Shuffle Shamir Secret Shares Uniformly with Linear Online Communication

Jiacheng Gao, Yuan Zhang, and Sheng Zhong

Nanjing University, China

Abstract. In this paper, we revisit shuffle protocol for Shamir secret sharing. Upon examining previous works, we observe that existing constructions either produce non-uniform shuffle or require large communication and round complexity, e.g. exponential in the number of parties. We propose two shuffle protocols, both of which shuffle uniformly within $O(\frac{k+l}{\log k}n^2 m \log m)$ communication for shuffling rows of an $m \times l$ matrix shared among n parties, where $k \leq m$ is a parameter balancing communication and computation. Our first construction is more concretely efficient, while our second construction requires only $O(nml)$ online communication within $O(n)$ round. In terms of overall communication and online communication, both shuffle protocols outperform current optimal shuffle protocols for Shamir secret sharing.

At the core of our constructions is a novel permutation-sharing technique, which can be used to permute arbitrarily many vectors by computing matrix-vector products. Once shared, applying a permutation becomes much cheaper, which results in a faster online phase. Letting each party share one secret uniform permutation in the offline phase and applying them sequentially in the online phase, we obtain our first shuffle protocol. To further optimize online complexity and simplify the trade-off, we adopt the shuffle correlation proposed by Gao et al. and obtain the second shuffle protocol with $O(nml)$ online communication and $O(n^2ml)$ online computation. This brings an additional benefit that the online complexity is now independent of offline complexity, which reduces parameter optimization to optimizing offline efficiency.

Our constructions require only most basic primitives in Shamir secret sharing scheme, and work for arbitrary field $\mathbb{F}$ of size larger than n. As shuffle is a frequent operation in algorithm design, we expect them to accelerate many other primitives in context of Shamir secret sharing MPC, such as sorting, oblivious data structure, etc.

Keywords: Shuffle, Multiparty Computation, Shamir Secret Sharing

1 Introduction

Secure multiparty computation has found various applications in the real world. In an MPC scenario, several parties coordinate with each other to compute a function. The goal is to keep each party's input secret from all other parties while efficiently obtaining the output. This strong security guarantee makes MPC a desirable solution for many real-world cooperations with privacy concerns.

Most MPC protocols are built from primitives. That is, the designer of a new protocol uses existing MPC protocols as its subroutines, enabling a succinct algorithm description and a clear proof with the composition theorem [1]. Among all primitives, the MPC shuffle protocol is a frequently used one. An MPC shuffle protocol takes as input an array of secret shared values, randomly permutes it, and outputs another re-randomized secret shared array such that no party knows which entry of the output comes from which entry of the input. Such an operation has found direct applications in the real world, e.g. in constructing an anonymous communication system [2][3][4], electronic voting [5][6][7], distributed database [8][9], etc. Most of these applications can be viewed as a direct result of an MPC shuffle protocol.

Beyond such immediate applications, shuffle is also frequently used as a building block for more complicated MPC protocols. For examples, Hamada et al. [10] shows that any comparison-based MPC sorting algorithm can be securely and efficiently implemented with an MPC shuffle protocol plus MPC comparison protocol. The reason is that after randomly shuffling the array, it is safe to disclose comparison results, which is equivalent to revealing random bits. This is concluded by the "shuffle-then-sort" paradigm, which generates highly efficient comparison-based MPC sorting protocols. Another useful MPC primitive is the MPC oblivious random access memory (ORAM) proposed by Keller and Scholl [11]. The proposed MPC ORAM uses MPC shuffle to obscure the data structure, which prevents the parties from learning the access pattern. Building upon MPC ORAM, [11] further constructs more MPC algorithms, including oblivious array, priority queue, shortest path algorithm, etc.

Despite the vast application of MPC shuffle, recent researches concentrate on MPC shuffle protocol for additive secret sharing. This is largely due to a recent advance in two-party additive secret sharing shuffle protocol by Chase et al. [12], which builds a semi-honest two-party shuffle protocol with computation and communication complexity both almost linear. The protocol consists of mostly oblivious transfer extension (OTE) and local computation of pseudorandom generator, which makes it considerably efficient. This protocol is enhanced to active security and multi-party cases with linear online communication/computation in a sequential works of [4][13][14]. Hence for additive secret sharing schemes, shuffle operation is already quite efficient and cheap.

In terms of Shamir secret sharing MPC, current techniques for building shuffle protocol are still limited. Currently, the most efficient Shamir secret sharing shuffle protocol is based on switching network [3][15] and sorting network [16][17], with communication $O(nml \log m)$ for n parties shuffling m vectors, each of length l. However, the most severe drawback of such approaches is that they either do not shuffle uniformly [3][15] or only shuffle uniformly with certain probability [16][17], as their authors pointed out respectively. This makes the argument of security difficult for MPC protocols built upon such shuffle protocols. To shuffle uniformly, the only approach available is the permute-in-turn method [18][11], which results in higher communication and round complexity. The construction of [18] requires $O(2^n n^{1.5})$ rounds, which is efficient only when the number of par-

ties is small. The construction of [11] requires $O(n^2ml\log m)$ communication and $O(n\log m)$ rounds, which is also weaker in communication and round complexity compared to above non-uniform shuffle protocols. In addition, the protocol designed in [18] requires parties to compute zero-knowledge proof to be actively secure. These approaches are further discussed in Section 2, and see Table 1 for a conclusion of existing constructions.

In this paper, we develop shuffle protocols for Shamir secret sharing, with unconditional security against malicious parties under honest majority. We adopt the approach of permute-in-turn, which guarantees the uniformity of shuffle. We propose two constructions, both having $O(\frac{k+l}{\log k}n^2m\log m)$ overall communication complexity and $O(\frac{kl}{\log k}n^2m\log m)$ computation complexity. The first construction is better in overall concrete complexity, while the second requires only $O(nml)$ online communication, which is not achieved before for Shamir secret sharing. At the core of our constructions is a new primitive, which shares a permutation among parties with $O(\frac{k}{\log k}nm\log m)$ communication and computation, and allows the parties to perform it on any m-long vector with only $O(\frac{1}{\log k}nm\log m)$ communication and $O(\frac{k}{\log k}nm\log m)$ computation, for parameter $k \leq m$. As shared permutations can be reused, permuting rows of $m \times l$ matrix boils down to permuting each column vector separately. Following the permute-in-turn paradigm, we obtain a uniform shuffle protocol that requires $O(\frac{k}{\log k}n^2m\log m)$ offline communication, $O(\frac{l}{\log k}n^2m\log m)$ online communication and $O(\frac{n\log m}{\log n})$ online rounds. By further utilizing the shuffle correlation of Gao et al. [14], we obtain a uniform shuffle protocol with $O(\frac{k+l}{\log k}n^2m\log m)$ offline communication, $O(nml)$ online communication and $O(n)$ online rounds. We remark that in terms of overall communication complexity, this result outperforms all previous actively secure uniform shuffle protocols, including those for additive secret sharing, which has optimal communication complexity $O(\frac{Bl}{\log k}n^2m\log m)$ and computation complexity $O(\frac{Bkl}{\log k}n^2m\log m)$, for some B grows with security parameter [19]. All our constructions are based on only most basic primitives for Shamir secret sharing (mostly multiplications). This not only makes our constructions compatible with other Shamir secret sharing MPC protocols, but also offers a fairly simple and straightforward security analysis via the composition theorem [1].

Our contributions are summarized as follows.

1. We develop a novel permutation protocol for Shamir secret sharing, which involves new primitives for sharing a permutation and applying the shared permutation to vectors of proper size. Following the naive permute-in-turn paradigm, we construct a shuffle protocol with $O(\frac{k+l}{\log k}n^2m\log m)$ communication and $O(\frac{n\log m}{\log k}+\log\log k)$ rounds in total, with parameter k balancing between computation, communication and round complexity. In terms of overall communication and round complexity, this protocol outperforms all existing uniform shuffle protocols for Shamir secret sharing. In terms of online communication and round complexity, for small $n = o(\log m)$, choosing $k = \omega(2^n)$ enables $o(nml\log m)$ online communication and $o(\log m)$ online

round complexity, both of which outperform even switching-network-based and sorting-network-based approaches, at the cost of larger offline communication.
2. We propose the first shuffle protocol for Shamir secret sharing that has linear online communication complexity $O(nml)$ while always shuffles uniformly. While its overall communication complexity outperforms all existing actively secure uniform MPC shuffle protocols, its online communication complexity outperforms all existing shuffle protocols in context of Shamir secret sharing, including those non-uniform ones. The overall communication complexity of this protocol is also $O(\frac{k+l}{\log k} n^2 m \log m)$, except larger in constant.
3. Complexity analysis is done to support the claims regarding communication, computation and round complexity. This analysis not only validates our claims, but also reveals the relationship between parameter k and communication, computation and round complexity, which helps determine k in practice and optimize concrete efficiency.

The rest of the paper is organized as follows. In Section 2, we briefly review previous approaches towards constructing MPC shuffle protocol. As we are mostly concerned about Shamir secret sharing MPC, we focus our discussion to the shuffle protocols that can be implemented in Shamir secret sharing. In Section 3, we introduce the basic notations and models for security. Section 4 introduces our new permutation-sharing technique, and shows how to build a shuffle protocol with it. We show how to use the developed permutation protocol to build a shuffle protocol with linear online phase in Section 5. Besides its linear online phase, the shuffle protocol developed in Section 5 has same overall complexity as the one in Section 4, except with larger constant. Lastly in Section 6, we discuss some related topics regarding our constructions. Due to page limitation, we offer our security analysis in Appendix A.

2 Related Works

The concept of "multi-party shuffle" traces back to the seminal work of Chaum [20], as a byproduct of the concept of "decryption shuffle". In such a scenario, there are several servers and clients engaging the shuffle. A client wishes to send an anonymous message first applies an iterative encryption, with keys consist of all servers' public keys. Then it sends the ciphertext to the first server. Upon receiving a batch of ciphertexts from clients, each server decrypts the ciphertexts with its own private key, randomly permutes them, then sends them to the next server. After the decryption and permutation applied by the last server, all messages in this batch is securely shuffled, in the sense that no server can match a specific message with its sender. While the original work of Chaum [20] does not provide security against malicious server, there are sequential works enhancing the protocol to be actively secure [21][22][23][24]. The most common approach is utilizing zero-knowledge proof, where each server proves that it has honestly applied a decryption and a permutation. Due to proving the knowledge of a permutation matrix, this requires $O(m^2)$ computation complexity of each

server, where m is the number of messages to be shuffled. Also, due to the application of public key primitives and zero-knowledge proofs, these constructions are generally heavy in computation.

Despite the complexity issue, the decryption shuffle technique does not fit in multi-party computation (MPC) task. This is because after decryptions and permutations, the output is exactly shuffled plaintext. However, to perform further secure computation over the data in MPC, the output is also required to be secret shared. For example, in the work of [25] and [26], the authors build an MPC sort protocol from MPC shuffle protocol. It is crucial that the output of MPC shuffle protocol remains secretly shared, so that further tasks can proceed securely.

Very recently, the problem of MPC shuffle re-emerges, due to a current advance in two-party MPC shuffle. Chase et al. [12] propose the permutation decomposition technique that decomposes a large m-permutation into $O(\frac{m \log m}{k \log k})$ many small k-permutations. By implementing k-permutation protocol with obliviously punctured matrix (OPM), which includes $O(kl + k \log k)$ communication and $O(k^2 l)$ local computation, they construct a shuffle protocol with $O((1 + \frac{l}{\log k})m \log m)$ communication for two-party shuffle. In the lowest level, the protocol utilizes only OTE and PRG, which are computation-cheap and concretely efficient. The protocol is semi-honest secure, and is later enhanced to having linear online communication for semi-honest security by Eskandarian and Boneh [4], and to be malicious secure by Song et al. [19]. The most recent advance is due to Gao et al. [14], which achieves both malicious security and linear online communication.

However, all above advances in building MPC shuffle protocol happen in field of additive secret sharing, which are not directly applicable to Shamir secret sharing. This is due to several fundamental differences between the two secret sharing schemes. For example, Shamir secret sharing scheme demands "structured shares", i.e. valid shares must be points on some polynomial with bounded degree. This prevents using generic PRG to re-randomize a secret, which is a common approach in additive secret sharing to trade off computation for communication. Thus, when it comes to building shuffle protocol in Shamir secret sharing, the techniques are much more limited.

Currently, there are three approaches towards constructing MPC shuffle protocol in Shamir secret sharing. The first is based on switching networks, which is proposed by Lu et al. [3] and Mardi et al. [15]. The idea is to use switching network to perform shuffle, which consists of layers of switches. A switch is a gate that randomly swaps two entries of an array depending on an extra random bit, and can be implemented by MPC multiplication. The optimal communication complexity is achieved by Mardi et al. [15], which is $O(nml \log m)$. However, the major drawback of such a technique is that the produced permutation is not uniform, i.e. some permutations are significantly more likely to be produced than others, as is shown in Table 4 in [15]. This makes such approach weaker in security guarantee. The second approach is to base the shuffle protocol on sorting network, which is proposed by Movahedi et al. [16][17]. The idea is to utilize

Table 1. Existing MPC Shuffle Protocols for Shamir Secret Sharing

Protocol	Uniform	On Comm	On Round	Off Comm	Off Round	Approach
[3]	No	$O(nml\log^2 m)$	$O(\log^2 m)$	$O(\mathcal{B}m\log^2 m)$	$O(1)$	Switching Network
[15]	No	$O(nml\log m)$	$O(\log m)$	$O(\mathcal{B}m\log m)$	$O(1)$	Switching Network
[16][17]	w.p.[1]	$O(nml\log m)$	$O(\log m)$	$O(\mathcal{C}m\log m)$	$O(1)$	Sorting Network
[18]	Yes	$O(2^n n^{1.5} ml\log m)$	$O(2^n/\sqrt{n})$	$O(1)$	$O(1)$	Permute-in-turn
[11]	Yes	$O(n^2 ml\log m)$	$O(n\log m)$	$O(n^2 m\log m)$	$O(1)$	Permute-in-turn
Ours	Yes	$O(\frac{n^2 ml\log m}{\log k})$	$O(\frac{n\log m}{\log k})$	$O(\frac{kn^2 m\log m}{\log k})$	$O(\log\log k)$	Permute-in-turn
Ours[2]	Yes	$O(nml)$	$O(n)$	$O(\frac{(k+l)n^2 m\log m}{\log k})$	$O(\frac{\log m}{\log k}+\log\log k)$	Permute-in-turn

n is the number of parties, m the number of items to be shuffled, l the bit length of each item and k is a parameter balancing between communication, computation and round complexity.
$\mathcal{B}$ is the cost for generating one shared random bit, $\mathcal{C}$ the cost for preparing one MPC comparison. Both vary by concrete implementation.
"Off/On Comm" stands for "offline/online communication", "Round" for "round complexity", etc.
[15][16][17][11] are generic, i.e. can be applied to other MPC framework other than Shamir's. However, working in different framework might result in asymptotically different overheads.

a sorting network, which results in an MPC sorting protocol. By appending the data entry with a random tag and sorting them by tag, the array is randomly shuffled. The major drawback of this approach is that it requires an MPC comparison protocol, which further requires many complex and expensive MPC constructions. Another problem is that designing a sorting network (or equivalently, oblivious sort algorithm) is a highly complicated task, and current sorting networks are either too deep (e.g. Batcher's odd–even merge sort and bitonic sort [27], both of depth $O(\log^2 m)$), or with impractical constant (e.g. AKS network [28]), or only sort with some probability (e.g. LP network [29]). By utilizing the sorting network proposed in [29], Movahedi et al. [16][17] achieves $O(nml\log m)$ communication, with a uniform permutation with probability $1-O(m^{-3})$. Nevertheless, as the protocol would shuffle non-uniformly with only small (albeit non-negligible) probability, this protocol suffices for many applications.

The last approach is permute-in-turn, which is adopted by Laur et al. [18] and Keller and Scholl [11]. The idea of construction in [18] is to let many subsets of parties agree on uniform permutations, apply it to the array and re-share the array to all parties. Such a subset must be large enough, so that reconstruction of shared secret is possible within the subset of parties. Such subsets must also be many, so that the adversary cannot learn all permutations by infiltrating all subsets. It is concluded in [18] that approximately $O(2^n/\sqrt{n})$ many subsets are needed, and the communication complexity is hence $O(2^n n^{1.5} ml\log m)$. However, remarkably, by using switching network for performing permutation, Keller and Scholl [11] builds a shuffle protocol where each party permutes in turn. Their construction lets each party share $O(m\log m)$ control bits, which specify the permutation applied to the data passing through the switching network. By letting all parties "permute in turn", the underlying elements are uniformly shuffled. This significantly accelerates this approach, and makes the communication complexity $O(n^2 ml\log m)$ within $O(n\log m)$ rounds. The advantage of this approach is that it always shuffles uniformly, and the construction of [18] is extremely fast when the number of parties is small. The main drawback is that the com-

munication and round complexity are both larger than previously mentioned approaches. The round complexity of the construction of [18] is exponential in n, and the round complexity of [11] is larger than the approaches based on switching/sorting network by a factor of n.

All the above works are summarized in Table 1.

Besides all the pros and cons mentioned, all these approaches ignore the importance of online complexity. Many complicated MPC protocols are separated into two phases, a data-independent offline phase and a data-dependent online phase. This separation helps shift most of the workload to offline phase and leave the online phase with fairly small overhead. This allows the parties to respond swiftly to real-world event that causes the data. However, almost all above protocols have trivial offline phase, which is merely a preparation phase for most fundamental random resources, i.e. shared random values and bits. For instances, the shuffle protocols based on switching network must wait until all data arrive, and then feeds the data into the network. Before that, only random bits can be prepared. The ones based on MPC sort protocol must also wait, or otherwise it has nothing to sort. And although the construction of Keller and Scholl [11] can let the parties share and verify control bits in offline phase, the online phase still consists of $O(n^2ml\log m)$ communication, which does not benefit much from the offline phase. This makes their online phase heavy, which could become bottleneck for MPC tasks built upon them.

In this paper, we build MPC shuffle protocols for Shamir secret sharing scheme that always shuffle uniformly. We first develop a permutation sharing technique, which offers a non-trivial trade-off between the communication overhead of sharing a permutation and of applying a permutation. Building upon such a technique, our first shuffle protocol requires $O(\frac{k}{\log k}n^2m\log m)$ offline communication and $O(\frac{l}{\log k}n^2m\log m)$ online communication. Remarkably, for small $n = o(\log m)$, choosing $k = \omega(2^n)$ enables an $o(nml\log m)$ online communication complexity that is even smaller than switching-network-based and sorting-network-based ones, at the cost of larger offline communication. Our second construction has $O(nml)$ online communication, which is currently optimal in context of Shamir secret sharing shuffle. This is achieved by applying the shuffle correlation technique of Gao et al. [14], at the cost of slightly increasing concrete complexity.

3 Preliminary

3.1 Basic Notation

In this paper, we consider an MPC scenario where n parties $P_1, P_2, ..., P_n$ are engaging computation. Let $\mathbb{F}$ be a field with $|\mathbb{F}| > n$. Denote the prime subfield

[1] According to [16], this probability is $1 - O(m^{-3})$.

[2] When $n \geq \kappa$, the offline communication can be $O(\frac{k+l}{\log k}\kappa nm\log m)$, online communication $O(\kappa ml)$ and online round complexity $O(\kappa)$, for statistical security parameter κ. Roughly, this is done by randomly select κ parties and require them to choose the permutations.

of $\mathbb{F}$ as $\mathbb{F}_p$. Let $\mathbb{K}$ be an extension of $\mathbb{F}$, such that $|\mathbb{K}| \geq 2^\kappa$ for statistical security parameter κ. Let $\{\alpha_i\}_{i=1}^n$ be n distinct non-zero elements in $\mathbb{F}$, which are public and known to all parties. Let $t < n/2$ be the number of corrupted parties.

We use bold font to indicate that a variable is vector, e.g. $\mathbf{x}, \mathbf{y}, \mathbf{z}$. Vectors are column vectors unless stated otherwise. We refer to the i-th entry of vector $\mathbf{x}$ by x_i or $\mathbf{x}(i)$. The latter is useful when the vector has its own subscript, e.g. vector $\mathbf{x}_1, \mathbf{x}_2$, etc. We use bold uppercase to indicate that a variable is a matrix, e.g. $\mathbf{A}, \mathbf{B}, \mathbf{C}$. Similar to the case of vector, we denote by the element on i-th row and j-th column of matrix $\mathbf{A}$ by $a_{i,j}$ or $\mathbf{A}(i, j)$. Suppose a vector $\mathbf{x} = (x_i)_{i\in[m]}$ is of length m. Then for any m-permutation $\pi : [m] \to [m]$, the result of applying it on vector $\mathbf{x}$ is defined as

$$\pi(\mathbf{x}) \coloneqq (x_{\pi(1)}, x_{\pi(2)}, ..., x_{\pi(m)}).$$

We define the permutation matrix of permutation π to be $\mathbf{P}_\pi \coloneqq (r_{i,j})_{i,j\in[m]}$, where

$$p_{i,j} = \begin{cases} 1 & \text{if } j = \pi(i), \\ 0 & \text{otherwise.} \end{cases}$$

Note that i is the index of row, and j of column, and

$$\pi(\mathbf{x}) = \mathbf{P}_\pi \cdot \mathbf{x}.$$

Define $[k] = \{1, 2, ..., k\}$ for any positive integer k. Denote by notation "$[\![s]\!]$" sharing s by Shamir secret sharing scheme among all parties. This means that there exists a polynomial $f \in \mathbb{F}[X]$ with $f(0) = s$ of degree t (the number of corrupted parties), such that $f(\alpha_i)$ is known only to P_i. When saying a vector or matrix is secret shared, e.g. $[\![\mathbf{x}]\!]$ or $[\![\mathbf{A}]\!]$, we mean each entry of it is secret shared. For notation simplicity, we denote a shared vector by

$$[\![\mathbf{v}]\!] = ([\![v_1]\!], [\![v_2]\!], ..., [\![v_m]\!])^\top = [\![v_1, v_2, ..., v_m]\!]^\top,$$

and same for matrix.

When calling a sub-protocol, three types of parameters may be involved. For example, consider the following:

$$\mathrm{Prot}(a, [\![b]\!], P_i : c).$$

This means to call protocol "Prot", with parameter a a public constant, b of secret shared form, and c a private input from P_i.

We assume that the items to be shuffled is m (field) elements/vectors, and assume m is power of two. The main reason for the assumption is to apply a technique introduced by Chase et al. [12], which uses Beneš network [30] for decomposing permutation. Originally, Beneš network is designed only for permutation of a power-of-two size. Since shuffling m-sized array can be easily used to shuffling array of smaller size $m' \leq m$, assuming m is power of two is without loss of generality.

3.2 Security Model

In the paper, we consider a malicious adversary that can corrupt up to $t < n/2$ parties. We assume secure pair-wise channels and a secure broadcast channel. The communication cost for broadcasting is assumed to be $O(n)$. For simplicity, we assume $\log|\mathbb{K}|$ is $O(1)$, hence broadcasting a field element in $\mathbb{F}$ or $\mathbb{K}$ both require $O(n)$ communication.

Our goal is to construct shuffle protocols that are secure-with-abort with unconditional security against static adversary, under honest majority. The adversary is static, meaning that it must choose and fix the parties it wants to corrupt before the start of the protocol. The security is unconditional, meaning that the security of our construction does not depend on computational hardness assumption, i.e. the adversary can be computationally unbounded. That is, as long as the number of corrupted parties is smaller than $n/2$, our construction remains statistically secure, where the honest parties will abort with overwhelming probability $p > 1 - C \cdot 2^{-\kappa}$ if the corrupted parties deviate from the protocol, where C is the number of MPC operations (a.k.a. size of the computation circuit). It is extensively studied and long been established that such a strong security guarantee can (only) be achieved in honest majority.

3.3 Primitives for Shamir Secret Sharing Scheme

Our constructions require only a very high-level view of Shamir secret sharing. Hence, we formalize our constructions with the ideal functionalities of Shamir secret sharing.

We assume an ideal MPC functionality $\mathcal{F}_{\text{Shamir}}$ that supports following commands:

- $[\![s]\!] \leftarrow \Pi_{\text{input}}(P_i : s)$, which shares a secret from P_i to all parties.
- $s \leftarrow \Pi_{\text{open}}([\![s]\!])$, which publicly opens a secret s to all parties.
- $s \leftarrow \Pi_{\text{open}}([\![s]\!], P_i)$, which opens s to party P_i.
- $[\![s]\!] \leftarrow \Pi_{\text{add}}([\![a]\!], [\![b]\!])$, which returns $s = a + b$. This protocol can be done without communication.
- $[\![s]\!] \leftarrow \Pi_{\text{mul}}([\![a]\!], [\![b]\!])$, which returns $s = ab$ (with re-randomization).
- $[\![s]\!] \leftarrow \Pi_{\text{inner}}([\![\mathbf{a}]\!], [\![\mathbf{b}]\!])$, which return inner product $s = \mathbf{a}^\top \cdot \mathbf{b}$ (with re-randomization).
- $[\![r]\!] \leftarrow \Pi_{\text{rand}}(\mathbb{J})$, which draws uniformly $r \xleftarrow{\$} \mathbb{J}$ where $\mathbb{J}$ is either $\mathbb{F}_p$, $\mathbb{F}$ or $\mathbb{K}$.
- $\lambda \leftarrow \Pi_{\text{challenge}}(\mathbb{K})$, which draws public uniform λ over designated field $\mathbb{K}$.

We assume that, the above functionalities are ideal, in the sense that any misbehavior leads to immediate abort. Note that this is different from real-world protocol, where honest parties will detect misbehavior only after certain correctness check.

Note that Π_{input}, Π_{open}, Π_{mul}, Π_{rand} and $\Pi_{\text{challenge}}$ can be implemented in $O(n)$ communication and $O(n)$ computation. Note also that Π_{inner}, when called with two vectors of length m, requires $O(n)$ communication and $O(nm)$ computation. Moreover, we assume Π_{add} can be done without communication.

A concrete construction supporting above primitives can be found in [31]. Note that in practice, current constructions for $\mathcal{F}_{\text{Shamir}}$ require batched processing for achieving amortized linear communication. As our construction does not require immediate and separate correctness check for multiplications, the amortized complexity can be faithfully achieved. Also note that due to efficiency consideration, concrete implementation of $\Pi_{\text{challenge}}$ produces only a distribution "close to" uniform, which costs $O(n)$ communication (see [31]). For the simplicity of analysis, we omit this issue and simply assume that the output is uniform. This will not harm the security of our protocol.

4 Permutation Protocol and Shuffle Protocol

In this section, we describe the construction of our first shuffle protocol. This protocol is constructed by first developing a protocol that allows a single party P_w to share a secret permutation π, such that parties can apply permutation π to any vector of compatible size. Once such a permutation sharing protocol is available, the shuffle protocol follows directly from permute-in-turn paradigm. That is, in the offline phase, each party shares a secret uniform m-permutation. In the online phase, all parties perform all the m-permutations sequentially. The uniformity of shuffle follows by the fact that as long as one of the permutation remains unknown and uniform, the produced permutation is uniform.

To implement such a permutation-sharing scheme, there are two straightforward approaches:

1. Share an $m \times m$ permutation matrix among parties. Sharing such a permutation requires $O(nm^2)$ communication, and applying it requires $O(nm)$ communication, $O(nm^2)$ computation and $O(1)$ rounds.
2. Share $m \log m$ bits among parties, each responds to a control bit in the permutation network [11]. Sharing a permutation hence requires $O(nm \log m)$ communication, and applying it requires $O(nm \log m)$ communication/computation and $O(\log m)$ rounds.

The first approach offers optimal communication overhead for applying the permutation, while the second offers optimal overall communication complexity. The main drawback of the first approach is its $O(m^2)$ communication/computation, which is unrealistic for large m. The main drawback of the second approach is its $O(\log m)$ rounds, and that applying a permutation is no cheaper than sharing it, i.e. the protocol does not benefit much from an offline phase.

It turns out that the above two approaches can be viewed as two extreme cases of a more sophisticated construction of ours. Roughly, it turns out that we can represent an m-permutation by $\frac{m \log m}{k \log k}$ many $k \times k$ permutation matrices. Applying a shared permutation is thus replaced by multiplying a bunch of $k \times k$ matrices with a bunch of k-long vectors, in appropriate order. To this end, we require a protocol for sharing (well-formed) permutation matrix. However, checking if an arbitrarily shared matrix is well-formed permutation matrix is a non-trivial task (see discussion in Section 6.1). To overcome this, we first let

party share k one-hot vectors. By checking if the k one-hot vectors are distinct from each other, the parties are able to check the well-formedness of permutation matrix.

Hence in the following, we first present our definition of shared permutation, and how parties could use it to perform permutation. We then give our construction for sharing one-hot vectors, which is then used to generate shared permutation matrix. After presenting the construction of our shuffle protocol, we analyze its complexity at the end of this section.

4.1 Shared Permutation

Definition 1 (k-Shared m-Permutation). *We say an m-permutation π is k-shared as $[\![\pi]\!]_k$, if*

$$[\![\pi]\!]_k := (k, s, \{\{[\![\mathbf{P}_{\pi'_{i,j}}]\!], T_{i,j}\}_{j\in[m/k]}\}_{i\in[s]}),$$

where

1. *Each $\pi'_{i,j}$ is a k-permutation.*
2. *$s = O(\frac{\log m}{\log k})$. This requirement is tight, c.f. Section 6.3.*
3. *Permutation matrix $\mathbf{P}_{\pi'_{i,j}}$ is secret shared among parties.*
4. *$T_{i,j} \subseteq [m]$ of size k, which is denoted as "task" of $\pi'_{i,j}$. For same i, $T_{i,j}$ are disjoint.*
5. *$\pi = \pi_{s,1} \circ \pi_{s,2} \circ \cdots \pi_{s,m/k} \circ \pi_{s-1,1} \circ \cdots \circ \pi_{1,m/k}$, where $\pi_{i,j}$ is defined as the m-permutation acquired by letting $\pi'_{i,j}$ acting on entries specified by $T_{i,j}$.*

For simplicity, we require in addition that m and k are power-of-two. When k is not important in context, we simply write a shared π as $[\![\pi]\!]$.

Once $[\![\pi]\!]$ is shared, parties can apply it to arbitrarily many m-long vectors. This is done by properly and iteratively replacing the entries of the vector by the matrix-vector product between the shared permutation matrices and the vector. As an example, when $k = m$, the entire shared permutation is simply an $m \times m$ permutation matrix, and multiplying it by the vector results immediately the permuted vector. This process is formalized in Algorithm 1.

4.2 Sharing One-hot Vector

A one-hot vector is a vector that has exactly one non-zero entry, whose value is 1. Define the "index" of a one-hot vector to be the position where 1 locates. Sharing one-hot vector serves as the first step towards sharing permutation matrix, as a $k \times k$ permutation matrix consists of k many one-hot vectors with distinct indices.

Suppose party P_w needs to share a one-hot vector of length k with index ind, for $k = 2^d$. It first decomposes the index into $\log k = d$ many bits $b_1, b_2, ..., b_d$, i.e.

$$\text{ind} = \sum_{i=1}^{d} 2^{i-1} b_i.$$

Algorithm 1 $[\![\pi(\mathbf{v})]\!] \leftarrow \text{Permute}([\![\pi]\!], [\![\mathbf{v}]\!])$

Require: π is a shared m-permutation and $\mathbf{v}$ is of length m.
Ensure: Return $[\![\pi(\mathbf{v})]\!]$.
  Parse $[\![\pi]\!]$ as $(k, s, \{\{[\![\mathbf{P}_{\pi'_{i,j}}]\!], T_{i,j}\}_{j\in[m/k]}\}_{i\in[s]})$
  for $i = 1$ to s **do**
    for $j = 1$ to m/k **do parallel**
      $[\![\mathbf{u}]\!] \leftarrow [\![\mathbf{v}(l)]\!]_{l\in T_{i,j}}$
      $[\![\mathbf{u}']\!] \leftarrow [\![\mathbf{P}_{\pi'_{i,j}}]\!] \cdot [\![\mathbf{u}]\!]$ ▷ k calls to Π_{inner}.
      $[\![\mathbf{v}(l)]\!] \leftarrow [\![\mathbf{u}'(l)]\!]$ for each index $l \in T_{i,j}$
    end for
  end for
  Return $[\![\mathbf{v}]\!]$.

Then it shares $[\![b_i]\!]$ to all parties. The parties can check if all shared values are Boolean by generating a challenge $\lambda \in \mathbb{K}$ and checking that

$$0 = \sum_{i=1}^{d} \lambda^{i-1} [\![b_i]\!][\![1 - b_i]\!].$$

Note that this is evaluating a polynomial $F \in \mathbb{K}[X]$ of degree at most $d-1$ at a random point $\lambda \in \mathbb{K}$. As a $(d-1)$-degree polynomial has at most $d-1$ roots in field, we have

$$\Pr[F(\lambda) \neq 0 \mid \exists i \text{ s.t. } b_i(1-b_i) \neq 0] \geq 1 - \frac{d-1}{|\mathbb{K}|} \geq 1 - \frac{d-1}{2^\kappa}.$$

This Boolean checking protocol is formally described in Algorithm 2.

Algorithm 2 $\text{BoolCheck}([\![b_1]\!], ..., [\![b_d]\!])$

Ensure: Abort if any b_i is not in $\{0, 1\}$.
  $\lambda \leftarrow \Pi_{\text{challenge}}(\mathbb{K})$
  $[\![u]\!] \leftarrow \sum_{i=1}^{d} \lambda^{i-1} [\![b_i]\!][\![1 - b_i]\!]$ ▷ One call to Π_{inner}.
  $u \leftarrow \Pi_{\text{open}}([\![u]\!])$
  All parties abort if $u \neq 0$.

If the check passes, parties convert the shared bits into a one-hot vector. This operation of unfolding $\log k$ bits into k-long one-hot vector is known as "demultiplex". In MPC, this can be done by the demux protocol proposed by Launchbury et al. [32], which computes the desired one-hot vector with $O(k)$ MPC multiplication within $O(\log \log k)$ rounds. This is done by first separating the bits into two parts and recursively generating corresponding one-hot vector for each part. By computing the tensor product of two returned one-hot vector, the parties obtain the one-hot vector with correct index. This protocol is formally presented in Algorithm 3.

Algorithm 3 $(\llbracket s_1 \rrbracket, ..., \llbracket s_{2^d} \rrbracket) \leftarrow \text{Demux}(\llbracket b_1 \rrbracket, ..., \llbracket b_d \rrbracket)$

Ensure: Output a one-hot vector with index $\sum_{i=1}^{d} 2^{i-1} b_i + 1$.
  if $d = 1$ **then**
    Return $(1 - \llbracket b_1 \rrbracket, \llbracket b_1 \rrbracket)$
  end if
  $t \leftarrow \lfloor d/2 \rfloor$
  $\llbracket u_1 \rrbracket, \llbracket u_2 \rrbracket, ..., \llbracket u_{2^t} \rrbracket \leftarrow \text{Demux}(\llbracket b_1 \rrbracket, ..., \llbracket b_t \rrbracket)$
  $\llbracket v_1 \rrbracket, \llbracket v_2 \rrbracket, ..., \llbracket v_{2^{d-t}} \rrbracket \leftarrow \text{Demux}(\llbracket b_{t+1} \rrbracket, ..., \llbracket b_d \rrbracket)$ ▷ Parallel recursive calls.
  for $i = 1$ to 2^t **do parallel**
    for $j = 1$ to 2^{d-t} **do parallel**
      $\llbracket s_{i+2^t(j-1)} \rrbracket \leftarrow \Pi_{\text{mul}}(\llbracket u_i \rrbracket, \llbracket v_j \rrbracket)$
    end for
  end for
  Return $\llbracket s_1, ..., s_{2^d} \rrbracket$

Algorithm 4 gives a formal description of our OneHot protocol, which helps party P_w share a one-hot vector. Note that parties can check all shared bits in one call to BoolCheck, instead of checking separately for each one-hot vector.

Algorithm 4 $(\llbracket s_1 \rrbracket, ..., \llbracket s_k \rrbracket) \leftarrow \text{OneHot}(k, P_w : \text{ind})$

Require: $k = 2^d$ is a power of two.
Ensure: Output a one-hot vector, with index ind.
  P_w computes bits $b_1, ..., b_d$, s.t. ind $= \sum_{i=1}^{d} 2^{i-1} b_i + 1$.
  P_w shares $\llbracket b_1 \rrbracket$, ..., $\llbracket b_d \rrbracket$.
  BoolCheck$(\llbracket b_1 \rrbracket, ..., \llbracket b_d \rrbracket)$
  Return Demux$(\llbracket b_1 \rrbracket, ..., \llbracket b_d \rrbracket)$.

4.3 Sharing Permutation Matrix

Note that a $k \times k$ permutation matrix is k many k-long one-hot vectors with distinct indices. By calling k times OneHot protocol and viewing the results as row vectors, the parties obtain a shared matrix $\mathbf{P} \in \{0,1\}^{k \times k}$, such that each row contains exactly one 1. However, a corrupted P_w could make $\mathbf{P}$ ill-formed by putting more than one 1 on same column. To prevent this, parties need to check if the shared matrix is well-formed. As there are in total k many 1 in the matrix and 0 elsewhere, it suffices to check that sum along column is never 0, which is equivalent to verifying that all sums along column are 1.

To this end, a naive approach is to open the sums along each column, and check if all of them are 1. To batch the check, the parties can first generate a challenge $\lambda \leftarrow \Pi_{\text{challenge}}(\mathbb{K})$, and check if

$$\sum_{i=1}^{k} \lambda^{i-1} \stackrel{?}{=} \sum_{i=1}^{k} \left(\lambda^{i-1} \left(\sum_{j=1}^{k} \mathbf{P}(j,i) \right) \right).$$

Since a non-zero polynomial of degree $k-1$ will have at most $k-1$ roots in field, the equality will not hold with probability $1-\frac{k-1}{|\mathbb{K}|} \geq 1-\frac{k-1}{2^\kappa}$ if $\mathbf{P}$ is not a permutation matrix.

The protocols are formally presented in Algorithm 5 and Algorithm 6. Note that the entire process of Algorithm 6 is a single call to protocol $\Pi_{\text{challenge}}$ plus Π_{open}. Also, parties can batch all permutation checks into one single check, instead of checking each matrix separately, which could improve concrete communication complexity.

Algorithm 5 $[\![\mathbf{P}]\!] \leftarrow \text{PermMat}(P_w : \pi)$

Require: π is a k-permutation, where k is power of two.
Ensure: Output a permutation matrix $\mathbf{P}$.
  for $i = 1$ to k **do parallel**
    $\mathbf{v}_i \leftarrow \text{OneHot}(k, P_w : \pi(i))$
  end for
  $[\![\mathbf{P}]\!] \coloneqq [\![v_{i,j}]\!]_{i,j\in[k]}$
  PermCheck($[\![\mathbf{P}]\!]$)
  Return $[\![\mathbf{P}]\!]$.

Algorithm 6 PermCheck($[\![\mathbf{P}]\!]$)

Require: $\mathbf{P}$ is of size $k \times k$, with each row a one-hot vector.
Ensure: Abort with high probability if $\mathbf{P}$ is not a permutation matrix.
  $[\![\text{sum}]\!] \leftarrow [\![0]\!]$
  $\lambda \leftarrow \Pi_{\text{challenge}}(\mathbb{K})$
  for $j = 1$ to k **do** ▷ Enumerate columns.
    $[\![\text{sum}]\!] \leftarrow [\![\text{sum}]\!] + \lambda^{j-1}(1 - \sum_{i=1}^{k} [\![\mathbf{P}(i,j)]\!])$
  end for
  $\text{sum} \leftarrow \Pi_{\text{open}}([\![\text{sum}]\!])$
  Abort if $\text{sum} \neq 0$.

4.4 Permutation Decomposition and Sharing Permutation

The Beneš network dates back to the seminal work of Beneš [30]. In short, it is a switching network capable of representing any m-permutation within $O(\log m)$ depth. By partitioning the network by layers and taking out intermediate permutations, one naturally obtains a reformulation of original permutation. Chase et al. [12] first introduce the permutation decomposition technique based on such reformulation, which decomposes an m-permutation into $O(\frac{m \log m}{k \log k})$ many smaller permutations, each of size k. Original Beneš network works only for m being power of two. Though there are works extending it to work with arbitrary input size, some important properties are lost, e.g. the independency of

decomposed small permutations, which is crucial for the round complexity to be $O(\log m)$.

To formalize this process, denote by

$$(\pi_1, \pi_2, ..., \pi_s) \leftarrow \text{Decompose}(\pi, k)$$

the process of decomposing π into

$$\pi = \pi_s \circ \pi_{s-1} \circ \cdots \circ \pi_1,$$

with each π_i an m-permutation. The concrete value is $s = (2\log m - 1)/\log k$ according to [12], which is $O(\frac{\log m}{\log k})$. Each π_i can be further decomposed as $\{\pi_{i,j}\}_{j\in[m/k]}$, each of which is an m-permutation that touches only k entries of the vector. Hence,

$$\pi = \pi_{s,1} \circ \pi_{s,2} \circ \cdots \pi_{s,m/k} \circ \pi_{s-1,1} \circ \cdots \circ \pi_{1,m/k}.$$

Define the "task" of each $\pi_{i,j}$ as

$$\text{Task}(m, k, i, j) = \{x \in [m] \mid \pi_{i,j} \text{ affects the } x\text{-th entry}\}.$$

Denote by $\pi'_{i,j}$ the "compressed" permutation of $\pi_{i,j}$, in the sense that it is now a k-permutation acting on its task. The decomposition guarantees that each task has size k.

We note that the decomposition enjoys task-invariant property, which means that the task of $\pi_{i,j}$ is independent of π. This allows all parties to agree on the tasks (i.e. which of the entries shall be permuted) even without knowing the permutation being currently applied.

Example 1. As a concrete example, consider decomposing a 8-permutation π, with $k = 4$. Suppose we have decomposed π and obtained π_1 as

$$\pi_1 = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 5 & 4 & 3 & 6 & 7 & 8 & 1 & 2 \end{pmatrix}.$$

It can further be decomposed into two disjoint permutations, i.e.

$$\pi_{1,1} = \begin{pmatrix} 1 & 3 & 5 & 7 \\ 5 & 3 & 7 & 1 \end{pmatrix}, \ \pi_{1,2} = \begin{pmatrix} 2 & 4 & 6 & 8 \\ 4 & 6 & 8 & 2 \end{pmatrix}.$$

Note that $\pi_{1,1}$ and $\pi_{1,2}$ are "disjoint" 8-permutations, in the sense that they are essentially two smaller permutations acting on disjoint fixed subsets (i.e. their tasks) of $\{1, ..., 8\}$. The corresponding tasks are

$$\text{Task}(8, 4, 1, 1) = \{1, 3, 5, 7\},$$
$$\text{Task}(8, 4, 1, 2) = \{2, 4, 6, 8\}.$$

Note that $\pi_{1,1}$ may vary by π, but its task will not, due to the task-invariant property of the decomposition. Hence, we may compress them into $\pi'_{1,1}$ and $\pi'_{1,2}$, where

$$\pi'_{1,1} = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 3 & 2 & 4 & 1 \end{pmatrix}, \ \pi'_{1,2} = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 3 & 4 & 1 \end{pmatrix}$$

We have thus represented a 8-permutation π_1 by two 4-permutations $\pi'_{1,1}$ and $\pi'_{1,2}$, each acting on disjoint entries.

Our construction for permutation sharing protocol is formally presented in Algorithm 7. Roughly, all parties firstly agree on the parameter k. Each party then obtains a sequence of task by locally computing $\{\text{Task}(m, k, i, j)\}$, and party P_w who chooses the permutation π locally decomposes π into a sequence of small permutation $\{\pi'_{i,j}\}$. Then P_w shares all $\pi'_{i,j}$ as permutation matrices. Since $\pi_{i,j}$ are disjoint for same i, this satisfies our definition for k-shared m-permutation.

Algorithm 7 $[\![\pi]\!]_k \leftarrow \text{SharePerm}(k, P_w : \pi)$

Require: π is m-permutation.
Ensure: Return k-shared m-permutation $[\![\pi]\!]_k$.
  Party P_w let $(\pi_1, ..., \pi_s) \leftarrow \text{Decompose}(\pi, k)$.
  Party P_w further decomposes each π_i into $\pi_{i,1}, ..., \pi_{i,m/k}$.
  for $i = 1$ to s **do parallel**
    for $j = 1$ to $[m/k]$ **do parallel**
      $T_{i,j} \leftarrow \text{Task}(m, k, i, j)$
      $[\![\mathbf{P}_{\pi'_{i,j}}]\!] \leftarrow \text{PermMat}(P_w : \pi'_{i,j})$
    end for
  end for
  Return $[\![\pi]\!]_k \coloneqq (k, s, \{\{[\![\mathbf{P}_{\pi'_{i,j}}]\!], T_{i,j}\}_{j\in[m/k]}\}_{i\in[s]})$.

4.5 Shuffle Protocol

Our first shuffle protocol consists of exactly n calls to the permutation protocol developed above. The two phases of the protocol are formally presented in Algorithm 8 and Algorithm 9. In the offline phase, parties prepare n shared permutations, one from each party. In the online phase, parties apply these permutations to the incoming data by protocol Permute.

Algorithm 8 $\{[\![\pi_i]\!]_k\}_{i\in[n]} \leftarrow \text{Shuffle1}_{\text{Off}}(k, m)$

Require: Both m and k are power of two.
  for $w = 1$ to n **do**
    Party P_w draws m-permutation π_w uniformly.
    $\text{SharePerm}(k, P_w : \pi_w)$
  end for
  Return $\{[\![\pi_i]\!]_k\}_{i\in[n]}$.

Algorithm 9 $[\![\mathbf{v}'_1, \mathbf{v}'_2, ..., \mathbf{v}'_l]\!] \leftarrow \text{Shuffle1}_{\text{On}}([\![\mathbf{v}_1, \mathbf{v}_2, ..., \mathbf{v}_l]\!])$

Require: Each $\mathbf{v}_i$ is of length m, which is a power of two.
Ensure: Return shuffled $[\![\pi(\mathbf{v}_i)]\!]_{i\in[l]}$, with secret uniform π.
  Fetch fresh $\{[\![\pi_i]\!]\}_{i\in[n]}$ generated by $\text{Shuffle1}_{\text{off}}$.
  for $w = 1$ to n **do**
    for $i = 1$ to l **do parallel**
      $[\![\mathbf{v}_i]\!] \leftarrow \text{Permute}([\![\pi_w]\!], [\![\mathbf{v}_i]\!])$
    end for
  end for
  Return $[\![\mathbf{v}'_1, \mathbf{v}'_2, ..., \mathbf{v}'_l]\!]$.

4.6 Complexity Analysis

Below we offer a detailed analysis of the complexity of above protocols. We discuss the offline and online complexity separately when necessary, and omit this separation for simple protocols. The complexities for shuffle protocols are concluded in Corollary 1.

Theorem 1. *For protocol* $\text{Permute}([\![\pi]\!]_k, [\![\mathbf{v}]\!])$ *described in Algorithm 1, the communication, computation and round complexity are* $O(\frac{1}{\log k} nm \log m)$, $O(\frac{k}{\log k} nm \log m)$ *and* $O(\frac{\log m}{\log k})$, *respectively.*

Proof: The parties are computing $O(\frac{m \log m}{k \log k})$ many matrix-vector products, each multiplying a $k \times k$ permutation matrix by k-long data vector. Such a multiplication is simply k calls to Π_{inner}, of each the communication complexity is only $O(nk)$. The communication complexity is hence $O(\frac{1}{\log k} nm \log m)$. The computation complexity of a single matrix-vector product is $O(nk^2)$, and the computation complexity is hence $O(\frac{k}{\log k} nm \log m)$.

For round complexity, note that the parties can parallelize all small permutation $\pi'_{i,j}$ with same i. The entire i-th iteration is thus done within $O(1)$ rounds, and the round complexity is simply $O(s) = O(\frac{\log m}{\log k})$.

Lemma 1. *For protocol* $\text{BoolCheck}([\![b_1, ..., b_d]\!])$ *described in Algorithm 2, the communication, computation and round complexity are* $O(n)$, $O(nd)$ *and* $O(1)$, *respectively.*

Proof: The only communication happens at one call to $\Pi_{\text{challenge}}$, Π_{open} and Π_{inner}, all within $O(n)$ communication and $O(1)$ rounds.

The computation bottleneck is computing

$$[\![u]\!] \leftarrow \sum_{i=1}^{d} \lambda^{i-1} [\![b_i]\!] [\![1 - b_i]\!],$$

which is $O(nd)$ computation in total.

Lemma 2. *For protocol* Demux($[\![b_1, ..., b_d]\!]$) *described in Algorithm 3, the communication, computation and round complexity are* $O(n2^d)$, $O(n2^d)$ *and* $O(\log d)$, *respectively.*

Proof: Denote by $M(d)$ the number of calls to primitive Π_{mul}. Then

$$M(d) = 2 \times M(d/2) + 2^d.$$

Since for all $d \geq 4$,

$$2^{d+1} \geq 2 \times 2^{d/2+1} + 2^d = 2^{d/2+2} + 2^d,$$

we conclude that $M(d) = O(2^{d+1}) = O(2^d)$.

Recall that in protocol Demux, only calls to Π_{mul} require communication. Hence, the communication complexity is $O(n2^d)$. The computation complexity follows similarly.

Denote by $R(d)$ the number of rounds. Note that

$$R(d) = R(d/2) + O(1).$$

This is because the two calls to Demux can be parallelized, and the remainder is parallel calls to Π_{mul}. Hence, the round complexity is $O(\log d)$.

Lemma 3. *For protocol* OneHot(k, P_w : ind) *described in Algorithm 4, the communication, computation and round complexity are* $O(nk)$, $O(nk)$ *and* $O(\log \log k)$, *respectively.*

Proof: The bottleneck is the call to Demux($[\![b_1, ..., b_d]\!]$), for $d = \log k$. According to Lemma 2, its communication, computation and round complexity are $O(n2^d)$, $O(n2^d)$ and $O(\log d)$, respectively. As $d = \log k$, the theorem follows trivially.

Lemma 4. *For protocol* PermCheck($[\![\mathbf{P}]\!]$) *described in Algorithm 6, the communication, computation and round complexity are* $O(n)$, $O(nk^2)$ *and* $O(1)$, *respectively.*

Proof: The communication happens only at $\Pi_{\text{challenge}}$ and Π_{open}, each costs $O(n)$ communication and $O(1)$ rounds.

The most computation-heavy step in the algorithm is each party locally computing the sum, which costs $O(nk^2)$ in total.

Lemma 5. *For protocol* PermMat($P_w : \pi$) *described in Algorithm 5, the communication, computation and round complexity are* $O(nk^2)$, $O(nk^2)$ *and* $O(\log \log k)$, *respectively.*

Proof: The protocol consists of k parallel calls to protocol OneHot($k, P_w : \pi(i)$) plus one call to PermCheck($\mathbf{P}$). By Lemma 3 and Lemma 4, this costs $O(nk^2)$ communication, $O(nk^2)$ computation and $O(\log \log k)$ rounds.

Theorem 2. *For protocol* $\text{SharePerm}(k, P_w : \pi)$ *given in Algorithm 7, the communication, computation and round complexity are* $O(\frac{k}{\log k} nm \log m)$, $O(\frac{k}{\log k} nm \log m)$ *and* $O(\log \log k)$, *respectively.*

Proof: Recall that the permutation decomposition of [12] decomposes an m-permutation π into $s = O(\frac{\log m}{\log k})$ many m-permutations $\pi_1, ..., \pi_s$. Each permutation π_i is then decomposed into m/k many permutations $\pi_{i,1}, ..., \pi_{i,m/k}$, each transformed into a k-permutation $\pi'_{i,j}$.

To share the permutations, P_w needs to share $O(\frac{m \log m}{k \log k})$ many $k \times k$ permutation matrices in total. According to Lemma 5, by parallel calls to PermMat, this is $O(\frac{k}{\log k} nm \log m)$ communication and computation, and $O(\log \log k)$ rounds.

Corollary 1. *Suppose the parties are shuffling vector of vectors, i.e. shuffling* $[\![\mathbf{V}]\!] = [\![\mathbf{v}_1, ..., \mathbf{v}_m]\!]$, *each* $\mathbf{v}_i$ *of length* l.

For our first shuffle protocol described in Algorithm 8 and 9, the offline communication, computation and round complexity are $O(\frac{k}{\log k} n^2 m \log m)$, $O(\frac{k}{\log k} n^2 m \log m)$ *and* $O(\log \log k)$, *respectively. The online complexities are* $O(\frac{l}{\log k} n^2 m \log m)$, $O(\frac{kl}{\log k} n^2 m \log m)$ *and* $O(\frac{n \log m}{\log k})$, *respectively.*

Proof: Note that all parties can share permutation matrices in parallel in the offline phase. Hence, the offline round complexity is simply $O(\log \log k)$.

Other complexities follow naturally from Theorem 1 and 2.

5 Shuffle Protocol with Linear Online Communication

In this section, we give the construction of our second shuffle protocol. We utilize the malicious shuffle correlation proposed by Gao et al. [14], and obtain a shuffle protocol with $O(nml)$ online communication with $O(\frac{k+l}{\log k} n^2 m \log m)$ offline communication. For a clarity of demonstration, below we describe the construction for $l = 1$, i.e. the case of shuffling vector entries.

5.1 Correlation Check of Public Values

The functionality of protocol $\text{CorrCheck}(\mathbf{a}, \mathbf{b}, [\![\beta]\!], [\![\mathbf{r}]\!])$ is to check for P_w whether $\beta\mathbf{a} = \mathbf{b} + \mathbf{r}$. For honest P_w, this protocol aborts w.h.p. if the correlation is not satisfied.

This protocol is for the online phase of the shuffle protocol, where each party P_i will send some $\mathbf{a}$ and $\mathbf{b}$ to P_{i+1}. Party P_{i+1} could use this protocol to make sure that it is receiving correct messages. If honest P_{i+1} passes the check, this means that P_i has sent correct message to P_{i+1}. In [14], it is shown that as long as honest P_w receives correct messages, the permutation chosen by it is protected.

To perform the check, party P_w first locally draws a challenge $\lambda \leftarrow \mathbb{K}$. It computes

$$u \leftarrow \sum_{i=1}^{m} \lambda^{i-1} \mathbf{a}(i), \ v \leftarrow \sum_{i=1}^{m} \lambda^{i-1} \mathbf{b}(i),$$

and broadcasts λ, u, v. Then all parties compute

$$[\![s]\!] \leftarrow [\![\beta]\!]u - v - \sum_{i=1}^{m} \lambda^{i-1}[\![\mathbf{r}(i)]\!],$$

and check if the result is zero. Note that communications happen only at a call to Π_{open} and Π_{mul}, besides the broadcast of λ.

This protocol is formally presented in Algorithm 10.

Algorithm 10 CorrCheck($P_w, \mathbf{a}, \mathbf{b}, [\![\beta]\!], [\![\mathbf{r}]\!]$)

Require: $\mathbf{a}$ and $\mathbf{b}$ are known to P_w. However, they are not secret.
Ensure: If $P_w = \emptyset$ or P_w is honest, abort w.h.p. if $\beta\mathbf{a} \neq \mathbf{b} + \mathbf{r}$.
  Let m be the length of $\mathbf{a}$, $\mathbf{b}$ and $[\![\mathbf{r}]\!]$.
  if $P_w \neq \emptyset$ **then**
    P_w draws uniformly $\lambda \leftarrow \mathbb{K}$.
    P_w computes $u \leftarrow \sum_{i=1}^{m} \lambda^{i-1}\mathbf{a}(i)$.
    P_w computes $v \leftarrow \sum_{i=1}^{m} \lambda^{i-1}\mathbf{b}(i)$.
    P_w broadcasts λ, u, v.
  else ▷ $\mathbf{a}$ and $\mathbf{b}$ are known to all.
    $\lambda \leftarrow \Pi_{\text{challenge}}()$
    All parties locally compute $u \leftarrow \sum_{i=1}^{m} \lambda^{i-1}\mathbf{a}(i)$.
    All parties locally compute $v \leftarrow \sum_{i=1}^{m} \lambda^{i-1}\mathbf{b}(i)$.
  end if
  $[\![s]\!] \leftarrow [\![\beta]\!]u - v - \sum_{i=1}^{m} \lambda^{i-1}[\![\mathbf{r}(i)]\!]$
  $s \leftarrow \Pi_{\text{open}}([\![s]\!])$
  All parties abort if $s \neq 0$.

5.2 Offline Phase

The main task of the offline phase is to generate a shuffle correlation, i.e. to generate some random (secret shared) values satisfying certain correlation. The definition of shuffle protocol is given below.

Definition 2 (m-Shuffle Correlation[14]). *The m-shuffle correlation is defined as*

$$\text{cor} = \left\{ \begin{array}{cccc} [\![\beta]\!] & [\![\mathbf{r}]\!] & [\![\beta\mathbf{r}]\!] & [\![\mathbf{s}]\!] \\ \pi_1 & \pi_2 & \cdots & \pi_n \\ [\![\pi_1(\mathbf{r}'_1)]\!] & [\![\pi_2(\mathbf{r}'_2)]\!] & \cdots & [\![\pi_n(\mathbf{r}'_n)]\!] \\ & \mathbf{z}_2 & \cdots & \mathbf{z}_n \end{array} \right\},$$

where

1. *π_i is an m-permutation known only to P_i. It is chosen by P_i and is uniform over all possible m-permutations if P_i is honest.*
2. *$[\![\beta]\!]$ is a secret shared random variable uniform over $\mathbb{K}$.*

3. $[\![\mathbf{r}]\!], [\![\mathbf{s}]\!], [\![\mathbf{r}'_1]\!], ..., [\![\mathbf{r}'_n]\!]$ *are secret shared vectors of length* m*, with each entry independently uniformly random over* $\mathbb{K}$*.* $[\![\beta\mathbf{r}]\!]$ *is secret shared* $\beta \cdot \mathbf{r}$*.*
4. $\mathbf{z}_i = (\mathbf{z}_{i,1}, \mathbf{z}_{i,2})$*, where* $\mathbf{z}_{i,j}$ *is a vector of length* m*. The entries of* $\mathbf{z}_{i,1}$ *are uniformly random, under the constraint*

$$\begin{cases} \mathbf{z}_{i,2} = \beta\mathbf{z}_{i,1} + \pi_{i-1}(\mathbf{r}'_{i-1}) - \mathbf{r}'_i & \forall i = 2, 3, ..., n \\ \mathbf{s} = \pi_n(\pi_{n-1}(\cdots \pi_2(\pi_1(\mathbf{r}) - \mathbf{z}_{2,1}) - \mathbf{z}_{3,1} \cdots) - \mathbf{z}_{n,1}) \end{cases}$$

To generate such a shuffle correlation, the parties first generate a random $[\![\beta]\!]$ and $2n$ random vectors in $\mathbb{K}$, each of length m. Denote them as

$$[\![\mathbf{r}_1]\!], [\![\mathbf{r}_2]\!], ..., [\![\mathbf{r}_n]\!], [\![\mathbf{r}'_1]\!], [\![\mathbf{r}'_2]\!], ..., [\![\mathbf{r}'_n]\!].$$

The parties then call protocol Π_{mul} and acquire

$$[\![\beta\mathbf{r}_1]\!], [\![\beta\mathbf{r}_2]\!], ..., [\![\beta\mathbf{r}_n]\!],$$

i.e. multiplying each entry by a same factor β.

The parties then call protocol Permute for n times, and acquire for $i \in [n]$

$$[\![\pi_i(\mathbf{r}_i), \pi_i(\beta\mathbf{r}_i), \pi_i(\mathbf{r}'_i)]\!] \leftarrow \text{Permute}(P_i : \pi_i, [\![\mathbf{r}_i, \beta\mathbf{r}_i, \mathbf{r}'_i]\!]).$$

Now the parties compute for each $i \geq 2$

$$\begin{aligned} [\![\mathbf{z}_{i,1}]\!] &\leftarrow [\![\pi_{i-1}(\mathbf{r}_{i-1})]\!] - [\![\mathbf{r}_i]\!], \\ [\![\mathbf{z}_{i,2}]\!] &\leftarrow [\![\pi_{i-1}(\beta\mathbf{r}_{i-1})]\!] + [\![\pi_{i-1}(\mathbf{r}'_{i-1})]\!] - [\![\beta\mathbf{r}_i]\!] - [\![\mathbf{r}'_i]\!]. \end{aligned}$$

Note that $[\![\mathbf{z}_i]\!] \coloneqq ([\![\mathbf{z}_{i,1}]\!], [\![\mathbf{z}_{i,2}]\!])$ is a vector of length $2m$.

The parties then open each $[\![\mathbf{z}_i]\!]$ to P_i, for each $i \geq 2$. And the shuffle correlation returned by this protocol is

$$\text{cor} \coloneqq \left\{ \begin{array}{cccc} [\![\beta]\!] & [\![\mathbf{r}_1]\!] & [\![\beta\mathbf{r}_1]\!] & [\![\pi_n(\mathbf{r}_n)]\!] \\ \pi_1 & \pi_2 & ... & \pi_n \\ [\![\pi_1(\mathbf{r}'_1)]\!] & [\![\pi_2(\mathbf{r}'_2)]\!] & \cdots & [\![\pi_n(\mathbf{r}'_n)]\!] \\ & \mathbf{z}_2 & \cdots & \mathbf{z}_n \end{array} \right\}.$$

Note that each $\mathbf{z}_i$ is held as plaintext by party P_i for $i \geq 2$, and variables with bracket is shared among all parties.

To see that this satisfies the definition of shuffle correlation, note that by substituting the $\mathbf{r}$ and $\mathbf{s}$ in the definition with $\mathbf{r}_1$ and $\pi_n(\mathbf{r}_n)$, all requirements will be satisfied.

This protocol is formally described in Algorithm 11.

5.3 Online Phase

In the online phase of the protocol, when parties are to shuffle $[\![\mathbf{v}]\!]$, the parties first compute

$$[\![\mathbf{z}_1]\!] = ([\![\mathbf{v}]\!] - [\![\mathbf{r}_1]\!], [\![\beta]\!][\![\mathbf{v}]\!] - [\![\beta\mathbf{r}_1]\!] - [\![\mathbf{r}'_1]\!]),$$

Algorithm 11 cor $\leftarrow \text{Shuffle2}_{\text{off}}(P_1 : \pi_1, ..., P_n : \pi_n)$

Require: For honest P_i, π_i is sampled uniformly from all m-permutations.
Ensure: Output a shuffle correlation.

$[\![\beta]\!] \leftarrow \Pi_{\text{rand}}(\mathbb{K})$.
for $i = 1$ to n **do parallel**
  $[\![\mathbf{r}_i(j)]\!] \leftarrow \Pi_{\text{rand}}(\mathbb{K})$ for $j \in [m]$
  $[\![\mathbf{r}'_i(j)]\!] \leftarrow \Pi_{\text{rand}}(\mathbb{K})$ for $j \in [m]$
  $[\![\beta\mathbf{r}_i]\!] \leftarrow \Pi_{\text{mul}}([\![\beta]\!], [\![\mathbf{r}_i]\!])$
  $[\![\pi_i]\!] \leftarrow \text{SharePerm}(k, P_i : \pi_i)$
  $[\![\pi_i(\mathbf{r}_i), \pi_i(\beta\mathbf{r}_i), \pi_i(\mathbf{r}'_i)]\!] \leftarrow \text{Permute}([\![\pi_i]\!], [\![\mathbf{r}_i, \beta\mathbf{r}_i, \mathbf{r}'_i]\!])$
  if $i \geq 2$ **then**
    $[\![\mathbf{z}_{i,1}]\!] \leftarrow [\![\pi_{i-1}(\mathbf{r}_{i-1})]\!] - [\![\mathbf{r}_i]\!]$
    $[\![\mathbf{z}_{i,2}]\!] \leftarrow [\![\pi_{i-1}(\beta\mathbf{r}_{i-1})]\!] - [\![\beta\mathbf{r}_i]\!] + [\![\pi_{i-1}(\mathbf{r}_{i-1})]\!] - [\![\mathbf{r}'_i]\!]$
    $[\![\mathbf{z}_i]\!] \coloneqq ([\![\mathbf{z}_{i,1}]\!], [\![\mathbf{z}_{i,2}]\!])$
  end if
end for
for $i = 2$ to n **do parallel**
  $\Pi_{\text{open}}([\![\mathbf{z}_i]\!], P_i)$
end for
Return cor $\coloneqq \begin{Bmatrix} [\![\beta]\!] & [\![\mathbf{r}_1]\!] & [\![\beta\mathbf{r}_1]\!] & [\![\pi_n(\mathbf{r}_n)]\!] \\ \pi_1 & \pi_2 & ... & \pi_n \\ [\![\pi_1(\mathbf{r}'_1)]\!] & [\![\pi_2(\mathbf{r}'_2)]\!] & \cdots & [\![\pi_n(\mathbf{r}'_n)]\!] \\ & \mathbf{z}_2 & \cdots & \mathbf{z}_n \end{Bmatrix}$.

and open them to P_1. P_1 then computes and sends

$$\mathbf{y}_1 = \pi_1(\mathbf{z}_1)$$

to party P_2. Note that here we slightly abuse the notation, and define

$$\pi_1(\mathbf{z}_1) \coloneqq (\pi_1(\mathbf{z}_{1,1}), \pi_1(\mathbf{z}_{1,2})).$$

This is also defined for other vector of length $2m$, where π is an m-permutation.

Then for $2 \leq i \leq n-1$, P_i first checks if the message from P_{i-1} is correct. This is done by a call to protocol

$$\text{CorrCheck}(P_i, \mathbf{y}_{i-1,1}, \mathbf{y}_{i-1,2}, [\![\beta]\!], [\![\pi_{i-1}(\mathbf{r}'_{i-1})]\!]).$$

If the check fails, all parties abort; otherwise P_i computes and sends to P_{i+1}

$$\mathbf{y}_i = \pi_i(\mathbf{z}_i + \mathbf{y}_{i-1}).$$

The last party P_n, after applying a similar check, computes and broadcasts $\mathbf{y}_n = \pi_n(\mathbf{z}_n + \mathbf{y}_{n-1})$. Then all parties check if

$$\text{CorrCheck}(\emptyset, \mathbf{y}_{n,1}, \mathbf{y}_{n,2}, [\![\beta]\!], [\![\pi_n(\mathbf{r}'_n)]\!]).$$

If the check passes, then the protocol takes

$$\mathbf{y}_n - [\![\pi_n(\mathbf{r}_n)]\!]$$

as the output.

Above process is formally presented in Algorithm 12. To see the correctness of the protocol, note that if all parties are honest, each P_{i+1} will receive

$$\begin{aligned}
\mathbf{z}_{i+1,1} &= \pi_i(\mathbf{r}_i) - \mathbf{r}_{i+1}, \\
\mathbf{z}_{i+1,2} &= \beta\mathbf{z}_{i+1,1} + \pi_i(\mathbf{r}'_i) - \mathbf{r}'_{i+1}, \\
\mathbf{y}_{i,1} &= \pi_i \circ \cdots \circ \pi_1(\mathbf{x}) - \pi_i(\mathbf{r}_i), \\
\mathbf{y}_{i,2} &= \beta\mathbf{y}_{i,1} - \pi_i(\mathbf{r}'_i).
\end{aligned}$$

Hence, it will send $\pi_{i+1}(\mathbf{y}_i + \mathbf{z}_{i+1})$, which is

$$\begin{aligned}
\mathbf{y}_{i+1,1} &= \pi_{i+1} \circ \cdots \circ \pi_1(\mathbf{x}) - \pi_{i+1}(\mathbf{r}_{i+1}), \\
\mathbf{y}_{i+1,2} &= \beta\mathbf{y}_{i+1,1} - \pi_{i+1}(\mathbf{r}'_{i+1}).
\end{aligned}$$

Thus, these invariants are maintained along all P_i, until lastly parties compute $\mathbf{y}_n + \pi_n(\mathbf{r}_n)$, which is exactly $\pi(\mathbf{x})$.

To see security, note that as the adversary does not know β and $\mathbf{r}'$, a corrupted P_i cannot forge wrong $\mathbf{y}_i$ and pass the following correlation check with non-negligible probability. Hence, the fact that the check passes means that correct $\mathbf{y}_i$ is used as input. For honest P_{i+1}, this means that it has received correct $\mathbf{y}_i$ from P_i. For corrupted P_{i+1}, this means that it knows correct $\mathbf{y}_i$, and has used it as input for the checking. This is formally proven in the work of Gao et al. [14].

Algorithm 12 $[\![\mathbf{x}']\!] \leftarrow \text{Shuffle2}_{\text{on}}([\![\mathbf{x}]\!])$

Ensure: Output shuffled $[\![\pi(\mathbf{x})]\!]$ for some secret uniform π.
  Fetch a fresh m-shuffle correlation cor generated by $\text{Shuffle2}_{\text{off}}$.
  $[\![\mathbf{z}_1]\!] \leftarrow ([\![\mathbf{x}]\!] - [\![\mathbf{r}_1]\!], [\![\beta]\!][\![\mathbf{x}]\!] - [\![\beta\mathbf{r}_1]\!] - [\![\mathbf{r}'_1]\!])$
  Open $[\![\mathbf{z}_1]\!]$ to P_1.
  P_1 computes $\mathbf{y}_1 \leftarrow \pi_1(\mathbf{z}_1)$.
  P_1 sends $\mathbf{y}_1$ to P_2.
  for $i = 2$ to n **do**
    $\text{CorrCheck}(P_i, \mathbf{y}_{i-1,1}, \mathbf{y}_{i-1,2}, [\![\beta]\!], [\![\pi_{i-1}(\mathbf{r}'_{i-1})]\!])$
    P_i computes $\mathbf{y}_i \leftarrow \pi_i(\mathbf{y}_{i-1} + \mathbf{z}_i)$.
    if $i < n$ **then**
      P_i sends $\mathbf{y}_i$ to P_{i+1}.
    else
      P_n broadcasts $\mathbf{y}_n$.
    end if
  end for
  $\text{CorrCheck}(\emptyset, \mathbf{y}_{n,1}, \mathbf{y}_{n,2}, [\![\beta]\!], [\![\pi_n(\mathbf{r}'_n)]\!])$
  Return $\mathbf{y}_{n,1} - [\![\pi_n(\mathbf{r}_n)]\!]$.

5.4 Complexity Analysis

Theorem 3. *For* $\text{Shuffle2}_{\text{off}}(P_1 : \pi_1, ..., P_n : \pi_n)$ *described in Algorithm 11, the (offline) communication, computation and round complexity are* $O(\frac{k}{\log k} n^2 m \log m)$, $O(\frac{k}{\log k} n^2 m \log m)$ *and* $O(\frac{\log m}{\log k} + \log\log k)$, *respectively.*

Proof: There are in total $n \cdot m$ many calls to each of Π_{rand}, Π_{mul} and Π_{open}. This requires $O(n^2 m)$ communication, which is not the bottleneck.

The bottleneck of all the complexities is the n calls to protocol SharePerm and Permute, each for permuting an $m \times 3$ matrix. Since these calls are in parallel, by combining the offline and online complexities presented in Theorem 1 and 2, we obtain that the communication, computation and round complexity are $O(\frac{k}{\log k} n^2 m \log m)$, $O(\frac{k}{\log k} n^2 m \log m)$ and $O(\frac{\log m}{\log k} + \log\log k)$, respectively.

Theorem 4. *For* $\text{Shuffle2}_{\text{on}}(\llbracket \mathbf{v} \rrbracket)$ *described in Algorithm 12, the communication, computation and round complexity are* $O(nm + n^2)$, $O(n^2 m)$ *and* $O(n)$, *respectively. Note that for* $n = O(m)$, *the communication is just* $O(nm)$.

Proof: Note that protocol CorrCheck, when called for vector of length m, costs $O(n)$ communication, $O(nm)$ computation and $O(1)$ round. Besides calls to CorrCheck, parties compute Π_{mul} over m-long vector, and each party P_i sends $\mathbf{y}_i$ of length $O(m)$ to P_{i+1}. P_n lastly broadcasts $\mathbf{y}_n$, which requires $O(nm)$ communication.

Summing all, the communication complexity is $O(n(n+m))$, the computation complexity is $O(n^2 m)$ due to n calls to CorrCheck and round complexity is $O(n)$.

Corollary 2. *For shuffling rows of* $m \times l$ *matrix, the shuffle protocol described in Algorithm 11 and 12 requires* $O(nml + n^2)$ *online communication,* $O(n^2 ml)$ *online computation and* $O(n)$ *online rounds, with an offline overhead of* $O(\frac{k+l}{\log k} n^2 m \log m)$ *communication,* $O(\frac{kl}{\log k} n^2 m \log m)$ *computation and* $O(\frac{\log m}{\log k} + \log\log k)$ *rounds.*

This concludes our main results in this section.

6 Discussion

6.1 Sharing Permutation Matrix

Recall that the parties wish to perform computation in some field $\mathbb{F}$ with prime subfield $\mathbb{F}_p$. Suppose party P_w needs to share an $m \times m$ permutation matrix. When p is larger than m, generating and checking permutation matrix can be much easier.

To generate a (purported) permutation matrix chosen by party P_w, the protocol simply requires P_w to share it. This matrix will be a permutation matrix, if the following conditions are satisfied:

1. Each entry of the matrix is either 0 or 1.
2. Sum along its row is always 1.

3. Sum along its column is always 1.

To see the reason, note that the first condition makes the matrix Boolean. Since $p > m$, sum along the row will not overflow, which makes the sum exactly the number of 1s. Hence, since this is a Boolean matrix with exactly 1 on each row and column, this is a permutation matrix. Also, the first condition can be checked by a single call to Π_{inner}, and all conditions can be batched into a single Π_{open}. Thus, this check has almost the same overhead as PermCheck protocol.

However, above argument is valid only when $p > m$. A counterexample is when $m = p + 1$, a malicious party can share an all one matrix and pass all the check, since $1 \equiv p + 1$ in such a field. In the case of $p \leq m$, checking whether an arbitrary matrix is a permutation matrix seems much harder.

Hence, in our construction, we pose specific constraint (via one-hot vector) to make such a check easier in general case. The communication complexities of the two approaches are equal asymptotically, both of which are $O(nm^2)$ for generating and checking one permutation matrix.

6.2 Online Complexity

There are several reasons for the online complexity of an MPC protocol to be much more important than the offline complexity. One is that, for example, if we construct anonymous communication system based on an MPC shuffle (e.g. Clarion by [4]), the online complexity stands for the latency that users/clients will experience. Similar arguments can also be made for other real-world MPC applications.

Another profound reason is that, the offline phases of multiple protocol sessions can be done in parallel. This enables a significant reduction in round complexity of the protocol. For example, suppose the parties wish to sequentially perform shuffle for k times, where each execution takes R_{off} rounds in offline phase and R_{on} rounds online. Then the total computation will require R_{off} offline rounds and $k \cdot R_{\text{on}}$ online rounds, via a naive parallelization of the data-independent operations. This could be even more significant if the protocol serves as a sub-routine of some high-level protocol. A concrete example is the MPC sorting algorithm by Hamada et al. [25], which requires MPC shuffle as a primitive. To sort data by 128-bit index, 128 sequential calls to shuffle protocol are required, which requires at least $R_{\text{off}} + 128 \times R_{\text{on}}$ rounds. Since only the online phase will be inevitably embedded (as subroutine) into higher-level protocol, only the online round complexity is subject to such an amplification. This makes the online round complexity much more important than the offline one.

6.3 Definition of Shared Permutation

The requirement of $s = O(\frac{\log m}{\log k})$ in Definition 1 is tight. To see this, note that there are $m!$ many m-permutations. The number of possible k-shared m-permutations in above definition is at most $(k!)^{ms/k}$, which must be no smaller

than $m!$. Hence,

$$s \geq \log_{(k!)^{m/k}}(m!) = \frac{k \log(m!)}{m \log(k!)}.$$

By noting $m! = \Theta(m \log m)$, we obtain $s = \Omega(\frac{\log m}{\log k})$.

Note that one may also consider a recursive definition, where $[\![\mathbf{P}_{i,j}]\!]$ is replaced by recursive $[\![\pi_{i,j}]\!]$. The recursion halts when some $[\![\pi]\!]$ is shared exactly as $[\![\mathbf{P}_\pi]\!]$. Nevertheless, for our purpose, the present definition suffices.

Bibliography

[1] Ran Canetti. Security and composition of multiparty cryptographic protocols. *J. Cryptol.*, 13(1):143–202, January 2000. ISSN 0933-2790. `https://doi.org/10.1007/s001459910006`. URL `https://doi.org/10.1007/s001459910006`.

[2] Nikolaos Alexopoulos, Aggelos Kiayias, Riivo Talviste, and Thomas Zacharias. {MCMix}: Anonymous messaging via secure multiparty computation. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1217–1234, 2017.

[3] Donghang Lu, Thomas Yurek, Samarth Kulshreshtha, Rahul Govind, Aniket Kate, and Andrew Miller. Honeybadgermpc and asynchromix: Practical asynchronous mpc and its application to anonymous communication. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 887–903, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450367479. URL `https://doi.org/10.1145/3319535.3354238`.

[4] Saba Eskandarian and Dan Boneh. Clarion: Anonymous communication from multiparty shuffling protocols. *Network and Distributed System Security Symposium*, 2021.

[5] Ralf Küsters, Julian Liedtke, Johannes Müller, Daniel Rausch, and Andreas Vogt. Ordinos: A verifiable tally-hiding e-voting system. In *2020 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 216–235. IEEE, 2020.

[6] Wang Jingzhong, Zhang Yue, and Li Haibin. Electronic voting protocol based on ring signature and secure multi-party computing. In *2020 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC)*, pages 50–55. IEEE, 2020.

[7] Allan B Pedin IV and Nazli Siasi. Secure and decentralized anonymous e-voting scheme. In *Proceedings of the 2023 ACM Southeast Conference*, pages 172–176, 2023.

[8] Yanping Zhang, Johes Bater, Kartik Nayak, and Ashwin Machanavajjhala. Longshot: Indexing growing databases using mpc and differential privacy. *Proceedings of the VLDB Endowment*, 16(8):2005–2018, 2023.

[9] Chenghong Wang, Johes Bater, Kartik Nayak, and Ashwin Machanavajjhala. Incshrink: architecting efficient outsourced databases using incremental mpc and differential privacy. In *Proceedings of the 2022 International Conference on Management of Data*, pages 818–832, 2022.

[10] Koki Hamada, Ryo Kikuchi, Dai Ikarashi, Koji Chida, and Katsumi Takahashi. Practically efficient multi-party sorting protocols from comparison sort algorithms. In *Information Security and Cryptology–ICISC 2012: 15th International Conference, Seoul, Korea, November 28-30, 2012, Revised Selected Papers 15*, pages 202–216. Springer, 2013.

[11] Marcel Keller and Peter Scholl. Efficient, oblivious data structures for MPC. In Palash Sarkar and Tetsu Iwata, editors, *Advances in Cryptology – ASIACRYPT 2014*, pages 506–525. Springer, 2014. ISBN 978-3-662-45608-8.

[12] Melissa Chase, Esha Ghosh, and Oxana Poburinnaya. Secret-shared shuffle. In *Advances in Cryptology–ASIACRYPT 2020: 26th International Conference on the Theory and Application of Cryptology and Information Security, Daejeon, South Korea, December 7–11, 2020, Proceedings, Part III 26*, pages 342–372. Springer, 2020.

[13] Peeter Laud. Linear-Time Oblivious Permutations for SPDZ. In Mauro Conti, Marc Stevens, and Stephan Krenn, editors, *Cryptology and Network Security*, pages 245–252, Cham, 2021. Springer International Publishing. ISBN 978-3-030-92548-2.

[14] Jiacheng Gao, Yuan Zhang, and Sheng Zhong. Multiparty shuffle: Linear online phase is almost for free. Cryptology ePrint Archive, Paper 2024/1936, 2024. URL `https://eprint.iacr.org/2024/1936`.

[15] Dhaneshwar Mardi, Surbhi Tanwar, and Jaydeep Howlader. Multiparty protocol that usually shuffles. *Security and Privacy*, 4(6):e176, 2021.

[16] Mahnush Movahedi, Jared Saia, and Mahdi Zamani. Secure multi-party shuffling. In *Structural Information and Communication Complexity: 22nd International Colloquium, SIROCCO 2015, Montserrat, Spain, July 14-16, 2015. Post-Proceedings 22*, pages 459–473. Springer, 2015.

[17] Mahnush Movahedi, Jared Saia, and Mahdi Zamani. Shuffle to baffle: Towards scalable protocols for secure multi-party shuffling. In *2015 IEEE 35th International Conference on Distributed Computing Systems*, pages 800–801. IEEE, 2015.

[18] Sven Laur, Jan Willemson, and Bingsheng Zhang. Round-efficient oblivious database manipulation. In *Information Security: 14th International Conference, ISC 2011, Xi'an, China, October 26-29, 2011. Proceedings 14*, pages 262–277. Springer, 2011.

[19] Xiangfu Song, Dong Yin, Jianli Bai, Changyu Dong, and Ee-Chien Chang. Secret-shared shuffle with malicious security. *Network and Distributed System Security Symposium*, 2023.

[20] David L Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Communications of the ACM*, 24(2):84–90, 1981.

[21] Ben Adida and Douglas Wikström. How to shuffle in public. In *Theory of Cryptography: 4th Theory of Cryptography Conference, TCC 2007, Amsterdam, The Netherlands, February 21-24, 2007. Proceedings 4*, pages 555–574. Springer, 2007.

[22] Jens Groth. A verifiable secret shuffle of homomorphic encryptions. *Journal of Cryptology*, 23:546–579, 2010.

[23] Susan Hohenberger, Guy N Rothblum, Abhi Shelat, and Vinod Vaikuntanathan. Securely obfuscating re-encryption. In *Theory of Cryptography Conference*, pages 233–252. Springer, 2007.

[24] Masayuki Abe. Mix-networks on permutation networks. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 258–273. Springer, 1999.

[25] Koki Hamada, Dai Ikarashi, Koji Chida, and Katsumi Takahashi. Oblivious radix sort: An efficient sorting algorithm for practical secure multi-party computation. *Cryptology ePrint Archive*, 2014.

[26] Dan Bogdanov, Sven Laur, and Riivo Talviste. A practical analysis of oblivious sorting algorithms for secure multi-party computation. In *Nordic Conference on Secure IT Systems*, pages 59–74. Springer, 2014.

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

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

[29] Tom Leighton and C Greg Plaxton. A (fairly) simple circuit that (usually) sorts. In *Proceedings [1990] 31st Annual Symposium on Foundations of Computer Science*, pages 264–274. IEEE, 1990.

[30] Václav E Beneš. Permutation groups, complexes, and rearrangeable connecting networks. *Bell System Technical Journal*, 43(4):1619–1640, 1964.

[31] Vipul Goyal, Yifan Song, and Chenzhi Zhu. Guaranteed output delivery comes free in honest majority mpc. In *Annual International Cryptology Conference*, pages 618–646. Springer, 2020.

[32] John Launchbury, Iavor S Diatchki, Thomas DuBuisson, and Andy Adams-Moran. Efficient lookup-table protocol in secure multiparty computation. In *Proceedings of the 17th ACM SIGPLAN international conference on Functional programming*, pages 189–200, 2012.

A Security Proofs

In this section, we analyze our protocols and prove that the shuffle protocols are secure. By the merit of ideal functionality $\mathcal{F}_{\text{Shamir}}$ and the work of Gao et al. [14], the security is obtained almost immediately from definitions. Nevertheless, we will provide in this section some insights about security of our protocols.

To simplify our analysis, we focus on detecting the misbehavior of the adversary. That is, we prove that if the adversary deviates from the protocol, the protocol will abort with overwhelming probability. This approach is justified by noting that, in standard simulation based security, one must construct a simulator that is able to simulate the view of the adversary, while being ignorant of the input of honest parties. This simulation, however, is almost always trivial in Shamir secret sharing, because the simulator stands for all honest parties, which makes it able to reconstruct all values shared from the adversary. Hence, in a typical proof, the simulator simply sends random values to adversary as its received messages, aborts if any check fails, and lastly modifies the shares for honest parties to make them consistent with the output from ideal functionality. This means that we need to only argue that any misbehavior will lead to abort with overwhelming probability, so that the output of the simulator will coincide with real distribution. Note that all basic operations of our construction are via ideal functionalities, which gives us (seemingly extremely) powerful security guarantee, e.g. even introducing error in Π_{add} will lead to immediate abort. However, such a strong condition is justified by composition theorem, which states that a protocol remains to be secure after replacing ideal functionalities with their secure implementations. Hence, any misbehavior against ideal functionalities will be detected, and we are left to only prove that misbehavior outsides the ideal functionalities will be detected.

Hence, below, we focus on arguing that a protocol will end with one of the two cases. One is that the adversary misbehaves, and the protocol aborts with overwhelming probability. Another is that all parties act honestly, and the honest parties receive correct shares.

As one of the results by Gao et al. [14], it is proved that if protocol Permute is secure then protocol $\text{Shuffle2}_{\text{off}}$ and $\text{Shuffle2}_{\text{on}}$ are secure. This is stated in the Theorem 5 below.

Theorem 5 ([14], informal). *Consider an ideal functionality*

$$[\![\pi(\mathbf{v})]\!] \leftarrow \Pi_{\text{perm}}(P_w : \pi, [\![\mathbf{v}]\!]),$$

which takes secret input π from P_w, and securely permutes $[\![\mathbf{v}]\!]$ as $[\![\pi(\mathbf{v})]\!]$.

If protocol Permute *securely implements Π_{perm}, then* $\text{Shuffle2}_{\text{off}}$ *and* $\text{Shuffle2}_{\text{on}}$ *are secure by the composition theorem. This means that misbehavior of corrupted parties will make all parties abort with overwhelming probability, and if the protocol does not abort, with overwhelming probability $\mathbf{v}$ is correctly shuffled and shared among honest parties, with adversary learning no information about the permutation applied.*

This theorem allows us to focus on the security of Permute protocol. Since Permute is only calls to basic primitives and earlier developed protocols, it suffices to prove the security of its sub-protocols.

A.1 Security of One-hot Protocol

The security of one-hot vector sharing protocol follows almost directly from basic primitives, which we briefly review here.

Lemma 6. *The protocol* BoolCheck($[\![b_1]\!], ..., [\![b_d]\!]$) *described in Algorithm 2 is secure, in the sense that if any input is not Boolean, it aborts with probability*

$$p \geq 1 - \frac{d-1}{2^\kappa}.$$

Proof: Since the computation is done via ideal functionalities Π_{open} and Π_{inner}, upon disclosure, the parties receive correctly

$$u = \sum_{i=1}^{d} b_i \cdot (1 - b_i) \cdot \lambda^{i-1}.$$

Since λ is generated by $\Pi_{\text{challenge}}$, it is uniform over $\mathbb{K}$. Note that in field $\mathbb{K}$, product $b_i(1-b_i)$ is zero if and only if one of them is zero. As any degree $d-1$ polynomial has at most $d-1$ roots in field $\mathbb{K}$, λ happens to be one of the roots with probability

$$p_{\text{fail}} \leq \frac{d-1}{|\mathbb{K}|} \leq \frac{d-1}{2^\kappa}.$$

Hence, u is nonzero with probability at least

$$p = 1 - p_{\text{fail}} \geq 1 - \frac{d-1}{2^\kappa}.$$

Lemma 7. *If all inputs are shared Boolean, the protocol* Demux($[\![b_1]\!], ..., [\![b_d]\!]$) *described in Algorithm 3 is secure, in the sense that if the protocol does not abort, honest parties share* $[\![\mathbf{s}]\!]$ *as a one-hot vector whose index is* $1 + \sum_{i=1}^{d} 2^{i-1} b_i$.

Proof: The basic case is $d = 1$, where all honest parties hold

$$[\![\mathbf{s}]\!] = ([\![1 - b_1]\!], [\![b_1]\!]),$$

which is both correct and secure.

For $d > 1$, the protocol is in essence computing a tensor product of two one-hot vector, with calls to ideal functionality Π_{mul}. The correctness and security hence follow trivially.

Theorem 6. *The protocol* OneHot(k, P_w : ind) *described in Algorithm 4 is secure, in the sense that upon normal exit, honest parties hold a valid one-hot vector, with index chosen by* P_w.

In addition, if P_w *is honest, then the corrupted parties will not learn any information about the chosen index.*

Proof: If P_w is honest, it is clear that upon normal exit, the result is a one-hot vector with index chosen by P_w. Further, in the protocol BoolCheck, since the opened value will be zero, which reveals nothing but that the vector is a Boolean one, which is trivial.

Suppose now P_w is malicious and shares $[\![b'_i]\!]$ during protocol OneHot. If any $b'_i \notin \{0,1\}$, then BoolCheck will fail with overwhelming probability due to Lemma 6. Otherwise, if all $b'_i \in \{0,1\}$, then there is a valid index $\text{ind}' = \sum_{i=1}^{k} 2^{i-1} b'_i + 1$ underlies the generated vector. As P_w is free to choose the index, this does not harm the security.

A.2 Security of Permutation Protocol

Theorem 7. *If the input k one-hot vectors do not form a valid $k \times k$ permutation matrix, protocol* PermCheck($[\![\mathbf{P}]\!]$) *described in Algorithm 6 aborts with probability*

$$p \geq 1 - \frac{k-1}{2^\kappa}.$$

Proof: Note that the parties are computing

$$[\![\text{sum}]\!] \leftarrow \sum_{j=1}^{k} \lambda^{j-1} \left(1 - \sum_{i=1}^{k} [\![\mathbf{P}(i,j)]\!]\right).$$

As these arithmetic operations are provided by ideal functionality, we may assume the parties always get correct $[\![\text{sum}]\!]$, unless the protocol already aborts due to misbehavior.

Hence, $[\![\text{sum}]\!]$ is (shared) zero if $\mathbf{P}$ is a well-formed permutation matrix. If $\mathbf{P}$ is not a permutation matrix, then at least one of the coefficients is 1, which makes $[\![\text{sum}]\!]$ a non-zero polynomial of λ of degree at most $k-1$. This means that the polynomial has at most $k-1$ roots in field $\mathbb{K}$. Since λ is uniform over $\mathbb{K}$, λ happens to be one of the roots with probability

$$p_{\text{fail}} \leq \frac{k-1}{|\mathbb{K}|} \leq \frac{k-1}{2^\kappa}.$$

Stated otherwise, if $\mathbf{P}$ is not a permutation matrix, $[\![\text{sum}]\!]$ is not zero with probability

$$p = 1 - p_{\text{fail}} \geq 1 - \frac{k-1}{2^k}.$$

Thus, the protocol aborts with such a probability if $\mathbf{P}$ is not a permutation matrix.

Corollary 3. *Protocol* PermMat *described in Algorithm 5 is secure, in the sense that if all parties act honestly, honest parties share a valid permutation matrix $[\![\mathbf{P}]\!]$, with permutation chosen by and known to P_w. And if the malicious adversary misbehaves, the protocol aborts with overwhelming probability.*

Also, if P_w is honest, then the corrupted parties cannot learn any information about $\mathbf{P}$.

Proof: This is immediate by combining Theorem 6 and Theorem 7.

Corollary 4. *Protocol* SharePerm *described in Algorithm 7 is secure, in the sense that any misbehavior of adversary will lead to abort with overwhelming probability. And if all parties act honestly, honest parties will share* $[\![\pi]\!]$ *with* π *chosen by and known to* P_w*.*

In addition, if P_w *is honest, the adversary cannot learn information about* π*.*

Proof: By Corollary 3, if the protocol hasn't aborted, the matrices are valid permutation matrices, which means the permutation is correctly shared.

Corollary 5. *Protocol* Permute($[\![\pi]\!], [\![\mathbf{v}]\!]$) *described in Algorithm 1 is secure, in the sense that any misbehavior of adversary will lead to abort with overwhelming probability. And if all parties act honestly, honest parties will share* $[\![\pi(\mathbf{v})]\!]$*.*

This follows trivially by the definition of shared permutation and the security of ideal functionalities.

This concludes the security of our permutation protocol.

A.3 Security of Shuffle Protocol

Lastly, we conclude the security of shuffle protocol by combining Theorem 5 and Corollary 4 and 5.

Corollary 6. *The* $\text{Shuffle1}_{\text{off}}$ *and* $\text{Shuffle1}_{\text{on}}$ *prototocols described in Algorithm 8 and 9 are secure, in the sense that they aborts w.h.p. if the adversary misbehaves. Otherwise, the protocols shuffle* $[\![\mathbf{V}]\!]$ *by a secret permutation uniformly over all* m*-permutations.*

Proof: This is a direct result from Corollary 4 and 5.

Corollary 7. *The* $\text{Shuffle2}_{\text{off}}$ *and* $\text{Shuffle2}_{\text{on}}$ *protocols described in Algorithm 11 and 12 are secure, in the sense that they aborts w.h.p. if the adversary misbehaves. Otherwise, the protocols shuffle* $[\![\mathbf{v}]\!]$ *by a secret permutation uniformly over all* m*-permutations.*

Proof: This is obtained directly from Theorem 5 and Corollary 4 and 5.