $k = 1, \ldots, q^l$. CacheShuffleRoot may be applied since the client has at least S blocks of client memory.

The proof of the following theorem can be found in the full version.

▶ **Theorem 4.** CacheShuffle *is an Oblivious Shuffle algorithm that uses $O(N \log_S N)$ blocks of bandwidth, $O(N)$ blocks of server memory and $O(S)$ blocks of client memory except with probability negligible in N.*

5 K-Oblivious Shuffling with $O(K)$ Client Memory

In this section, we present an K-oblivious shuffling algorithm, KCacheShuffleBasic, for clients with at least K blocks of client memory. The algorithm uses $2N$ blocks of bandwidth to shuffle N data blocks We remind the reader that the client of a K-oblivious shuffling receives two permutations (π, σ) as well as a subset of indices $\mathsf{Revealed} \subseteq [N]$. Identical to Oblivious Shuffling, $\mathsf{KCacheShuffleBasic}$ also receives ciphertexts of the N blocks in Source according to π and must place the N blocks in Dest according to σ.

$\mathsf{KCacheShuffleBasic}$ starts by downloading the ciphertexts from locations $\mathsf{Source}[\pi(\mathsf{Revealed})]$. That is, the encryption of every block whose index belongs to $\mathsf{Revealed}$ is downloaded. Each block is decrypted and re-encrypted. Then, $\mathsf{KCacheShuffleBasic}$ initializes $\mathsf{tbDown} = [N] \setminus \mathsf{Revealed}$, which is the set of indices of blocks that have not been downloaded yet.

Then, $\mathsf{KCacheShuffleBasic}$ runs N rounds. The goal of the i-th round is to place a ciphertext of $B_{\sigma^{-1}(i)}$ into $\mathsf{Dest}[i]$. If $\sigma^{-1}(i) \in \mathsf{tbDown}$, and thus the ciphertext of $B_{\sigma^{-1}(i)}$ has not been downloaded yet, $\mathsf{KCacheShuffleBasic}$ simply downloads $B_{\sigma^{-1}(i)}$ from $\mathsf{Source}[\pi(\sigma^{-1}(i))]$ and removes $\sigma^{-1}(i)$ from tbDown. If $\sigma^{-1}(i) \notin \mathsf{tbDown}$, then $B_{\sigma^{-1}(i)}$ has already been downloaded from $\mathsf{Source}[\pi(\sigma^{-1}(i))]$. If $\mathsf{tbDown} \neq \emptyset$, any arbitrary index $i' \in \mathsf{tbDown}$ is removed and $B_{i'}$ is downloaded from $\mathsf{Source}[\pi(i')]$. In any of the above cases, the downloaded block is decrypted and re-encrypted. Finally, $B_{\sigma^{-1}(i)}$ is placed into $\mathsf{Dest}[i]$.

From first look, it seems that $\mathsf{KCacheShuffleBasic}$ requires $O(N)$ roundtrips of client-server communication. However, the number of roundtrips may be reduced by grouping indexes of Dest together. Specifically, we can group indexes of Dest into groups of size $O(K)$ and perform the required downloads and uploads in $O(N/K)$ roundtrips.

The proof of the following theorem can be found in the full version.

▶ **Theorem 5.** $\mathsf{KCacheShuffleBasic}$ *is a K-Oblivious Shuffle that uses $2N$ blocks of bandwidth, $2N$ blocks of server storage and $O(K)$ blocks of client storage.*

6 K-Oblivious Shuffling with Smaller Client Memory

In this section, for any constant $\epsilon > 0$, we describe $\mathsf{KCacheShuffle}_{\epsilon,S}$, a K-oblivious shuffling that uses $O(S)$ blocks of client memory. For convenience, we fix ϵ and S and refer to $\mathsf{KCacheShuffle}_{\epsilon,S}$ as simply $\mathsf{KCacheShuffle}$.

$\mathsf{KCacheShuffle}$ will invoke a modification of $\mathsf{CacheShuffle}$ (using the same value of ϵ). Recall that $\mathsf{CacheShuffle}$ invokes $\mathsf{CacheShuffleRoot}$ before completion. $\mathsf{KCacheShuffle}$ invokes $\mathsf{CacheShuffle}$ such that the last $\mathsf{Recalibrate}$ phase of $\mathsf{CacheShuffleRoot}$ is skipped. Note, only the last $\mathsf{Recalibrate}$ phase of $\mathsf{CacheShuffleRoot}$ actually places ciphertexts of blocks into the destination array Dest. Therefore, the modified $\mathsf{CacheShuffle}$ does not actually use Dest at all. Formally, $\mathsf{CacheShuffle}$ invokes $\mathsf{KCacheShuffle}$ using $\mathsf{Source}[\mathsf{Revealed}]$ as the source array and (π, σ) restricted to the input set $\mathsf{Revealed}$ as the input permutations. The output of the modified $\mathsf{CacheShuffle}$ is:

1. q client-stored *destination buckets* $\mathsf{destInd}_1, \ldots, \mathsf{destInd}_q$, which is a partition of $\mathsf{Revealed}$;
2. q server-stored *temporary arrays* $\mathsf{temp}_1, \ldots, \mathsf{temp}_q$ containing exactly S block ciphertexts such that if $i \in \mathsf{Revealed}$ and $\sigma(i) \in \mathsf{destInd}_k$ then temp_k contains an encryption of B_i;

In the next step, $\mathsf{KCacheShuffle}$ merges the revealed and unrevealed blocks. $\mathsf{KCacheShuffle}$ will extend $\mathsf{destInd}_1, \ldots, \mathsf{destInd}_q$ from a partition of $\mathsf{Revealed}$ to a partition of $[N]$. That is, each index of $[N] \setminus \sigma(\mathsf{Revealed})$ is assigned uniformly and independent at random to one of $\mathsf{destInd}_1, \ldots, \mathsf{destInd}_q$. $\mathsf{KCacheShuffle}$ initializes $\mathsf{tbDown}_k = \sigma^{-1}(\mathsf{destInd}_k \setminus \sigma(\mathsf{Revealed}))$, which is all unrevealed blocks assigned to $\mathsf{Dest}[\mathsf{destInd}_k]$ by σ, for all $k = 1, \ldots, q$. Then, $\mathsf{KCacheShuffle}$ will run up to q rounds, one for each of $\mathsf{destInd}_1, \ldots, \mathsf{destInd}_q$. In the k-th round, $\mathsf{KCacheShuffle}$ downloads all S ciphertexts of temp_k.

We quickly diverge by mentioning an obvious, but incorrect, next step. In particular, it might be tempting to just download all of $\mathsf{Source}[\pi(\mathsf{tbDown}_k)]$ to upload into $\mathsf{Dest}[\mathsf{destInd}_k]$.

However, this step would reveal to the adversary the number of revealed (as well as unrevealed blocks) that are placed into $\mathsf{Dest}[\mathsf{destInd}_k]$ according to σ. This possible next step would cause our algorithm to lose K-obliviousness. In particular, our algorithm cannot leak knowledge about the number of revealed (and unrevealed) blocks that belong in the set $\mathsf{Dest}[\mathsf{destInd}_k]$.

Instead, KCacheShuffle downloads exactly $u_k := |\mathsf{destInd}_k| - (1-\epsilon)K/q$ unrevealed blocks in the k-th round. If $|\mathsf{tbDown}_k| > u_k$, then KCacheShuffle will fail and abort (in Theorem 6 we shall prove that this happens with negligible probability). On the other hand, suppose that $|\mathsf{tbDown}_k| \leq u_k$. In this case, the algorithm will download all blocks in $\mathsf{Source}[\pi(\mathsf{tbDown}_k)]$ and if more are needed, that is if $|\mathsf{tbDown}_k| < u_k$, then extra unrevealed blocks that are not assigned to $\mathsf{Source}[\mathsf{destInd}_k]$ are downloaded from the destination array with the largest index that has not been used yet. Formally, KCacheShuffle initializes $\mathsf{leftover} = q$ before any of the q rounds begins. The $|\mathsf{tbDown}_k| - u_k$ extra blocks are chosen arbitrarily from the set $\mathsf{Source}[\pi(\mathsf{tbDown}_{\mathsf{leftover}})]$, which have not been downloaded yet. If all blocks of $\mathsf{Source}[\pi(\mathsf{tbDown}_{\mathsf{leftover}})]$ have been been downloaded, then leftover is decremented.

All u_k blocks downloaded by KCacheShuffle in the k-th round will be decrypted and re-encrypted. Then, KCacheShuffle will upload all blocks belonging to $\mathsf{Dest}[\mathsf{destInd}_k]$ in any arbitrary order such as increasing in $\mathsf{destInd}_k$. Furthermore, KCacheShuffle has a set of extra unrevealed blocks, which we denote Rem_k, that are not assigned to $\mathsf{Dest}[\mathsf{destInd}_k]$ by σ. If $|\mathsf{Rem}_k| > 2\epsilon K/q$, then KCacheShuffle fails and aborts Otherwise, KCacheShuffle pads Rem_k to contain exactly $2\epsilon K/q$ ciphertexts by adding encryptions of dummy blocks. Afterwards, Rem_k is uploaded to the server.

At some point, leftover and k will be equal. Once leftover and k are the same, KCacheShuffle will stop running these rounds. However, KCacheShuffle still has to place blocks into $\mathsf{Dest}[\mathsf{destInd}_{\mathsf{leftover}}], \dots, \mathsf{Dest}[\mathsf{destInd}_q]$ from $\mathsf{temp}_{\mathsf{leftover}}, \dots, \mathsf{temp}_q$ and $\mathsf{Rem}_1, \dots, \mathsf{Rem}_{\mathsf{leftover}-1}$. To achieve this, KCacheShuffle invokes CacheShuffle using $\mathsf{temp}_{\mathsf{leftover}} \cup \dots \cup \mathsf{temp}_q \cup \mathsf{Rem}_1 \cup \dots \mathsf{Rem}_{\mathsf{leftover}-1}$ as the source array, $\mathsf{Dest}[\mathsf{destInd}_{\mathsf{leftover}} \cup \dots \cup \mathsf{destInd}_q]$ as the destination array and (π, σ) restricted to $\sigma^{-1}(\mathsf{destInd}_{\mathsf{leftover}}) \cup \dots \cup \sigma^{-1}(\mathsf{destInd}_q)$ as the permutations. In the full version, we show that $\mathsf{destInd}_{\mathsf{leftover}} \cup \dots \cup \mathsf{destInd}_q$ contains $O(K)$ indices.

If the client has $O(\sqrt{K})$ blocks of client storage, then we may replace CacheShuffle with CacheShuffleRoot above. We refer to this construction as KCacheShuffleRoot.

The proof of the following theorem can be found in the full version.

▶ **Theorem 6.** *For every $S = \omega(\log N)$ and for every ϵ, $\mathsf{KCacheShuffle}_{\epsilon,S}$ is a K-oblivious shuffling algorithm that uses $2N + O(K \log_S K)$ blocks of bandwidth, $O(N)$ blocks of server storage, $O(S)$ blocks of client storage and fails with probability negligible in N.*

7 Conclusions

In this paper, we studied the notion of *oblivious* algorithms for the problem of shuffling data. We introduce the notion of K-oblivious shuffling, a fine-grained notion of obliviousness, which accurately describes the adversary's knowledge about the initial position of the N blocks. In particular, we assume the adversary gains information about the initial position of exactly K blocks. This notion has direct application to the design of Oblivious RAMs. Previous notions only considered the extreme case where the adversary has complete knowledge about the initial positioning of all N input blocks (that is, $K = N$).

We present bandwidth efficient moved-based K-oblivious shuffling algorithms for any K and for any client with $S = \omega(\log N)$ blocks of available storage. The bandwidth required by our algorithms is of the form $2N + f(K, S)$. We also look at the previous considered the case of $K = N$ corresponding to previous oblivious shuffling notions. We present an oblivious shuffle using essentially $4N$ blocks of bandwidth. The previous, state-of-the-art oblivious shuffle [10] required approximately $18N$ blocks of bandwidth for similar failure probabilities.

We also consider the case where we shuffle N *real* blocks along with D *dummy* blocks. In this case, the contents of dummy blocks are irrelevant. By utilizing server-side computation, we shuffle using essentially $D + (2 + \epsilon)N$ blocks of bandwidth. Thus, our algorithm uses less bandwidth than any move-based algorithm uses at least $2(N + D)$ blocks of bandwidth.

References

1 M. Ajtai, J. Komlós, and E. Szemerédi. An $O(n \log n)$ sorting network. In *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing*, STOC '83, pages 1–9. ACM, 1983.

2 K. E. Batcher. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*, AFIPS '68 (Spring), pages 307–314. ACM, 1968.

3 Andrea Bittau, Úlfar Erlingsson, Petros Maniatis, Ilya Mironov, Ananth Raghunathan, David Lie, Mitch Rudominer, Ushasree Kode, Julien Tinnés, and Bernhard Seefeld. Prochlo: Strong privacy for analytics in the crowd. In *SOSP*, pages 441–459. ACM, 2017.

4 O. Goldreich. Towards a theory of software protection and simulation by oblivious RAMs. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, STOC '87, pages 182–194, New York, NY, USA, 1987. ACM.

5 Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *J. ACM*, 43(3):431–473, 1996.

6 Michael T. Goodrich. Randomized Shellsort: A simple data-oblivious sorting algorithm. *J. ACM*, 58(6):27:1–27:26, 2011. `doi:10.1145/2049697.2049701`.

7 Michael T. Goodrich. Zig-Zag sort: A simple deterministic data-oblivious sorting algorithm running in $O(n \log n)$ time. In *Proceedings of the Forty-sixth Annual ACM Symposium on Theory of Computing*, STOC '14, pages 684–693, New York, NY, USA, 2014. ACM. `doi:10.1145/2591796.2591830`.

8 Chang Liu, Liehuang Zhu, Mingzhong Wang, and Yu-An Tan. Search pattern leakage in searchable encryption: Attacks and new construction. *Inf. Sci.*, 265:176–188, 2014.

9 Muhammad Naveed, Seny Kamara, and Charles V. Wright. Inference attacks on property-preserving encrypted databases. In *CCS '15*, pages 644–655. ACM, 2015.

10 Olga Ohrimenko, Michael T. Goodrich, Roberto Tamassia, and Eli Upfal. The Melbourne shuffle: Improving oblivious storage in the cloud. In *Automata, Languages, and Programming: 41st International Colloquium, ICALP 2014, Copenhagen, Denmark, July 8-11, 2014, Proceedings, Part II*, pages 556–567, 2014.

11 Sarvar Patel, Giuseppe Persiano, and Kevin Yeo. CacheShuffle: A family of oblivious shuffles. *CoRR*, abs/1705.07069, 2017. `arXiv:1705.07069`.

12 Emil Stefanov, Elaine Shi, and Dawn Song. Towards practical oblivious ram. *arXiv preprint arXiv:1106.3652*, 2011.

13 Abraham Waksman. A permutation network. *J. ACM*, 15(1):159–163, 1968. `doi:10.1145/321439.321449`.