Repeatable Oblivious Shuffling of Large Outsourced Data Blocks

Zhilin Zhang*, Ke Wang*, Weipeng Lin*, Ada Wai-Chee Fu† and Raymond Chi-Wing Wong‡
* School of Computing Science, Simon Fraser University
Email: {zhilinz,wangk,weipengl}@sfu.ca
† Department of Computer Science and Engineering, The Chinese University of Hong Kong
Email: adafu@cse.cuhk.edu.hk
‡ Department of Computer Science and Engineering, Hong Kong University of Science and Technology
Email: raywong@cse.ust.hk

***Abstract*—As data outsourcing becomes popular, oblivious algorithms have raised extensive attentions since their control flow and data access pattern appear to be independent of the input data they compute on and thus are especially suitable for secure processing in outsourced environments. In this work, we focus on oblivious shuffling algorithms that aim to shuffle encrypted data blocks outsourced to the cloud server without disclosing the permutation of blocks to the server. Existing oblivious shuffling algorithms suffer from issues of heavy communication and client computation costs when blocks have a large size because all outsourced blocks must be downloaded to the client for shuffling or peeling off extra encryption layers. To help eliminate this void, we introduce the "repeatable oblivious shuffling" notation that restricts the communication and client computation costs to be independent of the block size. We present an efficient construction of repeatable oblivious shuffling using additively homomorphic encryption schemes. The comprehensive evaluation of our construction shows its effective usability in practice for shuffling large-sized blocks.**

I. Introduction

In recent years, data outsourcing has increased in popularity due to the great benefits available to users. It allows a third party cloud server to take over complicated and expensive tasks of storing, managing, and utilizing data for individual users called the client. Cloud servers are widely considered as "semi-trusted" or "honest-but-curious", in that they follow the protocol honestly but may passively attempt to learn protected information from all data observed during the execution of the protocol. For this reason, outsourced data are crucially encrypted by the client. However, encrypting outsourced data is not sufficient for providing privacy. For example, previous works [14], [16], [22], [13] show that disclosing 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 study oblivious algorithms for *shuffling* encrypted data on a server. We consider the scenario in which the client has outsourced the encryption of n identically-sized data blocks. At some later time, the client wishes to obliviously shuffle these encrypted blocks once in a while, according to some permutation. An *oblivious shuffle* is an algorithm whose patterns of block movements and computational operations do not leak any information about the actual permutation to the server. The ability to obliviously permute blocks of encrypted data is an important primitive for privacy-aware outsourcing services in cloud computing. We highlight some typical applications here.

- **Privacy-preserving data access**: The sequence of accesses to outsourced data (i.e., access pattern) can disclose sensitive user information, such as access privilege, access frequency, and visiting habits, etc. To hide access patterns, Oblivious RAM (ORAM) [9] and other lightweight solutions [8], [18] commonly depend on oblivious shuffling to continually move outsourced data around in the server's storage in a fashion that disallows the server to correlate the previous physical locations of the data with their new locations.
- **Privacy-preserving data integration and sharing**: Deploying a federated repository of encrypted data on a cloud server facilitates outsourcing multi-party computation (MPC) to the cloud [15], [27] and sharing the information among multiple owners [4], [30], [28]. Identity privacy requires that no one can associate an intermediate value with an individual party/owner that contributes to this value. For instance, the server may be eligible to find the minimum or maximum value in the federated dataset, but shouldn't know which party/owner provided it. This can be achieved by starting the protocol by obliviously shuffling data [5].
- **Privacy persevering computation**: secure computation over encrypted data enables the users to outsource various large-scale computational tasks (i.e., data analytics, data mining, machine learning) to a cloud. Many of these tasks require to separate certain sub-samples from the entire data (data filtering) or re-order the data for certain purposes (data sorting). For example, to train a deep neural network, within every epoch one often needs to sort the training data randomly and then operate iteratively on small subsets of the sorted data (mini-batch) at a time. When the data is encrypted due to privacy concerns, the above operations must be also formed obliviously because the exploitation of any side channels induced by disk, network, and memory access patterns may leak a surprisingly large amount of information [24]. Both oblivious filtering and sorting can be reduced to oblivious shuffling [17], [29], [24].

A. Problem Formalization

Our reference scenario is the typical cloud model [31] containing a fully trusted *client* and a "honest-but-curious" *server*. The client has limited computing resources (i.e., storage space and computational power) but the server does not have such limitations.

At initialization, the client has an array of n data blocks, $B = (B_1, \cdots, B_n)$, each of size m. The client encrypts each B_i to $[B_i]$, and outsources encrypted blocks $[B] = ([B_1], \cdots, [B_n])$ to the server. Note that n is termed as the number of blocks involved in a shuffle, not the total number of blocks in a data repository, thus can be sufficiently small. The block size m is measured by the number of encryption units for accommodating all data of a block. For example, assuming one unit allows 1Kb data for encryption, a block containing 1Mb data would have the block size $m = 1024$ that is represented as a column vector of length 1024 with each element being an encryption unit.

As the key of "obliviousness", the shuffling of $[B]$ should prevent the server from tracking any $[B_i]$ during the process, $1 \leq i \leq n$. This requirement can be formulated as below:

Definition 1 (Oblivious Shuffle (OS) [33])**.** *A random shuffling of n encrypted blocks $[B] = ([B_1], \cdots, [B_n])$ is oblivious if the server is unable to correlate any block before and after the shuffling.*

To prevent the server from tracking a shuffle, OS usually requires that outsourced blocks $[B]$ are encrypted with some semantically (IND-CPA) secure encryption schemes and all blocks in $[B]$ are *re-encrypted* during the shuffling, so that the ciphertexts of the same block become different and these ciphertexts are indistinguishable to the server [23], [26], [33]. The performance of an OS algorithm can be measured by *communication cost*, *client computation cost*, and *server computation cost*. Due to network bandwidth bottleneck and client resource limitation in the outsourcing scenario, minimizing communication cost and client computation cost is the focus of research interest [23], [26], [33].

Motivating Scenarios. In this paper, we focus on the application scenarios using OS for shuffling a small number of large-sized blocks, i.e., $m \gg n$, inspired by the following observations. First, in many cases, the data is represented as the blocks that have a large block size m (e.g., thousands), owing to the fact that more than 80% of all data is unstructured (e.g., images, videos, location information, and social media data) or semi-structured (e.g., XML documents or word processor files) large objects (LOBs) [6], [21].

On the other hand, for many applications oblivious shuffles often involve only a small number n of blocks (a typical n is in $[2, 10]$). For example, private data access requires to frequently permute the children of nodes within some tree-based storage for hiding access patterns [9], [18], [8], where each node normally has few children (such as 2 for binary trees in [9], [18]) but a child contains one or several LOB as one block each. Private data integration for outsourcing multi-party computation may require to shuffle the outsourced data from different parties for hiding identity privacy [5], where all LOBs belonging to the same parties create a gigantic data block but only a small number of parties ($\leq$10) is often involved in real applications [3]. Private computation for answering Top-k or k-NN query over LOB data may require to shuffle the k answers for hiding their relative ordering [20], [7], where every answer is an individual LOB but k is often sufficiently small ($\leq$10).

Under the scenario of $m \gg n$, it is important to eliminate the effect of block size m on the communication and client computation costs. This requires that OS is performed without moving outsourced blocks between the server and the client, regardless of how many shuffles are performed. We can formulate this problem as below:

Definition 2 (Repeatable Oblivious Shuffle (ROS))**.** *An oblivious shuffle of $[B] = ([B_1], \cdots, [B_n])$ is repeatable if its communication cost and client computation cost depend only on the number n of blocks, regardless of the number of shuffles.*

Essentially, ROS requires the data $[B]$ to be completely confined to the server during any shuffling, no matter how many shuffles are performed. Meeting this property turns out to be a major challenge. In fact, existing OS algorithms either treat the server as a simple storage device and perform the shuffle through downloading the outsourced data to the client and uploading the shuffled/re-encrypted data to the server [11], [12], [23], [26], [33], or permute outsourced data using server computation at the cost of increasing encryption layers each time, which requires periodically downloading outsourced data to the client for peeling off extra encryption layers [2], [9]. Therefore, existing OS algorithms are not repeatable because they all suffer from the $O(m)$ blowup in communication and client computation costs due to moving outsourced data blocks from the server to the client for re-encryption/peeling-off. The costly $O(m)$ blowup would limit their practical adoption for large-sized blocks.

B. Contributions

The goal of this work is to construct an efficient ROS algorithm that eliminates the $O(m)$ blowup in communication cost and client computation cost for the scenario of $m \gg n$. Our contributions are summarized as follows:

- (Section I-A) Motivated by the applications where $m \gg n$ holds, we present the notation of repeatable oblivious shuffle (ROS) as a tailored OS solution to overcome the typical communication bottleneck and limited client resources in outsourced environments.
- (Section IV) We construct the first practical ROS algorithm using efficient additively homomorphic encryption.
- (Section V) We give a rigorous security analysis of our ROS construction and show that it is secure.
- (Section VI) We show experimentally that our ROS construction outperforms the state-of-the-art OS algorithms in the motivated scenarios.

TABLE I: Comparison of OS algorithms over n data blocks of size m. For $m \gg n$, our ROS construction is asymptotically better than existing OS algorithms in terms of communication and client computation costs by avoiding the large term m.

OS Algorithms		Communication cost	Client computation cost	Server computation cost
Client-side shuffling	Zig-zag Sort [11]	$O(mn \log n)$	$O(mn \log n)$	—
	Melbourne Shuffle [23]	$O(mn)$	$O(mn)$	—
	Cache Shuffle [26]	$O(mn)$	$O(mn)$	—
	Buffer Shuffle [12]	$O(mn)$	$O(mn)$	—
	Interleave Buffer Shuffle [33]	$O(mn)$	$O(mn)$	—
Server-side shuffling	Layered Shuffle [2], [9] ($\ell \geq 1$)	$O(\ell n^2 + mn)$	$O(\ell n^2 + mn)$	$O(mn^2\ell^2)$
	Our ROS Construction	$\boldsymbol{O(n^2)}$	$\boldsymbol{O(n^2)}$	$\boldsymbol{O(mn^2)}$

II. Related Work

Existing OS algorithms fall into two general categories: *client-side shuffling* and *server-side shuffling*. The former depends on the client's computation for obliviously shuffling encrypted data, while the latter depends on the server's computation for achieving this purpose. Next, we review existing client-side and server-side shuffling algorithms and summarize them in Table I. The discussion is based on the shuffling of n outsourced data blocks of size m, i.e., $B = (B_1, \cdots, B_n)$.

A. Client-side Shuffling

Since outsourced data can be of unbounded size but the client has only limited storage, client-side shuffling commonly works in a multi-round manner. In each round, the client downloads a small portion of encrypted outsourced data to its local storage, shuffles it after decryption, re-encrypts the data and writes it back to the server. The early approach to oblivious shuffling in this category is based on oblivious sorting algorithms. The best bound is obtained by *Zig-zag Sort* [11], which involves $O(mn \log n)$ client cost and $O(mn \log n)$ communication cost. *Melbourne Shuffle* [23] is the first OS method that is not based on an oblivious sorting algorithm. The optimized Melbourne Shuffle has $O(mn)$ client cost and $O(mn)$ communication cost. Some other works such as *Buffer Shuffle* [12], *Interleave Buffer Shuffle* [33], and *Cache Shuffle* [26] also implement client-side shuffling with this cost complexity. All of these approaches have some constant factors in the aforementioned complexity.

B. Server-side Shuffling

This group of works leverages server-side computation to perform oblivious shuffle for reducing communication cost and client computation cost. It essentially performs a shuffle through computing a *homomorphic matrix multiplication* between outsourced data blocks and permutation matrix on the server.

Layered Shuffle [2] is the first concrete server-side shuffling. It requires a sequence of additively homomorphic encryption (AHE) schemes $\mathcal{E}_\ell$ where the ciphertext space of $\mathcal{E}_\ell$ is in the plaintext space of $\mathcal{E}_{\ell+1}$ and the ciphertext size of $\mathcal{E}_\ell$ is linear with ℓ, for all $\ell \geq 1$. Each scheme $\mathcal{E}_\ell$ is additively homomorphic meaning $\mathcal{E}_\ell(x) \oplus \mathcal{E}_\ell(y) = \mathcal{E}_\ell(x+y)$ and $\mathcal{E}_{\ell+1}(x) \otimes \mathcal{E}_\ell(y) = \mathcal{E}_{\ell+1}(\mathcal{E}_\ell(y) \cdot x)$. After $\ell-1$ consecutive shuffles, the current outsourced counterpart of B has ℓ encryption layers[1], notated by $\mathcal{E}^\ell(B)$. To perform the ℓ-th shuffle, the client encrypts a permutation matrix π with $\mathcal{E}_{\ell+1}$ and uploads $\mathcal{E}_{\ell+1}(\pi)$ to the server. Then the server performs homomorphic matrix multiplication using $\mathcal{E}^\ell(B)$ and $\mathcal{E}_{\ell+1}(\pi)$, which outputs the permuted result $\mathcal{E}^{\ell+1}(B) = \mathcal{E}_{\ell+1}\left(\mathcal{E}^\ell(B) \cdot \pi\right)$ that has $\ell+1$ encryption layers. Due to ciphertext expansion, the shuffling costs increase at a polynomial rate with the total number ℓ of shuffles so far. The average costs of the first ℓ consecutive shuffles include $O(\ell n^2)$ client cost for encrypting permutation matrix, $O(\ell n^2)$ communication cost for uploading encrypted permutation matrix, and $O(mn^2\ell^2)$ server cost for homomorphic matrix multiplication.

The costs of layered shuffle can become unbounded due to the unbounded increase of ℓ as more shuffles are performed. To solve this problem, [9] proposed to periodically, say after every ℓ shuffles, peel off extra encryption layers by downloading the current outsourced blocks $\mathcal{E}^{\ell+1}(B)$ of $\ell+1$ encryption layers to the client, removing extra layers and re-encrypting it, and uploading encrypted data of one layer to the server. This operation incurs $O(\ell mn)$ communication cost and $O(\ell mn)$ client cost amortized over the ℓ shuffles. Thus, the total cost per shuffle with peeling-off is $O(\ell n^2 + mn)$ client cost, $O(\ell n^2 + mn)$ communication cost, and $O(mn^2\ell^2)$ server cost.

Fully homomorphic encryption (FHE) enables an unlimited number of both homomorphic addition and multiplication. If both the blocks and permutation matrix were encrypted under FHE, the server can trivially perform homomorphic matrix multiplication on its own without interacting with the client. This ROS construction, however, is theoretically interesting because FHE is too far away from being practical [19].

From Table I, we can see that all existing OS algorithms suffer from the term $O(m)$ in client and communication costs. In Section IV, we will construct a ROS algorithm that eliminates this term and uses only the efficient AHE scheme.

[1]The initial outsourced blocks corresponds to $\ell = 1$, i.e., $\mathcal{E}^1(B) = \mathcal{E}_1(B)$.

III. Preliminaries

A. Cryptographic Primitives

Our ROS construction employs the Paillier cryptosystem [25], which is an AHE scheme providing semantic security. It has the public key N as the product of two large random primes[2], and the secret key as the least common multiple of these primes. In this paper, both the client and server have the public key but only the client has the secret key. Let $\mathbb{Z}_N$ denote the integers mod N and $\mathbb{Z}^*_{N^2}$ denote the integers coprime to N^2. Paillier cryptosystem encrypts a plaintext $x \in \mathbb{Z}_N$ to a ciphertext $[x] \in \mathbb{Z}^*_{N^2}$ with the public key and some randomness, so that encrypting the same plaintext multiple times yields different indistinguishable ciphertexts due to using different randomness each time. The exact encryption/decryption can be found in [25]. We focus on the following homomorphic properties that are essential to our construction later.

Let $x_i, y_i \in \mathbb{Z}_N$, $\vec{x} = (x_1, \cdots, x_n)$, $\vec{y} = (y_1, \cdots, y_n)^T$, and $\vec{x} \cdot \vec{y}$ be the dot product of $\vec{x}$ and $\vec{y}$.

1) Homomorphic addition

$$[x_1 + x_2] = [x_1] \times [x_2] \bmod N^2$$

2) Homomorphic multiplication

$$[x_1 \times x_2] = [x_1]^{x_2} \bmod N^2$$

3) Homomorphic dot product

Let $[\vec{x}] = ([x_1], \cdots, [x_n])$. Then we have the following equation from 1) and 2):

$$\begin{aligned} [\vec{x}] \odot \vec{y} &\stackrel{\text{def}}{=} ([x_1]^{y_1}) \times \cdots \times ([x_n]^{y_n}) \bmod N^2 \\ &= [\vec{x} \cdot \vec{y}] \end{aligned} \tag{1}$$

Here, each homomorphic multiplication $[x_i]^{y_i}$ is repeated homomorphic additions of $[x_i]$ to itself. Thus, the homomorphic dot product essentially computes a sequence of homomorphic additions involving all ciphertext in $[\vec{x}]$.

B. Computational Primitives

1) Matrix-based Data Shuffling: Any shuffling of n data blocks $B = (B_1, \cdots, B_n)$ can be executed by the matrix multiplication $B \cdot \pi$, for some $n \times n$ permutation matrix π. For example, the computation with $\pi = \left(\begin{smallmatrix} 0 & 1 \\ 1 & 0 \end{smallmatrix}\right)$ would swap the two blocks in $B = (B_1, B_2)$:

$$B \cdot \pi = (B_1, B_2) \cdot \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} = (B_2, B_1) \tag{2}$$

Assuming there are $\eta \geq 1$ consecutive shuffles over B where i-th shuffle permutes output of i-1-th shuffle according to permutation matrix $\pi^{(i)}$, for all $1 \leq i \leq \eta$. Then, the final result of these η shuffles would be given by $B \cdot \pi^\eta$, where π^η is the permutation matrix accumulating all η shuffles, i.e.,

$$\pi^\eta = \pi^{\eta\text{-}1} \cdot \pi^{(\eta)} = \pi^{(1)} \cdot \pi^{(2)} \cdots \pi^{(\eta)} \tag{3}$$

For consistency, we define π^0 as the $n \times n$ identity matrix so that $\pi^1 = \pi^0 \cdot \pi^{(1)} = \pi^{(1)}$.

[2]The actual public key of Paillier cryptosystem is (N, g) with $g = 1 + N$.

2) Matrix-based Data Scaling: Given n data blocks $B = (B_1, \cdots, B_n)$ and an $n \times n$ diagonal matrix C, the matrix multiplication $B \cdot C$ scales each block B_i with $C(i, i)$, for all $1 \leq i \leq n$. For example, if $B = (B_1, B_2)$ and $C = \left(\begin{smallmatrix} 2 & 0 \\ 0 & 3 \end{smallmatrix}\right)$, then we have

$$B \cdot C = (B_1, B_2) \cdot \begin{pmatrix} 2 & 0 \\ 0 & 3 \end{pmatrix} = (2B_1, 3B_2) \tag{4}$$

IV. Our Construction

From Definition 2, it is seen that a repeatable oblivious shuffle (ROS) protocol must satisfy both: (repeatability) the communication cost and client computation cost for shuffling n encrypted blocks $[B] = ([B_1], \cdots, [B_n])$ depend only on n, and (obliviousness) the shuffling is oblivious. In this section, we start by building a shuffle algorithm that satisfies repeatability but violates obliviousness. We explore nice properties of this algorithm and then give out our final ROS construction.

A. Basic Construction

As shown in Eqn (2), shuffling n blocks of B according to a permutation π is achieved through the matrix multiplication $B \cdot \pi$, which essentially computes the dot products between B and each column of π. If the outsourced blocks $[B] = ([B_1], \cdots, [B_n])$ are encrypted using Paillier cryptosystem, $[B]$ can be shuffled as follows: the client unloads the desired permutation π to the server, then the server can shuffle $[B]$ by computing the "homomorphic dot product" defined in Eqn (1) between $[B]$ and each column of π, i.e.,

$$[B] \odot \pi = [B \cdot \pi].$$

Clearly, this shuffling algorithm is not "oblivious" because the server learns the actual permutation π. However, the construction has some very nice properties: the client's job is to generate an $n \times n$ permutation matrix π and upload it to the server, which incurs $O(n^2)$ client cost and communication cost that depend only on n. In this sense, the construction is "repeatable".

These nice properties are given by the fact that the client "guides" the server to perform the shuffling homomorphically by sending the server some plaintext "helper instruction" that encodes the desired permutation π (here, π itself is sent). With helper instruction and $[B]$, the server's main job will be to trivially run "homomorphic dot product" operations.

B. Overview

To retain the nice properties above as well as make the shuffling become oblivious (that is, build a complete ROS), the key is to prevent the server from learning the actual permutation π from helper instruction. To achieve this purpose, we propose to accompany every shuffling of blocks (illustrated in Eqn (2)) by a scaling of these blocks (illustrated in Eqn (4)). We refine the ROS notation in Definition 2 by including such mix of shuffling and scaling as below:

Definition 3 (Refined ROS). *Consider $B = (B_1, \cdots, B_n)$, for any $\eta \geq 1$, the call of*

$$[B^{(\eta)}] \leftarrow ROS(\pi^{(\eta)}, C^{(\eta)}, [B^{(\eta\text{-}1)}])$$

permutes/scales $[B^{(\eta\text{-}1)}] = [B \cdot C^{(\eta\text{-}1)} \cdot \pi^{\eta\text{-}1}]$ into $[B^{(\eta)}] = [B \cdot C^{(\eta)} \cdot \pi^{\eta}]$ on the server without disclosing the permutation matrix $\pi^{(\eta)}$ and the scaling matrix $C^{(\eta)}$ to the server ($\pi^{\eta} = \pi^{\eta\text{-}1} \cdot \pi^{(\eta)}$ as defined in Eqn (3)*), while the communication cost and client computation cost depend only on n.*

Example 1. *Let B=(B_1, B_2) and $[B^{(\eta\text{-}1)}]$=$[B \cdot C^{(\eta\text{-}1)} \cdot \pi^{\eta\text{-}1}]$ w.r.t current scaling $C^{(\eta\text{-}1)}$=$\left(\begin{smallmatrix}2&0\\0&3\end{smallmatrix}\right)$ and accumulated permutation $\pi^{\eta\text{-}1}$=$\left(\begin{smallmatrix}0&1\\1&0\end{smallmatrix}\right)$. If inputting a permutation matrix $\pi^{(\eta)}$=$\left(\begin{smallmatrix}0&1\\1&0\end{smallmatrix}\right)$ and a scaling matrix $C^{(\eta)}$=$\left(\begin{smallmatrix}5&0\\0&4\end{smallmatrix}\right)$, ROS outputs $[B^{(\eta)}]$=$[B \cdot C^{(\eta)} \cdot \pi^{(\eta)}]$ that permutes $[B^{(\eta\text{-}1)}]$ according to $\pi^{(\eta)}$ and updates the scaling with $C^{(\eta)}$. For example, B_2 is moved to position 2 and scaled by 4 because $\pi^{(\eta)}(1,2) = 1$ and $C^{(\eta)}(2,2) = 4$. Table II shows the outcomes of such mixed shuffling and scaling.* □

TABLE II: Illustration of our ROS construction

	$[B^{(\eta\text{-}1)}] = ([3B_2], [2B_1])$
$[B^{(\eta)}] \leftarrow ROS(\pi^{(\eta)}, C^{(\eta)}, [B^{(\eta\text{-}1)}])$	$[B^{(\eta)}] = ([5B_1], [4B_2])$

To address of the challenge of hiding $\pi^{(\eta)}$ and $C^{(\eta)}$ from the server while allowing the client to guide the server to perform shuffling/scaling as specified, we propose the following strategy. The client constructs some *plaintext* helper instruction $H^{(\eta)}$ and the server generates some *encrypted* auxiliary blocks $[B_A^{(\eta)}]$. Here, $H^{(\eta)}$ and $B_A^{(\eta)}$ jointly encode $\pi^{(\eta)}$ and $C^{(\eta)}$, but $B_A^{(\eta)}$ is unknown to the server due to encryption. In this way, the server is unable to learn $\pi^{(\eta)}$ and $C^{(\eta)}$ from $H^{(\eta)}$, but still able to perform the shuffling/scaling through "homomorphic dot product" between $([B_A^{(\eta)}], [B^{(\eta\text{-}1)}])$ and $H^{(\eta)}$. The next example illustrates this idea.

Example 2. *To encode $\pi^{(\eta)}$ and $C^{(\eta)}$ in Example 1, let*

$$H^{(\eta)} = \begin{pmatrix} 3 & 0 \\ 0 & 1 \\ 1 & 1 \\ 4 & \text{-}1 \end{pmatrix}$$

and $[B_A^{(\eta)}] = ([-B_1 - B_2], [2B_1 + B_2])$. Then, the server can obliviously permute $[B^{(\eta\text{-}1)}]$ into $[B^{(\eta)}]$ by computing

$$[B^{(\eta)}] = ([B_A^{(\eta)}], [B^{(\eta\text{-}1)}]) \odot H^{(\eta)} \qquad (5)$$

or

$$([5B_1], [4B_2]) = \begin{pmatrix} [-B_1 - B_2] \\ [2B_1 + B_2] \\ [3B_2] \\ [2B_1] \end{pmatrix}^T \odot \begin{pmatrix} 3 & 0 \\ 0 & 1 \\ 1 & 1 \\ 4 & \text{-}1 \end{pmatrix}$$

During this computation[3]*, the LHS is the target shuffling/scaling specified by $\left(\begin{smallmatrix}5&0\\0&4\end{smallmatrix}\right) = C^{(\eta)} \cdot \pi^{\eta}$ with $\pi^{\eta} = \pi^{\eta\text{-}1} \cdot \pi^{(\eta)}$. The RHS, the actual computation to meet this target, can be rewritten into*

$$\begin{pmatrix} \vec{x_1} \cdot \vec{y_1} & \vec{x_1} \cdot \vec{y_2} \\ \vec{x_2} \cdot \vec{y_1} & \vec{x_2} \cdot \vec{y_2} \end{pmatrix}$$

where $\vec{x_i}$ is the coefficient vector of each block B_i in $([B_A^{(\eta)}], [B^{(\eta\text{-}1)}])$ and $\vec{y_i}$ is the i-th column of $H^{(\eta)}$, $i = 1, 2$. It is easy to see

$$\begin{cases} \vec{x_1} \cdot \vec{y_1} = (\text{-}1, 2, 0, 2) \cdot (3, 0, 1, 4)^T = 5 \\ \vec{x_2} \cdot \vec{y_1} = (\text{-}1, 1, 3, 0) \cdot (3, 0, 1, 4)^T = 0 \\ \vec{x_1} \cdot \vec{y_2} = (\text{-}1, 2, 0, 2) \cdot (0, 1, 1, \text{-}1)^T = 0 \\ \vec{x_2} \cdot \vec{y_2} = (\text{-}1, 1, 3, 0) \cdot (0, 1, 1, \text{-}1)^T = 4. \end{cases} \qquad (6)$$

In this sense, $H^{(\eta)}$ and $B_A^{(\eta)}$ jointly encode $\pi^{(\eta)}$ and $C^{(\eta)}$. □

Example 2 illustrates the following key ideas that underpin our ROS construction.

Correctness: For any $\pi^{(\eta)}$ and $C^{(\eta)}$, the client can *always* find the $H^{(\eta)}$ to encode them. Note that $\pi^{(\eta)}$ and $C^{(\eta)}$, accompanied by accumulated permutation $\pi^{\eta\text{-}1}$ so far, determine the target shuffling/scaling specified by $C^{(\eta)} \cdot \pi^{\eta} = \left(\begin{smallmatrix}5&0\\0&4\end{smallmatrix}\right)$, i.e., the RHS of Eqn (6). The client also knows $\vec{x_1}$ and $\vec{x_2}$ by tracking the previous shuffling/scaling in $[B^{(\eta\text{-}1)}]$ and the generation of $[B_A^{(\eta)}]$. Therefore, given $\vec{x_1}$ and $\vec{x_2}$, the client can always find a solution for $\vec{y_1}$ and $\vec{y_2}$ (i.e., $H^{(\eta)}$) using Eqn (6), because it is a under-determined linear system.

Obliviousness: $[B_A^{(\eta)}]$ and $H^{(\eta)}$ discloses no information about $\pi^{(\eta)}$ and $C^{(\eta)}$. In fact, for *any* choice of $\pi^{(\eta)}$ and $C^{(\eta)}$ (thus, any choice of the RHS of Eqn (6)), there always exists a solution for $\vec{x_i}$'s satisfying Eqn (6) w.r.t. the observed $\vec{y_i}$'s (i.e., $H^{(\eta)}$), due to the system being under-determined. The server cannot distinguish the actual $\pi^{(\eta)}$ and $C^{(\eta)}$ from these choices because actual $\vec{x_i}$'s are hidden in $[B^{(\eta\text{-}1)}]$ and $[B_A^{(\eta)}]$ and unknown to the server.

Repeatability: In this example, the client's job is to generate/upload the helper instruction $H^{(\eta)}$, the size of which depends solely on the block number n. Importantly, as we shall see in Section IV-C, the auxiliary blocks $[B_A^{(\eta)}]$ are generated by the server itself, using another helper instruction from the client that has a similar size to $H^{(\eta)}$. Therefore, the total client computation cost and communication cost, due to generating/uploading these two help instructions, depends on the block number n but nothing else.

C. Algorithm

Initialization. Recall that, n is the number of blocks in B and N is the public key of Paillier cryptosystem. Initially, the client randomly chooses an $n \times n$ diagonal matrix $C^{(0)}$ over

[3]All computations are over the ring $\mathbb{Z}_N$.

$\mathbb{Z}_N^*$, an $n\times n$ invertible full matrix $S^{(0)}$ over $\mathbb{Z}_N$, and computes $B^{(0)}$ and $B_A^{(0)}$ by

$$B^{(0)} = B \cdot C^{(0)} \mod N \tag{7}$$

$$B_A^{(0)} = B \cdot S^{(0)} \mod N \tag{8}$$

The client then encrypts $B^{(0)}$ to $[B^{(0)}]$ with Paillier cryptosystem, and uploads $\{[B^{(0)}], B_A^{(0)}\}$ to the server. Table III summarizes all notations used throughout the rest of the paper. We say that a full matrix is over $\mathbb{Z}_N$ or $\mathbb{Z}_N^*$ if all its elements are in that ring, and a diagonal matrix is over $\mathbb{Z}_N^*$ if all its diagonal elements are in $\mathbb{Z}_N^*$.

Main Protocol. Algorithm 1 describes the steps of our ROS construction. For any $\eta \geq 1$, with inputting an $n\times n$ permutation matrix $\pi^{(\eta)}$ and an $n\times n$ scaling matrix $C^{(\eta)}$ (that is, a diagonal matrix over $\mathbb{Z}_N^*$), it obliviously permutes/scales $[B^{(\eta\text{-}1)}]$ to $[B^{(\eta)}]$ in two phases.

- *Phase 1*. The client generates an $n\times n$ random matrix $S^{(\eta)}$, computes an $n\times n$ matrix $H_A^{(\eta)}$ (line 1) and an $2n\times n$ matrix $H^{(\eta)}$ (line 2), encrypts $H_A^{(\eta)}$ to $[H_A^{(\eta)}]$ element-wisely with Paillier cryptosystem and sends $\{[H_A^{(\eta)}], H^{(\eta)}\}$ to the server (line 3). The client also updates the accumulated permutation matrix π^{η} (line 4).
- *Phase 2*. The server computes $[B_A^{(\eta)}]$ using $\{[H_A^{(\eta)}], B_A^{(0)}\}$ (line 5), where $[H_A^{(\eta)}]$ is the helper instruction for guiding the server to generate $[B_A^{(\eta)}]$. Then the server computes $[B^{(\eta)}]$ from $[B^{(\eta\text{-}1)}]$ using $\{[B_A^{(\eta)}], H^{(\eta)}\}$ (line 6).

Complexity Analysis. During the η-th calling of ROS for shuffling n encrypted blocks, each of size m, the client generates an $2n\times n$ matrix $H^{(\eta)}$, an $n\times n$ matrix $[H_A^{(\eta)}]$, and sends them to the server. The client cost and the communication cost are bounded by the size of these matrices, i.e., $O(n^2)$. The server cost involves $O(mn)$ homomorphic dot products with $O(n)$ cost each to compute $[B_A^{(\eta)}]$ and $[B^{(\eta)}]$ (lines 5, 6), so the total cost is $O(mn^2)$.

As for space overhead, the client use $O(n)$ space for storing the information in $C^{(\eta\text{-}1)}$ and $\pi^{\eta\text{-}1}$, and $O(n^2)$ space for $S^{(0)}$. On the server, the space required is $O(mn)$ for storing $B_A^{(0)}$ and $[B^{(\eta\text{-}1)}]$. These spaces are not accumulated over different calls of ROS because they are reused by every call.

D. Correctness Analysis

Theorem 1. *For any $\eta \geq 1$, Algorithm 1 produces $[B_A^{(\eta)}] = [B \cdot S^{(\eta)}]$ and $[B^{(\eta)}] = [B \cdot C^{(\eta)} \cdot \pi^{\eta}]$.*

Proof. Let us consider the proof for $[B_A^{(\eta)}]$ first. $[B_A^{(\eta)}]$ is computed by Eqn (12) as follows,

$$\begin{aligned}
&[B_A^{(\eta)}]^T = [H_A^{(\eta)}]^T \odot (B_A^{(0)})^T \\
\Rightarrow &[(B_A^{(\eta)})^T] = [(H_A^{(\eta)})^T \cdot (B_A^{(0)})^T] && \text{(a)} \\
\Rightarrow &[B_A^{(\eta)}] = [B_A^{(0)} \cdot H_A^{(\eta)}] && \text{(b)} \\
\Rightarrow &[B_A^{(\eta)}] = [B \cdot S^{(0)} \cdot (S^{(0)})^{\text{-}1} \cdot S^{(\eta)}] && \text{(c)} \\
\Rightarrow &[B_A^{(\eta)}] = [B \cdot S^{(\eta)}] && \text{(d)}
\end{aligned}$$

Algorithm 1 $[B^{(\eta)}] \leftarrow ROS(\pi^{(\eta)}, C^{(\eta)}, [B^{(\eta\text{-}1)}])$

Require: The client has $S^{(0)}$, $C^{(\eta\text{-}1)}$, and $\pi^{\eta\text{-}1}$; the server has $B_A^{(0)}$ and $[B^{(\eta\text{-}1)}]$

Phase 1 (Client):

1: randomly generate $S^{(\eta)}$ as an $n\times n$ full matrix over $\mathbb{Z}_N^*$ and compute

$$H_A^{(\eta)} = (S^{(0)})^{-1} \cdot S^{(\eta)} \mod N \tag{9}$$

2: compute

$$H^{(\eta)} = \begin{pmatrix} H_1^{(\eta)} \\ H_2^{(\eta)} \end{pmatrix}$$

where $H_1^{(\eta)}$ is an $n\times n$ diagonal matrix over $\mathbb{Z}_N^*$ satisfying

$$C^{(\eta)} \cdot \pi^{\eta\text{-}1} \cdot \pi^{(\eta)} - S^{(\eta)} \cdot H_1^{(\eta)} \text{ is over } \mathbb{Z}_N^* \tag{10}$$

and $H_2^{(\eta)}$ is an $n\times n$ full matrix over $\mathbb{Z}_N^*$ satisfying

$$S^{(\eta)} \cdot H_1^{(\eta)} + C^{(\eta\text{-}1)} \cdot \pi^{\eta\text{-}1} \cdot H_2^{(\eta)} = C^{(\eta)} \cdot \pi^{\eta\text{-}1} \cdot \pi^{(\eta)} \mod N \tag{11}$$

3: encrypt $H_A^{(\eta)}$, send $[H_A^{(\eta)}]$ and $H^{(\eta)}$ to the server

4: $\pi^{\eta} \leftarrow \pi^{\eta\text{-}1} \cdot \pi^{(\eta)}$

Phase 2 (Server):

5: generate auxiliary blocks by computing

$$[B_A^{(\eta)}]^T = [H_A^{(\eta)}]^T \odot (B_A^{(0)})^T \tag{12}$$

6: perform the shuffle by computing

$$[B^{(\eta)}] = \left([B_A^{(\eta)}], [B^{(\eta\text{-}1)}]\right) \odot H^{(\eta)} \tag{13}$$

(a) follows from homomorphic dot product defined in Eqn (1). By removing the matrix transpose of (a), we get (b). (c) follows from Eqn (8) and Eqn (9). (d) holds because $(S^{(0)})^{\text{-}1}$ is the inverse of $S^{(0)}$. This shows $[B_A^{(\eta)}] = [B \cdot S^{(\eta)}]$.

The proof for $[B^{(\eta)}]$ is by induction on η. The basis is $[B^{(0)}] = [B \cdot C^{(0)} \cdot \pi^0]$, which comes from Eqn (7) with π^0 being the identity matrix. Assume

$$[B^{(\eta\text{-}1)}] = [B \cdot C^{(\eta\text{-}1)} \cdot \pi^{\eta\text{-}1}] \tag{14}$$

We show $[B^{(\eta)}] = [B \cdot C^{(\eta)} \cdot \pi^{\eta}]$. From Eqn (13), we have

$$\begin{aligned}
&[B^{(\eta)}] = ([B_A^{(\eta)}], [B^{(\eta\text{-}1)}]) \odot H^{(\eta)} \\
\Rightarrow &[B^{(\eta)}] = [B_A^{(\eta)} \cdot H_1^{(\eta)} + B^{(\eta\text{-}1)} \cdot H_2^{(\eta)}] && \text{(a)} \\
\Rightarrow &[B^{(\eta)}] = [B \cdot (S^{(\eta)} \cdot H_1^{(\eta)} + C^{(\eta\text{-}1)} \cdot \pi^{\eta\text{-}1} \cdot H_2^{(\eta)})] && \text{(b)} \\
\Rightarrow &[B^{(\eta)}] = [B \cdot C^{(\eta)} \cdot \pi^{\eta\text{-}1} \cdot \pi^{(\eta)}] && \text{(c)} \\
\Rightarrow &[B^{(\eta)}] = [B \cdot C^{(\eta)} \cdot \pi^{\eta}] && \text{(d)}
\end{aligned}$$

Recall that $H^{(\eta)}$ consists of $H_1^{(\eta)}$ and $H_2^{(\eta)}$. (a) comes from computing homomorphic dot product of Eqn (13). (b) is

TABLE III: Parameters and notations in the ROS construction ($\eta \geq 1$)

Notation	Meaning
B	n blocks of size m, $B = (B_1, \cdots, B_n)$
$[B^{(\eta)}]$	shuffling result of η-th calling, $[B^{(\eta)}] = [B \cdot C^{(\eta)} \cdot \pi^{\eta}]$
$B_A^{(0)}$	initially outsourced auxiliary blocks, $B_A^{(0)} = B \cdot S^{(0)}$
$[B_A^{(\eta)}]$	auxiliary blocks used by η-th calling, $[B_A^{(\eta)}] = [B \cdot S^{(\eta)}]$
$H^{(\eta)}$	helper instruction for computing $[B^{(\eta)}]$
$[H_A^{(\eta)}]$	helper instruction for computing $[B_A^{(\eta)}]$
$C^{(\eta)}$	scaling matrix of η-th calling
$\pi^{(\eta)}$	permutation matrix of η-th calling
π^{η}	accumulated permutation of η callings, $\pi^{\eta} = \pi^{\eta\text{-}1} \cdot \pi^{(\eta)}$
$S^{(0)}$	coefficient matrix of $B_A^{(0)}$
$S^{(\eta)}$	coefficient matrix of $B_A^{(\eta)}$

obtained from Eqn (14) and $B_A^{(\eta)} = B \cdot S^{(\eta)}$ shown in the first part. (c) is obtained from Eqn (11). Finally, (d) holds as $\pi^{\eta\text{-}1} \cdot \pi^{(\eta)} = \pi^{\eta}$. □

The next theorem shows the existence of $H^{(\eta)}$.

Theorem 2. *For any $\eta \geq 1$, $H^{(\eta)}$ constrained by Eqns* (10) *and* (11) *always exists.*

Proof. $H^{(\eta)}$ is mainly constrained by Eqn (11). Typically, $C^{(\eta)} \cdot \pi^{\eta\text{-}1} \cdot \pi^{(\eta)}$ specifies the target shuffling/scaling as $[B^{(\eta)}] = [B \cdot C^{(\eta)} \cdot \pi^{\eta}]$; $S^{(\eta)}$ specifies the coefficient of each block B_i in the auxiliary blocks $[B_A^{(\eta)}]$ as $[B_A^{(\eta)}] = [B \cdot S^{(\eta)}]$; $C^{(\eta\text{-}1)} \cdot \pi^{\eta\text{-}1}$ specifies the coefficient of each B_i in the current outsourced data $[B^{(\eta\text{-}1)}]$ as $[B^{(\eta\text{-}1)}] = [B \cdot C^{(\eta\text{-}1)} \cdot \pi^{\eta\text{-}1}]$. In a similar spirit to Eqn (6), Eqn (11) is an underdetermined linear system with $H^{(\eta)}$'s entries being unknown variables: the matrix computation in Eqn (11) defines n^2 linear equations but there are $n^2 + n$ unknown variables in $H^{(\eta)}$ (diagonal matrix $H_1^{(\eta)}$ has n unknowns and full matrix $H_2^{(\eta)}$ has n^2 unknowns). Thus, a solution for $H^{(\eta)}$ always exists. □

V. Security Analysis

For any $\eta \geq 1$, from the server's perspective, the calling of Algorithm 1 involves the following *observed data*:

- encrypted data: $Enc^{(\eta)} = \{[B^{(\eta\text{-}1)}], [B^{(\eta)}], [B_A^{(\eta)}], [H_A^{(\eta)}]\}$
- non-encrypted data: $Non\_Enc^{(\eta)} = \{B_A^{(0)}, H^{(\eta)}\}$

and *non-observed data*:

- $\Theta_0 = \{B, S^{(0)}\}, \Theta_1^{(\eta)} = \{C^{(\eta)}, \pi^{\eta\text{-}1}, \pi^{(\eta)}\}, \Theta_2^{(\eta)} = \{S^{(\eta)}, C^{(\eta\text{-}1)}\}$

To claim the security of our construction, we show that the observed data discloses no information about the non-observed data. We first show that encrypted data $Enc^{(\eta)}$ observed by the server discloses nothing in Section V-A, then complete our security analysis by showing that the non-encrypted data $Non\_Enc^{(\eta)}$ discloses no information about Θ_0, $\Theta_1^{(\eta)}$, and $\Theta_2^{(\eta)}$ in Section V-B.

A. Security of $Enc^{(\eta)}$

The data in $Enc^{(\eta)}$ is either encrypted/uploaded by the client, or computed by the server through homomorphic dot product operations defined in Eqn (1). Thanks to the semantic security of Paillier cryptosystem (i.e., any ciphertext discloses nothing about the corresponding plaintext) and its homomorphic properties (i.e., any computation through homomorphic operations preserves the privacy of original data and computed results), the privacy of all data in $Enc^{(\eta)}$ is guaranteed.

Oblivious shuffling also requires $[B^{(\eta)}]$ to be a re-encryption of $[B^{(\eta\text{-}1)}]$ using different randomness, so that the server cannot track the permutation from the ciphertexts. Next, we show that our construction indeed achieves such re-encryption for outsourced blocks. Recall that homomorphic addition of Paillier cryptosystem propagates the randomness of both inputs $[x_1]$ and $[x_2]$ into the output $[x_1 + x_2]$ [25]. As homomorphic dot product defined in Eqn (1) is composed of multiple homomorphic additions involving each element of $[\vec{x}]$, it propagates the randomness of all ciphertexts of $[\vec{x}]$ into the result. During the η-th calling of Algorithm 1, the client generates a newly encrypted matrix $[H_A^{(\eta)}]$ with some fresh randomness. These fresh randomness are propagated into the ciphertexts $[B_A^{(\eta)}]$ first due to computing the homomorphic dot products of Eqn (12) and finally propagated into the ciphertexts $[B^{(\eta)}]$ due to computing the homomorphic dot product of Eqn (13). Thus, $[B^{(\eta)}]$ is a re-encryption of $[B^{(\eta\text{-}1)}]$ with fresh randomness.

B. Security of $Non\_Enc^{(\eta)}$

The rationale behind $Non\_Enc^{(\eta)}$ disclosing no information about non-observed data Θ_0, $\Theta_1^{(\eta)}$, and $\Theta_2^{(\eta)}$ is that there are many different choices of non-observed data for producing the same $Non\_Enc^{(\eta)}$ but the server is unable to identify the true choices as they are unobserved.

Theorem 3. *Given* $Non\_Enc^{(\eta)}$*, for any choice of* $\tilde{\Theta}_1^{(\eta)} = \{\tilde{C}^{(\eta)}, \tilde{\pi}^{\eta\text{-}1}, \tilde{\pi}^{(\eta)}\}$*, there exists a choice of* $\tilde{\Theta}_2^{(\eta)} = \{\tilde{S}^{(\eta)}, \tilde{C}^{(\eta\text{-}1)}\}$ *such that Eqn* (10) *and* (11) *remain to hold if* $\{\Theta_1^{(\eta)}, \Theta_2^{(\eta)}\}$ *is replaced with* $\{\tilde{\Theta}_1^{(\eta)}, \tilde{\Theta}_2^{(\eta)}\}$.

Proof. By replacing $\{\Theta_1^{(\eta)}, \Theta_2^{(\eta)}\}$ with $\{\tilde{\Theta}_1^{(\eta)}, \tilde{\Theta}_2^{(\eta)}\}$, Eqn (11) becomes

$$\tilde{S}^{(\eta)} \cdot H_1^{(\eta)} + \tilde{C}^{(\eta\text{-}1)} \cdot \tilde{\pi}^{\eta\text{-}1} \cdot H_2^{(\eta)} = \tilde{C}^{(\eta)} \cdot \ \tilde{\pi}^{\eta\text{-}1} \cdot \tilde{\pi}^{(\eta)} \mod N$$

This equation is a generalized form of Eqn (6). Similar to the argument of "Obliviousness" in Example 2, for any choice of $\tilde{\Theta}_1^{(\eta)}$ and the given $H^{(\eta)}$, this linear system is under-determined with $\tilde{\Theta}_2^{(\eta)}$=$\{\tilde{S}^{(\eta)}, \tilde{C}^{(\eta\text{-}1)}\}$ being the unknown variables: the matrix computation above defines n^2 equations but there are $n^2 + n$ unknowns (full matrix $\tilde{S}^{(\eta)}$ has n^2 variables and diagonal matrix $\tilde{C}^{(\eta\text{-}1)}$ has n variables). So a solution for $\tilde{\Theta}_2^{(\eta)}$ letting Eqn (11) remain to hold always exists.

Next, to satisfy Eqn (10), the above $\tilde{C}^{(\eta\text{-}1)}$ and $\tilde{\pi}^{\eta\text{-}1}$ should also enforce that $\tilde{C}^{(\eta\text{-}1)} \cdot \tilde{\pi}^{\eta\text{-}1} \cdot H_2^{(\eta)}$ is over $\mathbb{Z}_N^*$ (the above equation gives Eqn (10) by moving $\tilde{S}^{(\eta)} \cdot H_1^{(\eta)}$ to the RHS). This condition indeed holds according to [10] because $\tilde{C}^{(\eta\text{-}1)}$ is a diagonal matrix over $\mathbb{Z}_N^*$, $H_2^{(\eta)}$ is a full matrix over $\mathbb{Z}_N^*$, and $\tilde{C}^{(\eta\text{-}1)} \cdot \tilde{\pi}^{\eta\text{-}1} \cdot H_2^{(\eta)}$ corresponds to permute/scale the rows of $H_2^{(\eta)}$ using $\tilde{\pi}^{\eta\text{-}1}$ and $\tilde{C}^{(\eta\text{-}1)}$. □

From Theorem 3, given the observed $Non\_Enc^{(\eta)}$, the server cannot distinguish $\{\Theta_1^{(\eta)}, \Theta_2^{(\eta)}\}$ from $\{\tilde{\Theta}_1^{(\eta)}, \tilde{\Theta}_2^{(\eta)}\}$. Since $\tilde{\pi}^{(\eta)}$ and $\tilde{C}^{(\eta)}$ in $\tilde{\Theta}_1^{(\eta)}$ are arbitrarily chosen (subject to the constraints required), the server cannot determine the true permutation matrix $\pi^{(\eta)}$ and true scaling matrix $C^{(\eta)}$ from the *η-th calling* of ROS. The next corollary shows that no information about the non-observed Θ_0, $\Theta_1^{(\eta)}$, $\Theta_2^{(\eta)}$ is disclosed even if the server has access to all $Non\_Enc^{(i)}$ for *all previous calls* $1 \le i \le \eta$.

Corollary 1. *Even if the server has access to* $Non\_Enc^{(i)}$ *for all* $1 \le i \le \eta$*, the server is still unable to infer* $\Theta_0, \Theta_1^{(\eta)}, \Theta_2^{(\eta)}$.

Proof. Theorem 3 shows that, for any $\tilde{\Theta}_1^{(\eta)}$, we can find a $\tilde{\Theta}_2^{(\eta)}$ to satisfy Eqn (10) and (11), with the given $Non\_Enc^{(\eta)}$. This $\{\tilde{\Theta}_1^{(\eta)}, \tilde{\Theta}_2^{(\eta)}\}$ explicitly gives $\{\tilde{C}^{(\eta\text{-}1)}, \tilde{\pi}^{\eta\text{-}1}\}$.

Let $\tilde{\Theta}_1^{(\eta\text{-}1)} = \{\tilde{C}^{(\eta\text{-}1)}, \tilde{\pi}^{\eta\text{-}2}, \tilde{\pi}^{(\eta\text{-}1)}\}$ w.r.t. the above $\{\tilde{C}^{(\eta\text{-}1)}, \tilde{\pi}^{\eta\text{-}1}\}$ and $\tilde{\pi}^{\eta\text{-}2}$ be an arbitrary permutation, repeating the argument of Theorem 3 for the $(\eta\text{-}1)$-th calling, we can find a $\tilde{\Theta}_2^{(\eta\text{-}1)}$. This argument can be repeated for all $1 \le i \le \eta$, in the order of $i = \eta, \eta\text{-}1, \cdots, 1$. Note that the observed $Non\_Enc^{(\eta)}, \cdots, Non\_Enc^{(1)}$ are preserved through all these replacements. Therefore, the server cannot infer $\Theta_1^{(\eta)}$ and $\Theta_2^{(\eta)}$ from these $Non\_Enc^{(i)}$.

Lastly, to see that Θ_0 cannot be inferred, consider any choice of an $n \times n$ invertible matrix $\tilde{S}^{(0)}$ over $\mathbb{Z}_N$, with the inverse $(\tilde{S}^{(0)})^{-1}$, and define $\tilde{B} = B \cdot S^{(0)} \cdot (\tilde{S}^{(0)})^{-1} \mod N$. Then we have $\tilde{B} \cdot \tilde{S}^{(0)} \mod N = B \cdot S^{(0)} \mod N$; that is, $B_A^{(0)}$ is preserved (Eqn (8)) after replacing $\Theta_0 = \{B, S^{(0)}\}$ with $\tilde{\Theta}_0 = \{\tilde{B}, \tilde{S}^{(0)}\}$. Therefore, the server cannot distinguish $\Theta_0 = \{B, S^{(0)}\}$ from $\tilde{\Theta}_0 = \{\tilde{B}, \tilde{S}^{(0)}\}$. Although such replacement will affect $B^{(\eta)}$ and $B_A^{(\eta)}$, these data are encrypted and thus can not help the server's attacks. □

VI. Performance Evaluation

This section reports the empirical evaluation of our ROS construction by comparing it with the competitors discussed in Section II. All methods are implemented in C with OpenMP parallel programming on a server machine 96 Intel Core i7-3770 CPUs at 3.40 GHz, and a client machine with 2 Intel Core e7-4860 CPUs at 2.60 GHz. Both run a Linux system.

The empirical comparisons are based on applying each OS algorithm to permute n encrypted blocks of size m. The implementation details of each OS algorithm in the experiments are summarized as below:

- *ClientShuffle*. Different client-side shuffling algorithms have significantly varied implementations but commonly permute outsourced data in multi-rounds and each round downloads and permutes a small portion of the data on the client. To unify diverse client-side shuffling algorithms in our experiments, we adopt a simplified single-round implementation for client-side shuffling that downloads all encrypted blocks to the client, re-encrypts these blocks, and uploads them to the server in the permuted order. This simplification is in favor of client-side shuffling algorithms by reducing their shuffling costs, because every block is downloaded and re-encrypted only once while their original multi-round implementations involve downloading and re-encrypting each block multiple times. We follow the same encryption setting as [33] to implement such simplified client-side shuffling through encrypting the blocks with AES-128 from Crypto++ Library [1] (each encryption unit contains 128-bit data).
- *LayeredShuffle*: Layered Shuffle [2], [9] is the sole server-side shuffling algorithm but must peel off extra encryption layers after every ℓ shuffles, where ℓ is usually a small number due to the higher shuffle cost associated with the increased ℓ. Let LayeredShuffle ($\ell = 2$) and LayeredShuffle ($\ell = 10$) denote layered shuffle with ℓ being 2 and 10. Layered shuffle is implemented by adopting the library of Damgard-Jurik cryptosystem in [32] with 1024-bit key size for encryption (each encryption unit contains 1024-bit data).
- *ROS*. Our ROS construction (Section IV) is implemented by adopting the library of Paillier Cryptosystem in [32] with 1024-bit key size for encryption (each encryption unit contains 1024-bit data).

As different OS algorithms adopt encryption schemes that vary the size of an encryption unit, the block size m in the experiments indicates the size of a plaintext block in MB, instead of the number of encryption units for holding a block.

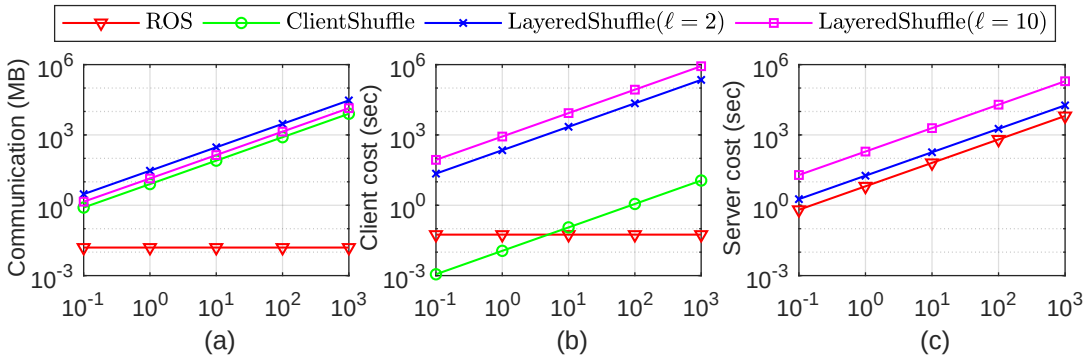

Fig. 1: Shuffle cost w.r.t. block size m (MB) ($n = 4$, ClientShuffle has no server computation and thus not reported)

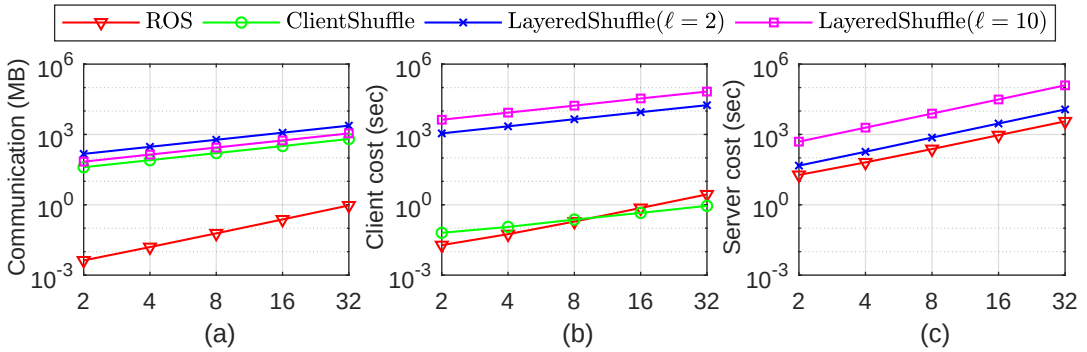

Fig. 2: Shuffle cost w.r.t. block number n ($m = 10$MB, ClientShuffle has no server computation and thus not reported)

A. *Effect of Block Size*

We first conduct an experiment to examine the performance of different OS algorithms for shuffling a fixed number ($n = 4$) of blocks with varying the block size m from 0.1 MB (e.g., a LOB of a document file) to 1,000 MB (e.g., a LOB of a video file). We evaluate them with three measurements: communication cost (MB), client computation cost (second), and server computation cost (second).

As shown in Figure 1(a) and 1(b), the communication cost and client computation cost of both client-side shuffling and layered shuffle grow linearly to the block size m, while those of our ROS construction stay at a constant around 16KB for communication and 0.05 second for client computation. ROS successfully achieves our design goals for eliminating the effect of the block size m on these two costs, because of strictly limiting the outsourced data blocks to the server and only communicating "helper instruction" of size $O(n^2)$ with the server. In contrast, client-side shuffling incurs an unbounded increase in communication and client computation costs due to the client's performing shuffles through downloading all outsourced data to its local storage and re-encrypting them; layered shuffle has the same drawback due to downloading all outsourced data of $\ell+1$ encryption layers for peeling off extra layers after every ℓ shuffles. These results are consistent with the asymptotical superiority of ROS in Table I.

In Figure 1(b), we observe that client-side shuffling outperforms layered shuffle in client cost. This seems counter intuitive as their client cost complexity are similar (recall that $m \gg n$ holds in our setting and ℓ is a small constant, thus from Table I the client costs of both algorithms are dominated by $O(mn)$). The reason is that client-side shuffling does not require any homomorphic operation and thus adopts the secret key cryptosystem AES for fast encryption/decryption, while layered shuffle adopts the public-key Damgard-Jurik cryptosystem for allowing additively homomorphic but having much slower encryption/decryption. The same reason also explains why client-side shuffling may beat ROS in client cost when the block size m is sufficiently small (e.g., ≤ 1 MB).

Figure 1(c) presents our results on server computation cost. Although both ROS and layered shuffle show a linear increase with the block size m, ROS outperforms layered shuffling on all settings of m. The costs of ROS are independent of the number of shuffles performed so far due to ROS never increasing encryption layers of outsourced data. However, layered shuffle adds an extra encryption layer after every shuffle and thus leads to the increase of the shuffling costs (especially the client and server computation costs) with the number ℓ of consecutive shuffles before a peeling-off. Nevertheless, layered shuffle is inapplicable to the scenarios in which shuffling is intensively involved.

B. Effect of Block Number

We also compare the performance of different OS methods with respective to varied number n of blocks while fixing the block size $m = 10$ MB (e.g., a LOB of an image file).

As shown in Figure 2, all algorithms exhibit an increase in communication cost and client computation cost. ROS grows slightly faster than client-side shuffling. The major reason is that the two costs of ROS ($O(n^2)$) increase quadratically to the block number n due to generating/uploading helper instructions of size $O(n^2)$ by the client, while those of client-side shuffling ($O(mn)$) are linear to the block number n, as summarized in Table I. In this sense, the saving of the communication cost and client computation cost using ROS becomes more significant for small n.

C. Summary

Through the experiments, we can see that ROS evidently outperforms existing OS algorithms when a small number of large-sized blocks are shuffled (i.e., $m \gg n$) because its communication and client costs grow with the square of n but independent of m. On the other hand, client-side shuffling becomes a better option if a large number of small-sized blocks are shuffled (i.e., $n \gg m$) because its communication and client costs are linear to both m and n. Lastly, layered shuffling can only be used when the outsourced data is shuffled by a limited number of times; otherwise, expensive peeling-off operations are frequently involved that introduce overwhelming communication and client costs.

VII. Conclusion

In this paper, we study the problem of oblivious algorithms for shuffling outsourced data blocks. We introduce repeatable oblivious shuffling (ROS), a fine-grained notation of oblivious shuffling that eliminates the effect of block size on the communication and client computation costs. ROS provides a tailored OS solution for the scenario of shuffling a small number of large-sized blocks to overcome the typical network bandwidth bottleneck and client resource limitation in outsourced environments. We present the first practical ROS construction using Paillier cryptosystem. According to experimental results, our construction significantly outperforms the state-of-the-art OS algorithms in the motivated scenarios.

References

[1] Crypto++. https://www.cryptopp.com/.

[2] B. Adida and D. Wikström. How to shuffle in public. In *Proc. TCC'07*, page 555. 2007.

[3] D. W. Archer, D. Bogdanov, Y. Lindell, L. Kamm, K. Nielsen, J. I. Pagter, N. P. Smart, and R. N. Wright. From keys to databasesreal-world applications of secure multi-party computation. *The Computer Journal*, 61(12):1749–1771, 2018.

[4] S. Bajaj and R. Sion. Trusteddb: A trusted hardware-based database with privacy and data confidentiality. *IEEE Transactions on Knowledge and Data Engineering*, 26(3):752–765, 2014.

[5] D. Bogdanov, L. Kamm, S. Laur, P. Pruulmann-Vengerfeldt, R. Talviste, and J. Willemson. Privacy-preserving statistical data analysis on federated databases. In *Annual Privacy Forum*, pages 30–55. Springer, 2014.

[6] L. Cai and Y. Zhu. The challenges of data quality and data quality assessment in the big data era. *Data Science Journal*, 14, 2015.

[7] N. Cao, C. Wang, M. Li, K. Ren, and W. Lou. Privacy-preserving multi-keyword ranked search over encrypted cloud data. *IEEE Transactions on Parallel and Distributed Systems*, 25(1):222–233, 2014.

[8] S. De Capitani di Vimercati, S. Foresti, S. Paraboschi, G. Pelosi, and P. Samarati. Efficient and private access to outsourced data. In *Proc. ICDCS'11*, page 710, 2011.

[9] S. Devadas, M. van Dijk, C. W. Fletcher, L. Ren, E. Shi, and D. Wichs. Onion oram: A constant bandwidth blowup oblivious ram. In *Proc. TCC'16*, page 145. 2016.

[10] D. S. Dummit and R. M. Foote. *Abstract algebra*, volume 3. Wiley Hoboken, 2004.

[11] M. T. Goodrich. Zig-zag sort: A simple deterministic data-oblivious sorting algorithm running in o(nlogn) time. In *Proc. STOC'14*, page 684, 2014.

[12] M. T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia. Practical oblivious storage. In *Proc. CODASPY'12*, pages 13–24, 2012.

[13] P. Grubbs, K. Sekniqi, V. Bindschaedler, M. Naveed, and T. Ristenpart. Leakage-abuse attacks against order-revealing encryption. In *Proc. SP'17*, page 655, 2017.

[14] M. S. Islam, M. Kuzu, and M. Kantarcioglu. Access pattern disclosure on searchable encryption: ramification, attack and mitigation. In *Proc. NDSS'12*, page 12, 2012.

[15] S. Kamara, P. Mohassel, and M. Raykova. Outsourcing multi-party computation. *IACR Cryptology ePrint Archive*, 2011:272, 2011.

[16] G. Kellaris, G. Kollios, K. Nissim, and A. O'Neill. Generic attacks on secure outsourced databases. In *Proc. CCS'16*, page 1329, 2016.

[17] S. Laur, J. Willemson, and B. Zhang. Round-efficient oblivious database manipulation. In *Proc. ISC'11*, pages 262–277, 2011.

[18] P. Lin and K. S. Candan. Hiding traversal of tree structured data from untrusted data stores. In *Proc. WOSIS'04*, page 314, 2004.

[19] P. Martins, L. Sousa, and A. Mariano. A survey on fully homomorphic encryption: An engineering perspective. *ACM Computing Surveys (CSUR)*, 50(6):83, 2017.

[20] X. Meng, H. Zhu, and G. Kollios. Top-k query processing on encrypted databases with strong security guarantees. In *Proc. ICDE'18*, pages 353–364, 2018.

[21] O. Müller, I. Junglas, J. v. Brocke, and S. Debortoli. Utilizing big data analytics for information systems research: challenges, promises and guidelines. *European Journal of Information Systems*, 25(4):289–302, 2016.

[22] M. Naveed, S. Kamara, and C. V. Wright. Inference attacks on property-preserving encrypted databases. In *Proc. CCS'15*, page 644, 2015.

[23] O. Ohrimenko, M. T. Goodrich, R. Tamassia, and E. Upfal. The melbourne shuffle: Improving oblivious storage in the cloud. In *Proc. ICALP*, page 556. 2014.

[24] O. Ohrimenko, F. Schuster, C. Fournet, A. Mehta, S. Nowozin, K. Vaswani, and M. Costa. Oblivious multi-party machine learning on trusted processors. In *Proc. USENIX Security'16*, pages 619–636, 2016.

[25] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *Proc. EUROCRYPT'99*, pages 223–238. Springer, 1999.

[26] S. Patel, G. Persiano, and K. Yeo. Cacheshuffle: An oblivious shuffle algorithm using caches. *arXiv preprint arXiv:1705.07069*, 2017.

[27] A. Peter, E. Tews, and S. Katzenbeisser. Efficiently outsourcing multiparty computation under multiple keys. *IEEE transactions on information forensics and security*, 8(12):2046–2058, 2013.

[28] A. Rosenthal, P. Mork, M. H. Li, J. Stanford, D. Koester, and P. Reynolds. Cloud computing: a new business paradigm for biomedical information sharing. *Journal of biomedical informatics*, 43(2):342–353, 2010.

[29] F. Shaon, M. Kantarcioglu, Z. Lin, and L. Khan. Sgx-bigmatrix: A practical encrypted data analytic framework with trusted processors. In *Proc. CCS'17*, pages 1211–1228. ACM, 2017.

[30] T. Skripcak, C. Belka, W. Bosch, C. Brink, T. Brunner, V. Budach, D. Büttner, J. Debus, A. Dekker, C. Grau, et al. Creating a data exchange strategy for radiotherapy research: towards federated databases and anonymised public datasets. *Radiotherapy and Oncology*, 113(3):303–309, 2014.

[31] J. Tang, Y. Cui, Q. Li, K. Ren, J. Liu, and R. Buyya. Ensuring security and privacy preservation for cloud data services. *ACM Computing Surveys (CSUR)*, 49(1):13, 2016.

[32] M. Tiehuis. libhcs. https://github.com/tiehuis/libhcs.

[33] D. Xie, G. Li, B. Yao, X. Wei, X. Xiao, Y. Gao, and M. Guo. Practical private shortest path computation based on oblivious storage. In *Proc. ICDE'16*, pages 361–372, 2016.